

FALKSTER.COM

The PM Handbook

How product gets built in the AI era.
Prototypes, evals, agents, and the margin.



Falk Gottlob

FREE · [FALKSTER.COM/EBOOK](https://falkster.com/ebook)

The PM Handbook

How product gets built in the AI era

Falk Gottlob

2026-10-06

About this book

Part 1: The Thesis

Why This Exists

The AI Product Operating Model

Kill the Roadmap

Old PM vs Product Builder, The Ledger

Kill the Status Meeting

Engineering Builds the Substrate, Not Features

The PM 30/60/90

Part 2: Discover

Continuous Discovery

Build Your First Opportunity Solution Tree

The Interview Guide That Actually Works

The Assumption Testing Playbook

Continuous Listening: Every Customer, Every Day

Part 3: Build & Ship

Prototype Before You Spec

The Impact Loop

The Eval Is The Spec

Building Evals: Error Analysis, Eval Types, the Loop, the Gate

Ship With Observability or Don't Ship

The Deprecation Playbook

Incident Response Is a PM Ritual

Agent-to-Agent Dispatch

Part 4: AI Craft

PM AI Agent Fleet, Mapped to the 7-Stage Operating System

When Not to Use AI

Gross Margin Is Your Job Now

Pricing for AI Products

Prompt Ops

The Living Changelog

Trust, Safety, and the Guardrail as a Product Decision

The PM Agent Stack: Open-Source Tools Mapped to PM Work

Direction Metrics for AI-Native Velocity

Part 5: Lead the Shift

Your Weekly Playbook

Strategy From Signals, Not Slides

The Anti-Backlog

The Builder PM 30/60/90

Hiring the Builder PM

PM-as-Editor: Managing a Fleet of Agents

The Cannibalization Decision Framework

Dual Transformation: Running Two Clocks

Pricing Migration: The 18-Month Quarterly Playbook

The Investor and Board Narrative for AI-Era SaaS

The Pricing-Tier Sunset Playbook

The CPO 30/60/90

The Skill Stack: What PMs and CPOs Must Learn Now

Part 6: Forward Deployed

The Forward Deployed Engineer Is a Product Builder in Disguise

FDE vs Sales Engineer: The Job Starts Where the Contract Is Signed

Becoming an FDE: The Transition From PM, Engineer, or SE

What Has to Change in the Product Before FDEs Can Work

The Deployment-to-Product Loop: How FDE Work Becomes Roadmap

FDE Economics: Margins, Outcome Pricing, and When to Stop

Building the FDE Team: Reporting Line, Bar, Pairing, and Comp

Keep going

About this book

This is the PM Handbook from <https://falkster.com/handbook>, compiled into one file on 2026-10-06: 49 chapters across 6 parts. It is the same text as the website. The web version is the living one; chapters get updated as the practice changes, and every chapter here links back to its page.

It is free. Share it with your team. If you quote it, link to the chapter.

The six skills it is organized around are defined in the Product Builder Standard at <https://falkster.com/pm-standard>. The templates every chapter mentions are at <https://falkster.com/toolkit>, and the agent blueprints at <https://falkster.com/handbook/ai-agent-army>.

Falk Gottlob

Part 1: The Thesis

Why the PM role was rewritten.

Why This Exists

The backstory: why I started documenting how I work, what I've learned so far, and what I'm still figuring out.

Web version: <https://falkster.com/handbook/manifesto> · last updated 2026-06-12

The short version

The PM role is splitting into two tracks. Traditional PMs write specs and manage process. Product Builders use AI to collapse the distance between customer insight and working product. Builders prototype in hours using tools like Claude Code, validate with real customers the same week, and ship faster because they're learning continuously instead of planning quarterly. Three practices made the biggest difference in my own work: talking to customers every week (not quarterly), building clickable prototypes before writing any spec, and using AI agents for the mechanical work (monitoring dashboards, drafting reports, summarizing calls) so the actual PM job gets more time. Falkster.com is where I document what's working at the intersection of product management and AI, organized around the four areas I keep returning to: the Product Operating Model, Continuous Discovery, Outcome Orientation, and AI-Native Execution.

The thing that bugged me

Twenty-plus years building products. [Microsoft Research](#), Adobe, Salesforce, four startups (including one 6.5b exit, and one acquired by Microsoft), CPO at Commure, Crisis Text Line, SOCi, and Smartcat. Across all of that, I kept spending my weeks doing things that felt productive but weren't moving the needle. Specs nobody read carefully. Meetings that should've been async. Manually checking dashboards. Roadmap decks that stakeholders forgot by Monday. Sounds like you?

Here's what's changed: AI can now do most of that mechanical work. Not in theory, in practice. Agents that pull data, draft reports, monitor dashboards, summarize meetings, triage feedback, and flag anomalies. The technology is here. The question is whether PMs will use it, or keep grinding through the 80% manually while competitors don't.

I know I'm not the only one. Most PMs I talk to say the same thing: 80% of their week goes to mechanical work that creates zero customer value. The actual PM job, understanding customers, making good calls, figuring out what to build next, gets squeezed into whatever time is left.

That bugged me. So I started experimenting.

What changed

Three shifts happened around the same time:

I started talking to customers every week. Not quarterly research projects. Not NPS surveys. Real conversations, every week, even when they were just 20 minutes. My product decisions got better almost immediately. I stopped guessing. Credit to [Teresa Torres](#) for the framework. Her work on continuous discovery gave me the structure to make it sustainable.

I started prototyping instead of spec'ing. Instead of two weeks on a requirements doc, I'd build a clickable prototype in a couple hours and show it to customers. Feedback was richer, faster, more honest. Some of my best product decisions came from prototypes customers hated, because I learned that in a day instead of after three months of development.

Then I started using AI agents for the mechanical work. Not a chatbot. Actual autonomous agents running on a schedule, reading my tools, delivering reports. Clunky at first. The first two weeks of agent reports were mostly noise. But after tuning, they changed my mornings. I went from 45 minutes of dashboard-checking to a 5-minute report scan.

And then I realized agents change the product itself, not just the process. When you can ship an AI agent that does in seconds what used to take a human analyst hours, you're not optimizing a feature, you're rethinking the value proposition. The companies that figure this out first don't just save costs. They build moats. The product IS the agent. The platform IS the orchestration layer. This is the biggest shift in product strategy since SaaS ate on-premise.

None of these ideas are original. I borrowed from the [Product Operating Model](#) (Marty Cagan, SVPG), [Continuous Discovery Habits](#) (Torres), outcome-driven planning ([Reforge](#), John Cutler, many others), and the AI agent ecosystem ([Anthropic](#), [OpenAI](#), the [Model Context Protocol](#)). What I'm trying to do is document how I put them together in practice, and share what's actually working at the intersection of product management and AI.

Working through this at your company? I do a small number of product org audits each quarter where I write the honest assessment and a 90-day plan against the Product Builder Standard. [See current openings](#) →

But even that feels dated now

Everything I just described, the continuous discovery, the prototyping, the AI agents doing mechanical work, that was the *first* wave. We've entered something different. **The Era of the Product Builder.**

Here's what changed: PMs can now build. Not “build” as in write a spec and hand it off. Build as in sit down with an AI coding agent, describe what the customer needs, feed it your source code, your design system, your product context, and have a working prototype in hours instead of sprints. The gap between “idea” and “something a customer can touch” has collapsed.

This isn't a marginal improvement. It's a role redefinition. The PM who can prototype with AI isn't just faster, they think differently. They test assumptions in real products instead of slide decks. They ship experiments instead of requesting engineering bandwidth. They show customers working software instead of wireframes. The feedback loops shrink from weeks to hours.

And it goes beyond prototyping. The Product Builder generates their own Claude Code prompts from customer requirements, actual source code, designs, and product context. They orchestrate agents that auto-analyze discovery calls, monitor production metrics, and flag when the data contradicts the roadmap thesis. They don't manage a backlog, they build the thing, validate it, and then decide whether engineering should scale it.

The PM role isn't dying. It's splitting. There will be PMs who operate the way we have for thirty years, writing specs, attending standups, managing stakeholders. And there will be Product Builders who collapse the distance between customer insight and working product to near-zero. The second group will ship 10x more validated ideas, and the market will notice.

For the row-by-row breakdown of what changes between the two, see [Old PM vs Product Builder, The Ledger](#), the founder/CEO comparison of output, cycle time, unit of work, deliverable, and accountability across the role rewrite.

If you're reading this and thinking “that sounds like what I want to become”, that's exactly who this site is for.

What this site is

My working notebook, organized around four areas I keep coming back to:

AI Product Operating Model. Empowered teams, product trios, owning problems instead of feature requests. Now add AI to the mix: which decisions can agents make autonomously? Where do humans stay in the loop? How do you structure a team when half the “work” is done by models? Still working a lot of this out.

Continuous Discovery. Weekly customer conversations, opportunity mapping, assumption testing. Now supercharged with AI: agents that auto-summarize transcripts, extract patterns across dozens of calls, and flag when assumptions are invalidated by new data. Still the single highest-ROI practice. AI just made it 10x faster.

Outcome Orientation. OKRs that measure behavior changes, not feature launches. AARRR dashboards. QBRs that tell a real story. AI agents now monitor these metrics continuously and surface anomalies before the weekly review. Writing good OKRs is still hard. But knowing when they're off track is now instant.

AI-Native Execution. The 65 agents we are building and shipping, the MCP setup, the scheduling, the tuning. This is the frontier. Most experimental area. Things break. Agents hallucinate. But the time savings are real when it works, and getting more real every week.

The AI Toolkit. The actual agents, prompts, and workflows I use daily. Prototype Prompt Generators that take customer requirements, your source code, your design system, and your product context, and produce a Claude Code prompt that spins up a working prototype in hours. Discovery agents that auto-analyze interview transcripts. Metric monitors that flag anomalies before standup. These aren't theoretical, they're the tools I build with and share here so you can fork them.

Who this is for

Two audiences:

Senior IC PMs who want to up their personal practice. Talk to more customers, make better decisions, automate the grind, do more impactful work. If you're a PM with 3-7 years of experience and feel like you should be operating at a higher level, a lot of this will land.

PM leaders (Directors, VPs, CPOs) who want better systems across their teams. Different challenges at this level. You're not just running discovery yourself, you're trying to get 5 teams to run it. I share what's worked at Smartcat and where I've struggled.

How to use it

Start wherever you want. There's a suggested order in the handbook, but most people should start with whatever problem they're feeling right now:

- Disconnected from customers? Start with [Continuous Discovery](#).
- Drowning in status updates? Start with [Your AI Agent Fleet](#).
- Shipping features but not moving metrics? Start with [The Impact Loop](#).
- Want to prototype faster? Start with [Prototype Before You Spec](#).
- Want to build with AI agents? Start with [Your AI Agent Fleet](#) and the toolkit section.

Everything here is a living doc. I update it as I learn. Some of what I wrote three months ago I'd write differently today. That's the point.

If something helps, or you've found a better way, tell me. Most of the best ideas here started as conversations with other PMs.

The AI Product Operating Model

What worked before AI, what's breaking now, and how I'm rewiring my practice.

Web version: <https://falkster.com/handbook/product-operating-model> · last updated 2026-07-21

I've been through two big shifts in how product gets done. First was waterfall to agile. Sequential handoffs to cross-functional teams. That took most companies a decade.

The second one is happening now: human-only teams to human-plus-AI. This one's going faster. PMs who don't adjust how they operate are going to wake up doing a job that doesn't exist the way they learned it.

Here's how I think about the product operating model. What worked before, what's breaking, and what I'm building toward.

For the row-by-row ledger of what changes between the old PM role and the Product Builder role, see [Old PM vs Product Builder, The Ledger](#), built for founders and CEOs trying to figure out whether to rewire the product org.

The short version

The AI product operating model has three phases. The pre-AI foundation (Marty Cagan's empowered teams, Teresa Torres's continuous discovery, the PM-designer-engineer trio) was correct and its core principles carry forward: customer obsession, small empowered teams, outcome measurement. What AI breaks: the PRD is dead because a working prototype is faster to build than a spec to write, the trio is becoming a quartet or a duo as role boundaries blur, discovery compresses from four weeks to four days when agents synthesize signals and generate prototypes automatically, and execution overhead (tickets, dashboards, status updates, release notes) drops from 40% of PM time to near zero. What the new model looks like: Monday starts with AI-synthesized insights instead of manual report-pulling, Wednesday produces working clickable prototypes instead of wireframes, Friday ships the first iteration instead of planning the next sprint. The biggest shift: Friday went from "plan what to build" to "ship what you built." That's not a tweak; that's a different operating rhythm.

Part 1: The Operating Model Before AI

This is the model most of us grew up with. [Marty Cagan's](#) empowered teams, [Teresa Torres'](#) continuous discovery, the product trio. If you've been doing PM for a few years, this should feel familiar. If not, you need this foundation before anything else makes sense.

The Three Core Shifts

From output to outcomes. Stop measuring by features shipped. Start measuring by problems solved for customers. Sounds simple. Changes everything about how you plan your week.

From sequential handoffs to concurrent discovery. Instead of product defining the problem, handing it to design, who hands it to engineering, everyone works together *during* discovery. Before anything is “spec’d.”

From backlog prioritization to outcome alignment. Instead of managing a list of features fighting for priority, you align teams around outcomes they own.

When you actually live this model, you spend less time in estimation meetings and more time understanding why customers do what they do. You ship less “stuff” but what you ship moves the needle. You become someone who can say what success looks like, not just what’s on the list.

The Product Trio

PM + designer + engineer(s), working together. Most orgs say they have trios. What they really have is a PM who takes decisions to design and engineering one at a time.

A real trio works like this:

Monday, the three of you spend an hour exploring the customer problem. You bring customer clips, transcripts, or data. Your designer brings patterns or sketches. Your engineer says what’s feasible and what constraints matter. You walk out with a shared understanding and some hypotheses. Not a spec. Not a wireframe.

Then you work in parallel but stay in sync. Designer explores options. Engineer spikes on technical risk. You do more customer validation. Check in every two days.

When you reconvene, decisions happen fast because everyone has context. No formal review. No spec approval meeting. Just: “Given what we know about feasibility and what customers told us, which way?”

The Weekly Cadence (Pre-AI)

Monday - Problem discovery sync (1 hour). PM, designer, and engineer sit down together. Walk out with a shared understanding of the problem and some hypotheses. Not a spec. Not a wireframe.

Tuesday + Wednesday - Parallel work. Designer explores 2-3 approaches. Engineer runs a tech spike and maps constraints. PM does customer validation and brings back data. Everyone works solo but stays loosely in sync.

Wednesday - Mid-week sync (30 min). Direction confirmed, blockers surfaced. Quick and focused.

Thursday - Decision sync (45 min). “This is what we’re building and why.” Everyone has context from their parallel work, so decisions happen fast.

Friday - Start building. Engineer codes the first iteration. PM and designer run tight feedback loops as it takes shape.

Four hours of synchronous time per week. Everything else is individual work feeding those conversations.

Already a huge improvement over the feature factory, where you’d burn 10-15 hours a week on sprint planning, backlog grooming, design reviews, standups, and stakeholder steering, and still not know what success looked like.

What This Model Got Right

Worth naming, because the temptation with AI is to throw everything out. Don’t.

Customer obsession as a daily practice. Talking to customers weekly, not quarterly. Building hypotheses from real behavior, not stakeholder opinions. This doesn’t change with AI. It gets more important.

Small, empowered teams. Give a trio ownership of an outcome and the freedom to figure out how to move it. The best product work I’ve done was always with a tight group that had real authority.

Outcome measurement. “Did the customer behavior change?” instead of “Did we ship the thing?” Most important practice in product management, and it predates AI entirely.

Part 2: What AI Is Breaking

Here’s what’s changed in my own practice over the last 18 months. Some of it’s uncomfortable.

The Spec Is Dead

I haven’t written a PRD in over a year. Not because I got lazy. Because I can build a working prototype faster than I can write a spec describing what it should do.

The spec used to be the primary artifact of PM work. Days writing it, days getting it reviewed and approved. That’s how you communicated intent.

Now I describe what I want to an AI coding agent and have something clickable in two hours. [The prototype is the spec](#). Customers react to a real thing, not a document. The feedback is on a different level.

This doesn't mean you stop thinking hard about problems. The thinking still matters. The 15-page doc doesn't.

The Trio Is Becoming a Quartet (or a Duo)

The old trio assumed three humans with non-overlapping skills. That's shifting.

With AI tools, a designer can build a functional prototype without waiting for engineering. A PM can run data analysis that used to need a data scientist. An engineer can generate UI variations that used to require a designer.

Roles aren't disappearing. But the *boundaries* are blurring. The trio still meets, but each person shows up with more done, more explored, more validated, because AI accelerated their solo work between syncs.

Sometimes I've seen the trio compress into a PM + engineer duo where AI handles design exploration (especially for internal tools). Other times it grows into a quartet because the product is the model and you need an ML engineer on the core team.

Discovery Gets Compressed

Before AI, a discovery cycle: Week 1, customer interviews. Week 2, synthesize. Week 3, prototype and test. Week 4, decide.

Now: Monday, 18 agents monitored customer behavior, support tickets, competitive moves, and usage data over the weekend. Tuesday morning, review synthesized insights over coffee. Tuesday afternoon, generate three prototype variations. Wednesday, test with customers. Thursday, decide.

A month becomes a week. A week becomes a day. The cycle time on learning collapsed.

This is the biggest change. PM is a learning speed game. Whoever figures out what customers need and validates it fastest wins. AI compressed that loop hard, if you set up your operating model to use it.

Execution Overhead Shrinks

Before AI, I spent maybe 40% of my time on execution overhead. Writing tickets, updating dashboards, creating status reports, grooming backlogs, writing release notes.

I've automated most of that now. Agents write first-draft release notes. They watch dashboards and ping me when something looks off. They draft weekly status updates from commit logs and Jira.

The time I got back goes into discovery and strategic thinking. AI doesn't replace the PM. It kills the parts of the job that were never the real job anyway.

Working through this at your company? I do a small number of product org audits each quarter where I write the honest assessment and a 90-day plan against the new operating model. [See current openings →](#)

Part 3: The Operating Model I'm Building Now

Still evolving. I don't have all the answers. But here's how my week looks now and where I think this is going.

The New Weekly Cadence

Monday AM - Used to be the problem discovery sync with the trio. Now I start by reviewing AI-synthesized insights: support trends, usage anomalies, competitive moves. Agents did the prep over the weekend. I show up with context instead of spending the first hour building it.

Monday PM - Used to be the start of customer outreach. Now the trio syncs, but everyone arrives with AI-assisted pre-work done. Richer starting point, faster alignment.

Tuesday - Customer interviews haven't changed. Still talking to real people. What changed: real-time AI transcription and pattern extraction. Same interviews, way faster synthesis. No more spending Wednesday morning re-reading notes.

Wednesday - Used to be solo design exploration. Now it's prototype generation. PM or designer creates 2-3 working prototypes with AI tools. Working artifacts, not wireframes. Customers react to real things.

Thursday - Used to be a direction decision on incomplete info. Now it's customer validation of actual prototypes plus a decision backed by real data. Decide on evidence, not gut.

Friday - This is the big one. Used to be sprint planning and backlog grooming. Now it's ship the first iteration, because the prototypes from Wednesday are closer to production-ready than anything we used to have at this point in the week.

The biggest shift: **Friday went from “plan what to build” to “ship what you built.”** That's not a tweak. That's a different operating rhythm.

Five Practices I'm Adopting

1. Prototype before you plan.

Skip the brief-then-spec-then-design-then-build chain. Go straight to a prototype. Use AI to get something tangible in hours. Put it in front of customers. Let their reaction guide the planning.

You still need to understand the problem. But the artifact you use to communicate and validate that understanding is now a working prototype, not a document.

2. Run an always-on sensor network.

I have 34 AI agents on daily and weekly cadences covering all seven stages of the AI Product Operating Model (Sense → Discover → Decide → Build → Ship → Measure → Amplify). They monitor customer behavior, synthesize support tickets, track competitors, and flag metric anomalies. Monday morning I have a view of what's happening without pulling a single report myself.

This flips the PM role from “go looking for signals” to “decide what to do about signals that come to you.” Big difference. You move from hunting to decision-making. [See the full agent fleet and download the setup script.](#)

3. Compress your discovery cycles.

If your discovery still takes 4 weeks, you're leaving speed on the table. AI synthesizes interviews in minutes, generates prototypes in hours, runs quant analysis in seconds. Use that speed to run more experiments, not to take longer breaks between them.

My target: no discovery cycle longer than one week. Problem on Monday, validated by Friday. Not every cycle hits that, but that's the bar.

4. Put your reclaimed time into judgment work.

The time AI gives you back shouldn't go to more meetings or more Slack. Put it into the stuff only you can do: customer relationships, hard trade-off calls, coaching your team, thinking about where the product should go in 12 months.

I track where my time goes. Before AI I was maybe 30% on judgment work and 70% on overhead. Now I'm closer to 60/40. Goal is 80/20.

5. Get good at evaluating AI output.

New skill, and it matters more than most PMs think. When your agent hands you a competitive analysis or a set of customer themes, your job isn't to redo the work. It's to spot what's right, what's missing, and what it means.

This is closer to how an exec operates: reviewing and deciding, not producing. Except you're doing it as an IC, with AI as your analyst team. PMs who can evaluate and edit AI output fast will outrun PMs who insist on doing everything from scratch.

What's Coming Next

I don't know exactly, but here's where I think this goes based on what I'm seeing:

Agents will run parts of discovery directly. Not just summarizing what customers said. Sending surveys, analyzing responses, finding patterns, recommending experiments. The PM becomes the research director, not the researcher.

The trio restructures around what AI can do. Teams stop organizing by discipline (PM, design, eng) and start organizing by outcome, each person covering more ground. One PM might own what used to take three, because agents handle the operational load.

"Speed to insight" replaces "velocity" as the metric that matters. Feature factories measured output velocity. Empowered teams measured outcome impact. AI-native teams measure how fast they go from signal to validated insight to shipped solution.

And PMs who can't work with AI fall behind. Not because AI replaces them. Because a PM with AI tools does in a day what one without them does in a week, and that gap compounds fast.

Start This Week

Pick one.

Build your first AI prototype. Take a feature on your [roadmap](#). Skip the spec. Describe what you want to an AI coding tool and get a working prototype. Show it to a customer.

Set up one monitoring agent. Pick the metric you check most often. Set up an agent that checks it daily and sends you a summary. A Slack message every morning with your key numbers and anything weird. ([Here's how I set mine up.](#))

Audit your time for one week. Track every hour. Judgment work (strategy, decisions, customer conversations) vs. overhead (status updates, ticket writing, report building). Then ask: which overhead could an agent handle?

Compress one discovery cycle. Take your current project. Use AI to synthesize existing research, generate prototype options, and get to a decision in one week instead of four.

The operating model isn't fixed. It changes as the tools change. Part 1 was right for its time and its foundations are solid. Part 3 is what matters for the next five years.

Keep the principles. Rewire the execution.

Kill the Roadmap

The roadmap is the most expensive lie in product management. Retire it and run a live bet portfolio instead.

Web version: <https://falkster.com/handbook/kill-the-roadmap> · last updated 2026-06-12

The short version

The roadmap is the most expensive lie in product management. It freezes a plan based on last month's signals, rewards conviction theater, and creates a re-plan tax so high that teams execute against plans they know are wrong. I stopped publishing roadmaps 14 months ago and replaced them with a one-page live bet portfolio, updated every Monday. Three sections: active bets (5-7 things you are testing with hypothesis, signal, kill condition, and decision date), recently killed (what you stopped and what you learned), and standing queue (fewer than 20 things waiting for a trigger). The bet portfolio is not a planning failure. It is honest about uncertainty in a market that updates daily. The hardest part is emotional: trading the false comfort of claimed certainty for the real authority that comes from knowing what is actually happening right now.

The lie everyone is performing together

I've made roadmaps at Microsoft, Adobe, Salesforce, and four startups. Every one of them was fiction within six weeks of the date I put on the title slide. The customer need shifted. A competitor shipped something. The model changed. A top-10 account asked for something I hadn't seen coming. The plan was already wrong.

But I'd still show it at the board meeting. Sales would still forecast against it. Engineering would still plan capacity against it. We all knew it was stale. Nobody said so. Instead we'd have a "re-alignment" meeting two weeks later, which is corporate code for "we all knew it was wrong but needed permission to admit it."

That's not a planning problem. That's a group performance. And it's the single most expensive piece of theater in our job.

Why the roadmap keeps breaking

Three things about roadmaps are load-bearing and broken at the same time.

The first is conviction theater. The PM who writes the most confident roadmap wins the most trust. That PM isn't more correct. Just more practiced at the performance. We reward the wrong skill.

The second is fossilized signal. The roadmap you're building today rests on customer conversations from last month, competitive data from the last quarter, adoption curves from before the last model release. By the time it's published it's already a fossil. And then you hold yourself accountable to the fossil.

The third is the re-plan tax. Everyone knows the roadmap is wrong. But changing it requires a whole process: exec review, stakeholder comms, a Slack thread about "the changes." So you don't. You execute against a plan you know is wrong because the cost of changing the plan is higher than the cost of being wrong.

Before AI, this was survivable. A quarterly plan was close enough. Now? Your competitor can ship a prototype in an afternoon. Your customer's context shifts weekly. A locked roadmap is a commitment to being slow. The deeper change underneath this is the role rewrite itself, covered in [Old PM vs Product Builder](#), [The Ledger](#): when the unit of work shifts from a spec to a working prototype, the planning artifact has to shift too.

What I run instead

I stopped publishing roadmaps about 14 months ago. I run a one-page live bet portfolio that updates every Monday. Three sections. That's it.

Active bets (5 to 7 maximum). Each bet has: the hypothesis, the surface we're testing it on, the signal we're watching, the kill condition, the next decision date, and cost-so-far. Not a feature list. A set of things we believe and are testing right now.

Recently killed (rolling 30 days). What we stopped, why, what we learned. This is the most important section. It's also the section most orgs refuse to make public because admitting failure feels expensive. Make it public anyway. It earns more trust than ten successful launches.

Standing queue. Things we'll pick up if the signal shifts. Not things we're working on. Things waiting for a trigger. The queue has fewer than 20 items. If it has more, you're rebuilding a backlog.

The document updates every Monday morning. It's visible to the whole company. Same version goes to the board, to sales, to CS, to engineering. If I ever feel the urge to "clean it up" for a board meeting, that's the signal my internal version isn't honest enough yet.

Preparing your next board deck? I review board decks for CPOs in one week, against the 2026 CPO framework, with specific rewrites and a walkthrough call before you present. [See the Board Deck Review](#) →

What a bet looks like

Here's a real one from a quarter I ran at Smartcat last year (details sanitized):

- **Hypothesis:** Mid-market buyers will choose us over the incumbent if we can show 70%+ cost reduction with comparable quality.
- **Surface:** The new pricing landing page plus the inline quality compare widget on the product dashboard.
- **Signal we're watching:** Win rate in competitive deals tagged "migration from Incumbent X," plus the 30-day retention of accounts acquired that way.
- **Kill condition:** Win rate stays below 25% for 6 weeks with no identified lever to pull.
- **Next decision:** Week 8, against Week 1 baseline.
- **Cost so far:** ~1 sprint of eng, ~2 weeks of PM/design, ~\$3k in Claude API to run the comparisons.

That's the whole bet. One card. I could publish 100 of these in the time it takes to make one slide deck that would be dead in two months.

The stakeholder conversation you're about to have

A senior stakeholder will push back. It usually sounds like: "We need predictability for planning purposes."

That's not a bad-faith ask. They're trying to do their job. The answer isn't "planning is dead." The answer is:

"We'll commit to the bets we're making this quarter. We won't commit to the specific features we'll have shipped by the end of it. Those are different commitments. The first one is honest. The second one is a lie we were all performing together."

If that lands, you have a convert. If it doesn't, run the bet portfolio alongside the roadmap for one quarter. At the end, compare: which document predicted what actually shipped? The portfolio wins every time. Now run it without the roadmap.

The hardest part isn't the process

It's emotional. A roadmap that claims certainty feels safer than a portfolio that admits bets. You want the false comfort. Resist it.

The roadmap does not protect you. It delays the moment you have to admit the plan was wrong, and the delay costs you trust when the truth arrives. The portfolio puts the uncertainty in the open, in

writing, where everyone can see what you're betting on and why. That feels exposed the first few weeks. Then it feels normal. Then the idea of going back to quarterly roadmaps feels insane.

The shift I had to make in my own head: from *“here is what we will do”* to *“here is what we are learning.”* I'm not losing authority by making this move. I'm trading theatrical authority for the real kind, the kind that comes from being the person in the room who actually knows what's happening right now.

Pick one thing this week

You probably can't delete your company's roadmap on Monday. You probably can build a bet portfolio alongside it. For the strategy layer that lives behind the bets, see [Strategy From Signals](#). Here's the 30-minute version.

1. Open a single page in Notion, Linear, or whatever you use. Title it “Active Bets.”
2. Write down the 5 things your team is actually working on right now, as bets. Each one: hypothesis, surface, signal, kill condition, next decision date.
3. If you can't write the kill condition, the bet isn't a bet. It's a feature you're committed to regardless of signal. Note that as a finding. The [assumption testing playbook](#) shows how to define testable hypotheses.
4. Share the page with your team. Ask if the bets are accurate. If anyone says “wait, I thought we were doing X,” that's the gap between what the roadmap claims and what's actually happening.

That's the first honest artifact. Keep doing it weekly. Within a quarter the roadmap will feel like a PR document and the bet portfolio will feel like the real plan. At that point you can have the conversation about retiring the roadmap. The evidence will already be on your side.

Old PM vs Product Builder, The Ledger

The line-by-line ledger of what changes when product management gets rewritten by AI: output, cycle time, unit of work, deliverable, accountability.

Web version: <https://falkster.com/handbook/old-pm-vs-product-builder> · last updated 2026-06-12

The short version

Product management got rewritten because the cost of being wrong collapsed. The old PM wrote specs to insure against expensive engineering bets, ran rituals to translate between functions, and shipped on a quarterly cadence. The Product Builder ships prototypes in an afternoon, writes evals as the contract, and runs on a daily eval-driven cadence. The unit of work moved from document to working artifact. The cycle time moved from weeks to days. The accountability moved from scope-and-timeline to outcome-and-cost-per-request. This ledger lays it out, line by line, so a CEO, CFO, or CPO can read it once and price the gap between their current org and where the market is going. If your product org is still optimized for the left column, you are paying for both jobs and getting neither.

The full role definition is at [the PM Standard](#). The career ladder is at [Builder PM](#). The org-design implication is at [The Triad Is Dead, Pods Are Dead](#). This page is the comparison.

The ledger

Twelve dimensions. Each row is one thing that changed. Read left-to-right and decide which column your org is paying for right now.

1. Core output

Old PM	Product Builder
A document. PRD, spec, one-pager, roadmap slide.	A working artifact. Clickable prototype, agent prompt, eval rubric.

The document was the deliverable. Engineering read it, asked clarifying questions, and built something slightly different. The cycle repeated. Now the prototype is the deliverable. There is nothing to interpret. The prototype either works or it does not.

2. Unit of work

Old PM	Product Builder
The feature. Scoped, estimated, sequenced.	The hypothesis. Surfaced, prototyped, evaluated.

A feature is a thing you ship. A hypothesis is a thing you test. Features lock in scope before learning. Hypotheses leave scope open until the evidence comes back. The Product Builder does not commit to building before the prototype tells them what to commit to.

3. Cycle time

Old PM	Product Builder
Weeks to months. Quarterly planning.	Hours to days. Weekly eval review.

The roadmap was built for weeks-long cycles because that was the speed of building. Cycles collapsed. The roadmap did not. That mismatch is the single biggest source of waste in old-shape product orgs.

4. Deliverable to engineering

Old PM	Product Builder
A spec to interpret.	A prototype to harden, plus an eval to gate it.

The handoff used to be “here is what to build, please build it.” The handoff is now “here is the thing that works, please make it survive a million users.” Engineering gets to build the right thing the first time because the right thing has already been validated.

5. Definition of done

Old PM	Product Builder
Acceptance criteria from the spec.	The eval passes against a real test set with named slices.

The PRD’s “done when X, Y, Z” was a list of subjective check boxes. The eval is a measured score against a curated set of inputs. You either crossed the threshold or you did not. There is nothing left to argue about in the review meeting.

6. Discovery cadence

Old PM	Product Builder
Quarterly research projects. NPS surveys.	Continuous, weekly, sometimes daily. Synthesis is the bottleneck, not interviews.

Continuous discovery was a Teresa Torres idea before AI made it cheap. Now it is the default. The constraint moved from interviewing to synthesizing. Whichever PM has the best synthesis pipeline learns the fastest.

7. Tools

Old PM	Product Builder
Jira, Confluence, Figma, Notion.	Claude Code, Cursor, v0, Replit, an eval harness, the four agents that monitor the dashboard.

Old tools optimized for coordination. New tools optimize for shipping. If your stack is mostly coordination, you built infrastructure for the old job. If it's mostly building, you built it for the new one.

8. Stakeholder rhythm

Old PM	Product Builder
Weekly status updates. Quarterly business reviews.	Live dashboards. Async digests written by agents.

The PM who spent six hours every Friday writing the status update was performing a function the dashboard can perform automatically. The time freed up goes back into shipping. The stakeholder gets a better update because it is always fresh.

9. Accountability

Old PM	Product Builder
Scope, timeline, headcount.	Outcome, cost per request, eval score, gross margin per feature.

The old accountability was about whether you delivered the plan. The new accountability is about whether the thing you delivered moves a number. Cost per request is the line item that did not exist three years ago and now sits next to revenue on the P&L the CPO presents to the board.

10. Team shape

Old PM	Product Builder
PM, designer, engineer triad inside a pod.	A Product Builder, a developer, and an agent department.

The triad assumed three humans were the unit of execution. With an agent fleet shipping work, the unit changed. The full case is in [The Triad Is Dead](#). The org chart has not caught up at most companies.

11. Hiring rubric

Old PM	Product Builder
MBA, frameworks, communication, stakeholder management.	Can ship a prototype in an afternoon, has an eval rubric in their portfolio, can defend cost per request to finance.

Resumes are tells. A Product Builder has a GitHub link, a list of shipped surfaces with metrics, eval rubrics, and an opinion on LLM provider tradeoffs. A traditional PM resume has frameworks and project-management certifications. Neither is wrong, they are answers to different questions.

12. Failure mode

Old PM	Product Builder
Spec the wrong thing, ship it exactly, learn after launch.	Ship a prototype nobody adopts, learn in three days, throw it out.

The old failure was expensive and slow. The new one is cheap and fast. The companies that compound through this transition treat the cheap failure as the signal it is, and stop paying for the expensive one.

What this means for each function

CEOs and founders, this is the most expensive transition since the move from on-premise to SaaS. The cost of standing still is not paid in a single quarter. It is paid in compounded slowness across eight quarters. The companies that made the SaaS transition in 2010 won the decade. The companies that make the Product Builder transition in 2026 win the next one.

CFOs, the new line item is cost per request. Cost per request is to AI-native product orgs what cost of goods sold is to a manufacturing business. If your CPO cannot show you cost per request per feature and the margin trajectory of each, the operating model has not landed yet. The [Eval-First Product Org](#) chapter walks through the Economics Unit pattern that owns this number.

CPOs, you cannot run this transition alongside the old org. The two operating models compete for the same people, the same calendars, and the same exec attention. Pick three teams, run the new model in full, kill the old artifacts entirely, and compare against the rest of the org at the end of the quarter. The evidence is what ends the debate, not the memo.

CTOs, the bottleneck moved from coordination to architecture. The Product Builders are going to ship prototypes faster than your platform team can absorb them. Decide now what the shared context layer looks like, where the eval pipelines run, what the agent department's mandate is, and how the prototype-to-production handoff works. The architectural decisions you make this quarter compound for two years.

Heads of GTM, the launch cycle compressed. The Product Builder writes the launch note before the prototype, sometimes a tweet before the eval. Marketing is no longer downstream of engineering. It is upstream of the prototype. The teams that get this right have the GTM lead in the prototype review, not the launch review.

What stays the same

This ledger is mostly about what changed. Three things did not.

Customer judgment did not change. Knowing which customer pain to solve next is still the hardest call in the building. AI does not help with this much. The Product Builders who win are the ones who spent the freed-up time on customer proximity, not the ones who spent it on tool fluency.

Opinionated prioritization did not change. You still have to make a call when the evidence is incomplete and the team is split. The frameworks are the same. The cadence is faster.

Taste did not change. Two prototypes that test the same on the eval rubric can be wildly different products. The one a person wants to use is the one with taste. Taste is the thing that does not fall out of any rubric. Hire for it. Promote on it. Refuse to compromise on it.

Pick one thing to try this week

If you are a CEO or founder reading this and your product org is still optimized for the left column, the smallest first step is the three-team experiment. Pick three teams. Give each PM a Claude Code license, kill the PRD requirement for the quarter, replace the weekly status update with a weekly prototype review. Track cycle time, customer-validated decisions, and ship rate. Compare against the rest of the org at the end of the quarter. The numbers will end the conversation faster than any memo.

If you are a PM or product leader reading this, the smallest first step is to ship one prototype this week without writing a spec first. One afternoon. One hypothesis. One clickable thing. Show it to a

customer. Decide what to build next from the conversation, not from the document you would have written. Do that four times in a month and the left column will feel like a foreign country.

The cost of standing still is high and rising. The cost of moving is one afternoon.

Related Foundation-wave chapters: [Why This Exists](#) (the manifesto behind the rewrite), [The AI Product Operating Model](#) (how the team rewires around the new role), [Kill the Roadmap](#) (what the planning artifact looks like after the role rewrite).

Kill the Status Meeting

The status meeting exists because nobody trusts the dashboard. Fix the dashboard once. Stop paying the tax weekly.

Web version: <https://falkster.com/handbook/kill-the-status-meeting> · last updated 2026-06-12

The short version

The status meeting exists because nobody trusts the dashboard. Fix the dashboard once and the meeting disappears. Build one URL per product: six live strips covering product health, adoption, customer signal, active bets, cost, and incidents, all updated automatically. The meeting that was 45 minutes of prep plus 45 minutes of verbal repetition becomes a 20-minute optional walk of the page, or disappears entirely. At Smartcat, killing the status meeting and replacing it with a live product page reclaimed about 6 hours a week of calendar time per PM. The only meetings that survive are the ones where a human decision gets made.

A thing I used to do every Monday for years

Block 30 to 60 minutes. Pull numbers. Build a slide. Walk through it. Get a question I'd already answered in the slide because no one read it carefully. Make a "decision." Reverse the decision by Friday because new information arrived. Repeat.

If you've been a PM for more than a year, you know this ritual. I estimate I've spent about 2,000 hours of my career in it across various companies. That's a full year of working time, mostly spent verbally repeating what should already have been visible on a screen.

The status meeting isn't a planning practice. It's a trust deficit being papered over with calendar time. And the calendar time costs more than fixing the trust deficit would.

What the meeting actually does

Look at what actually happens in a typical weekly status meeting:

- Someone presents numbers.
- Someone asks a clarifying question.
- Someone presents engineering progress.
- Someone asks why X is delayed.
- Someone presents design updates.
- Someone asks if the design is final (it's not).

- Someone summarizes.
- A “decision” is either avoided or reached with half the context.

The meeting is 45 minutes. Prep was 90 minutes. Post-meeting notes take 20 minutes and are read by maybe two people. Total cost: four person-hours to communicate roughly 200 words of new information.

That’s a 100x write-amplification ratio. You’d never tolerate that in your product. Somehow we tolerate it in our own workflow.

The replacement: a live product page

I run a single URL for each product I own. It updates automatically. Anyone in the company can refresh it at any time. It replaces the weekly status meeting almost entirely.

Six strips:

Product health. Eval score per surface, with delta from last week. Color coded. If something is red, I click in and see the failing inputs. No prep required.

Adoption. Daily active usage of the top 5 features, with 7-day and 30-day trend lines. Includes the leading indicator metric for each one (like “percent of users hitting the new flow in week 1”), not just the lagging metric.

Customer signal. Top 5 themes from support tickets, sales calls, and churn surveys in the last 7 days, auto-clustered by an agent. Each cluster links to the underlying conversations. This is the new voice of the customer, except it’s voice from yesterday, not voice from last quarter.

Bets. Active experiments, their hypotheses, their kill conditions, their next decision date. Pulled from the bet portfolio (see Kill the Roadmap).

Cost. Cost per successful action by surface, 7-day trend. The number that’s about to ruin gross margin shows up here before it shows up in the QBR.

Incidents. Open incidents, time since open, severity. Closed incidents in the last 7 days with their post-mortems linked.

At the bottom: “last updated: auto.” The page never goes stale. That’s the whole point.

What survives, what dies

Once the page exists, the weekly status meeting doesn’t need to exist. What survives:

- **Weekly product review (20 minutes, optional attendance).** I walk the page. We identify the one or two surfaces that need a decision. We decide. We end early. I don't prep slides. The page is the prep.
- **Decision conversations (called as needed, 20-30 minutes).** When a bet hits its kill condition or a regression needs human judgment. Everyone shows up having read the page. The meeting is the decision, not the briefing.
- **Customer signal sync (weekly, 30 minutes, cross-functional).** PMs, support, sales, CS, design, eng. We go through the top customer signal clusters together. This is the only meeting where new information actually gets generated.

What dies:

- The weekly status deck.
- The monthly roll-up of weekly status decks.
- The quarterly board prep that compiles all the rollups.
- Most of the recurring 1:1s that were really status updates disguised as 1:1s.

When I did this transition at Smartcat, I reclaimed about 6 hours a week of calendar time. I filled it with shipping. Not more meetings.

The build

“But how do I build the page?” It's less than a week of work, most of which is instrumentation you should already have.

- Notion, Linear, a stitched Looker dashboard, or a small Next.js page. Format doesn't matter. Discipline does.
- Connect your eval runner output, your analytics, your signal clustering agent, your cost telemetry, your incident tool.
- Autogenerate a one-paragraph summary at the top, written by an agent that reads the strips every Monday morning. Execs get their narrative without you spending two hours writing it.

If your team can't stand up a v1 of the page in two weeks, that's your discovery exercise. The act of figuring out what goes on the page is the act of figuring out what your team actually cares about, and where your instrumentation is missing.

The pushback you'll get

“But what about people who don't read the page?” They don't get to participate in the decision. That's the new contract. I'm not reading aloud at the meeting. The social pressure this creates is the

point. It moves the cost of being uninformed from the team to the individual, which is where it belongs.

“But the exec team wants a written narrative.” Give them one. Auto-generated at the top of the page. I write the prompt once, the agent regenerates the paragraph every Monday morning, execs get their narrative without me spending two hours on it.

“But my company’s culture won’t let me cancel the meeting.” Fine, keep the meeting on the calendar. But change its content. Start with “has anyone read the page?” If the answer is yes, the meeting ends in 10 minutes. If the answer is no, the meeting ends in 15 minutes and you schedule a follow-up for after they’ve read the page. Within a month, everyone reads the page.

Pick one thing this week

Don’t try to kill the meeting Monday. Do this instead.

1. Pick one product or surface you own.
2. Build a one-page dashboard for it. 3-5 strips. Rough is fine. It doesn’t need to be pretty.
3. Share the URL in the channel where your status meeting happens.
4. On Monday, start the meeting by saying “I’ll just walk through the page.” Do it. Notice that the meeting ends 15 minutes early.
5. Next week, say “you don’t need me to walk through it. Read it, then come with questions.” Notice how much faster the meeting is.
6. The third week, suggest that the meeting become “decisions only, optional attendance.”

By the fourth week, the meeting is either dead or transformed. Either is a win.

Every status meeting on your calendar is a tax on your team’s trust in its own dashboard. Pay the tax once to fix the dashboard. Stop paying it every week forever.

The “Bets” strip references the active experiment portfolio from [Kill the Roadmap](#). The customer signal strip feeds from the [Continuous Discovery on Autopilot](#) agent pipeline. For the backlog equivalent of this move (replacing a growing artifact with a live signal-fed queue), see [The Anti-Backlog](#).

Engineering Builds the Substrate, Not Features

In an AI-native org, engineering's highest-leverage work is the substrate that lets PMs and designers ship safely: guardrails, evals, isolated deploys.

Web version: <https://falkster.com/handbook/substrate-first-engineering> · last updated 2026-07-22

The short version

In an AI-native org, engineering's highest-leverage work is not features. It is the substrate: the infrastructure and toolkit that lets PMs and designers ship working software into production safely. Four pieces make it up. Scaffolded environments, so a builder gets a safe, running sandbox in minutes instead of a week. Guardrails, so no build can touch or spend more than it should. An eval harness, so every build is scored against a bar before it graduates. Isolated deploys, so a prototype reaches a real customer without ever touching the real system. When the substrate is good, a hundred people can build and only the survivors reach production. When it is missing, every prototype is a risk and engineering becomes the queue everything waits in. The role does not shrink. It moves from writing features to owning the ground everyone else builds on.

The bottleneck moved and the org chart didn't

For twenty years the constraint in product was building the software. Engineers were scarce, so everything queued behind them. The PM wrote a spec, designed handed off mockups, and both waited in line for engineering capacity. The whole operating model was built around rationing that capacity.

That constraint broke. A PM with the [two-hour prototype method](#) can now produce working software. A designer can prototype without waiting for a sprint. The cost of building a first version collapsed. But most engineering orgs are still shaped like building is the bottleneck, so they keep engineers busy shipping features that a builder PM could have prototyped in an afternoon, and the real scarce thing goes unbuilt.

The real scarce thing is safe speed. Anyone can generate a prototype. Almost no one can let a hundred prototypes run without a hundred ways to break production, leak data, or burn margin. That is the problem worth an engineer's time, and it is a platform problem, not a feature problem.

What the substrate actually is

Four pieces. Miss one and either safety or speed breaks.

Scaffolded environments. A builder should get a running, isolated environment in minutes, with the data, auth, and connections already wired, and no way to reach anything they should not. The measure is time-to-first-prototype. If it takes a builder a week and a favor from a platform engineer to start, you do not have a substrate, you have a gate.

Guardrails. Hard limits, enforced by the platform, on what any build can touch, delete, spend, or send. A builder should not be able to run up a five-figure model bill or write to a production table by accident, because the environment will not let them. Guardrails are what let you say yes to speed without saying yes to risk. This is the same instinct as treating [the guardrail as a product decision](#), pushed down into the platform.

An eval harness. Every build is scored against a bar that defines what good looks like, the contract from [The Eval Is The Spec](#). The harness is shared infrastructure, not something each builder reinvents. It is what turns “this looks good in a demo” into “this cleared the bar,” and it is the gate a prototype must pass to graduate to hardening.

Isolated deploys. A prototype must be able to reach a real customer without touching the real system. Preview environments, shadow traffic, feature-flagged surfaces, whatever fits, but the property is non-negotiable: customer contact without production risk. This is what makes same-day customer feedback safe rather than reckless.

What engineers do instead of features

This is not a demotion, it is a promotion. Worth saying plainly, because good engineers will hear “stop building features” as “you matter less.”

Engineers own the problems that got harder, not easier, when everyone started shipping. Reliability at scale. Security and data boundaries. The architecture that lets a prototype survive contact with real users instead of falling over at the first thousand of them. The hardening step where a graduated prototype becomes a real system, described in the handoff in [Old PM vs Product Builder](#). And the substrate itself, which is a product with internal customers and deserves to be treated like one.

The shift is from being a queue that features wait in to being the team that decides what everyone else is allowed to do safely. Fewer engineers touch any single feature. Every engineer’s work touches every feature, because it is the ground all of them run on. That is more leverage, not less.

How you know the substrate is working

You do not measure it by features shipped. You measure it by what non-engineers can do without an engineer in the loop, and by what still cannot go wrong when they do.

A healthy substrate looks like this: a builder starts a safe environment in minutes, builds a prototype, puts it in front of a customer the same day through an isolated deploy, and the whole time there was no path for them to touch production, leak data, or blow the budget, and nothing graduated without clearing the eval. Engineers spent their week on reliability, security, and the substrate itself, not on translating a builder's prototype into a spec and back.

If instead your engineers are the bottleneck every prototype waits behind, the substrate is thin and you are paying senior engineers to be a queue.

Start this week

Pick the single sharpest edge and blunt it. Find the one thing a builder cannot currently do without an engineer, or the one accident the environment currently allows, and fix that first.

Usually it is time-to-first-prototype (make a safe sandbox a builder can start alone) or the missing guardrail (make it impossible to touch production or overspend by accident). Ship that one piece of substrate, then watch how many “can an engineer help me with” requests disappear. That number is your ROI, and it is how you make the case for building the rest.

The PM 30/60/90

A week-by-week 90-day plan for PMs starting a new job: decision archaeology, a signal map, a manager contract, and one compounding artifact by day 60.

Web version: <https://falkster.com/handbook/pm-30-60-90> · last updated 2026-06-04

Starting from zero is an advantage for about 90 days

Every new PM job starts the same way: you know nothing, nobody trusts you yet, and everyone is too busy to onboard you properly. The standard advice is to schedule coffee chats and absorb. That advice wastes the one advantage you will never have again: for about 90 days, you are the only person in the building who can see the product without the org's story wrapped around it. After that, you have a story too.

I have started over enough times, Microsoft to Adobe to Salesforce to four startups to Smartcat, to know the window closes fast. This chapter is the plan for spending it well. It is the sibling of [The Builder PM 30/60/90](#), which covers transforming a role you already have. This one covers landing in a role you don't understand yet.

The short version

The PM 30/60/90 for a new job runs in three moves. Before day 1 and through days 1 to 30: set up your personal stack before you start, run decision archaeology on the last 90 days of your product area, build a signal map instead of a stakeholder map, and write a manager contract in week one. Days 31 to 60: ship one compounding artifact, a call synthesis, a live dashboard, an eval set, or a debate-ending prototype, and use it to earn your seat in decision rooms. Days 61 to 90: take ownership of one bet with a falsifiable success measure, kill or renegotiate one inherited commitment, and write a day-90 note to your manager that sets the scoreboard for the next year. Templates for the contract, the signal map, and the day-90 note are in [the PM First-90 Kit](#).

Day zero: infrastructure before access

You do not need a company laptop to start. Three setups before day 1.

Stand up your second brain. A notes system with a daily capture habit, so the firehose of your first weeks gets stored somewhere queryable instead of evaporating. Mine is described in [the PM second brain](#). The first month of a new job is the highest-value input stream of your tenure and most PMs let it run straight to the floor.

Start your prompt repo and agent scaffolding. The synthesis agents you will want in week three, call synthesis, doc digestion, meeting summaries, take an evening to set up and zero company data. Build them now on public material. The full setup is in [the PM agent stack](#).

Use the product as a customer. Personal email, real task, hit the paywall, file a ticket. Write down everything that confused you. In six weeks you will be incapable of this read. It is the cheapest user research you will ever produce and it makes your first customer calls ten times sharper.

Days 1 to 30: archaeology, not coffee chats

You will be told to “meet everyone.” Fine, meet them. But structure the month around evidence, not introductions.

Run decision archaeology. Reconstruct the last 90 days of your product area: merged PRs and what shipped, launch posts, the killed projects, the escalations, the pricing or packaging changes. For each significant decision, trace who raised it, where it got debated, who actually decided, and what evidence was in the room. Two effects. First, you learn how the org really works, which never matches the wiki. Second, within three weeks you are the best-informed person in most meetings about recent history, which is a strange and useful kind of credibility for someone with no tenure.

Build a signal map, not a stakeholder map. A stakeholder map tells you who has power. A signal map tells you where truth enters the building: which support queue, which sales call recordings, which dashboard, which customer Slack channel, which engineer who quietly knows everything. Your job long-term is to sit closer to the signal than anyone else. Find the sources in month one and subscribe to all of them. Route them into a daily digest the way [Continuous Listening](#) describes.

Write the manager contract in week one. One page, co-written with your manager: what you own, what good looks like at day 90, how decisions get made between you, what they are silently worried about. Ask that last one directly. Most PM onboarding failures I have witnessed were expectation mismatches that were never written down, discovered at month four, fatal by month six. The contract converts an ambient vibe into an editable artifact. Template in [the PM First-90 Kit](#).

Sit in eight raw customer calls. Before reading anyone’s synthesis, form your own. The [interview guide](#) gives you the structure. Feed the transcripts to your synthesis agent and compare its read with yours and with the team’s official narrative. The three-way gap is your first real discovery.

What you do not do in month one: share conclusions, propose strategy, or critique the roadmap. Observations and questions only. The deep dive on this month is [the PM first 30 days, mapped](#).

Days 31 to 60: ship one compounding artifact

Month two is where you convert evidence into standing.

The standard advice says find a quick win. Quick wins are usually stunts: small enough to be ignored, cosmetic enough to be resented, forgotten by the next sprint. Ship an artifact instead. An artifact keeps working after you ship it.

Four proven options, pick the one your team's weather calls for:

The synthesis. Twenty customer calls, synthesized into themes with verbatim quotes and one recommended action each, delivered to the team. Costs you two days with the agent stack you built at day zero. The [call triage recipe](#) is the exact pipeline.

The eval set. Take the next feature on the roadmap and write its eval set before anyone asks. Thirty to 150 labeled examples that define what good looks like. You just became the person who defines quality for that feature. [The Eval Is the Spec](#) is the method.

The live page. One URL with the signals the team currently assembles by hand every week: adoption, eval scores, cost, open customer themes. The [staff meeting dashboard pattern](#) shows what this replaces.

The prototype. Find the debate that has been circling for weeks and end it with two working prototypes built in an afternoon. [Prototype Before You Spec](#) is the craft; [60 minutes is enough](#) is the proof.

Ship it, then watch the reactions. Who used it, who ignored it, who got defensive. That reaction data feeds the same coalition logic a CPO runs at larger scale in [The CPO 30/60/90](#). The month is broken down in [days 31 to 60, first artifact](#).

Days 61 to 90: own a bet

Month three converts standing into ownership.

Take one bet and put your name on it. Not the whole roadmap. One outcome with a falsifiable success measure and a date: “activation for segment X moves from A to B by date Y, and here is the evidence this is the right lever.” Write it as a one-pager. Socialize it through the signal map you built. This is the moment you stop being new.

Kill or renegotiate one inherited commitment. Every PM inherits at least one commitment that the evidence no longer supports: a promised feature, a standing report, a ceremony. Pick the safest one, lay out the evidence, and either kill it or renegotiate it openly. Doing this once, calmly, with receipts, teaches the org how you operate and makes every future no cheaper. The mechanics live in [the anti-backlog](#).

Write the day-90 note. One page to your manager: what you found, what you shipped, the bet you own, what you killed, and what you want to be measured on for the next year. This closes the manager contract from week one and opens the next loop. It is also, quietly, the first draft of your next promotion case. Structure in [the PM First-90 Kit](#), and the month deep dive in [days 61 to 90, own a bet](#).

What you'll feel

Day 5: useless, because everyone references history you don't have. Archaeology fixes this faster than tenure does.

Day 25: pressure to have opinions. Hold. An opinion at day 25 is a guess wearing your reputation.

Day 50: the artifact lands and someone senior asks who made it. This is on schedule.

Day 80: the first real pushback on your bet. Welcome. Pushback at day 80 means you proposed something with actual content.

Pick one thing this week

If you have signed but not started: set up the second brain and the synthesis agent tonight. Two hours, no company access needed.

If you are inside the first 30 days: start decision archaeology this afternoon. Pull the last quarter's launch posts and killed projects and start the timeline.

If you are past day 90 and none of this happened: run the plan anyway, compressed. The window for the fresh read is gone, but the ledger, the map, the artifact, and the bet work at any tenure. The templates are in [the PM First-90 Kit](#).

Sources: [Michael Watkins, The First 90 Days](#) for the classic frame, [Teresa Torres, Continuous Discovery Habits](#) for the interview cadence the signal map feeds, [Julie Zhuo, The Making of a Manager](#) on manager expectation-setting.

Part 2: Discover

Find the right problem before you build.

Continuous Discovery

AI ingests every call, ticket, and email, extracts the signal, and hands you a prototype the same day. Continuous discovery is no longer a weekly ritual.

Web version: <https://falkster.com/handbook/continuous-discovery-autopilot> · last updated 2026-07-22

The short version

Continuous discovery used to mean two customer interviews a week and a monthly synthesis session. AI changes the math entirely. Agents now ingest every sales call, support ticket, NPS response, and app review your company generates, extract signals automatically, and surface a ranked opportunity brief every Monday morning. When a signal emerges, you prototype in hours using AI coding tools and put something working in front of the customer the same day. The cycle that used to take four to eight weeks now runs same-day when the signal is clear, and inside a week when the problem needs a human in the room. Teresa Torres was right about the habit of continuous discovery. AI removed the excuse that you don't have bandwidth to do it.

What hasn't changed

Before I blow this up, let me be clear about what still matters.

Talking to customers matters. Understanding their world matters. Empathy matters. The discipline of asking “why” five times until you get to the real problem, that matters. [Teresa Torres](#) was right about the habit of continuous discovery. Nothing I'm about to say replaces the human act of sitting across from a customer and listening.

But everything around that conversation has changed.

The way you find signals has changed. The way you process what you hear has changed. The way you validate whether it matters has changed. The speed at which you can go from “I think this is a problem” to “here, try this” has changed. And most importantly, the ratio of signal to noise has flipped. You used to have to hunt for insights. Now you have to filter them.

If you're still doing discovery the way you did it two years ago, you're leaving speed on the table. And speed to insight is the only competitive advantage a PM has left.

The old model is too slow

Here's how discovery worked before AI. You'd schedule two customer interviews a week. Good PMs did this consistently. Great PMs synthesized the patterns monthly. You'd do this for a quarter, build an Opportunity Solution Tree, design some experiments, run them over weeks, and eventually decide what to build.

That cycle took four to eight weeks from signal to decision. Sometimes longer.

The problem isn't that it was wrong. It was right. The problem is the world moves faster now. Your competitor is shipping prototypes while you're still synthesizing interview notes. Customers are switching to whoever solves their problem first, not best.

And here's the uncomfortable part: while you were doing two interviews a week, your company was generating thousands of customer signals every single day. Support tickets. Sales call recordings. Customer emails. NPS responses. App reviews. Slack threads. Product analytics. Every one of those is a customer telling you something. And 99% of it sat in different systems, unread by anyone on the product team.

Two interviews a week was the best we could do with human bandwidth. It's not the best we can do anymore.

The new discovery loop

Here's the model I'm running now. It looks nothing like what I was doing 18 months ago.

Step 1: Ingest everything. Every customer touchpoint your company generates flows into an AI processing layer. Every sales call gets auto-transcribed and analyzed. Every support ticket gets categorized and pattern-matched. Every customer email gets scanned for pain signals. Every NPS response, app review, and Slack mention gets processed. You're not reading any of this manually. Agents do it.

Step 2: Extract the signal. AI doesn't just transcribe. It extracts. From a 45-minute sales call, it pulls: what problems the customer described, what they're currently using, what they asked about, what made them hesitate, what competitor they mentioned, what timeline they need. From 200 support tickets this week, it surfaces: the top 5 emerging pain points, which customer segments are affected, how severity compares to last week, and whether anything new is trending.

The extraction that matters most is the outcome, not the feature. When a customer asks for a specific button or setting, the agent records the result they are trying to reach, because the feature request is only their guess at a solution and the outcome is the thing you actually have to serve. Extract the job. The feature they named is disposable. The outcome is not.

Step 3: Surface opportunities. Every Monday morning, before I’ve opened Slack, I have a synthesized view of what customers are struggling with. Not from two interviews. From every customer interaction the company had last week. Thousands of data points, compressed into the 5 to 10 things that actually matter. Patterns I’d never catch manually because they span sales calls, support tickets, and usage data simultaneously.

Step 4: Prototype immediately. When you spot an opportunity, you don’t write a spec. You don’t schedule a design review. You don’t estimate engineering effort. You build a working prototype. Today. In hours, not weeks. AI coding tools make this possible. Describe the solution, get something clickable, and put it in front of a customer by Wednesday.

Step 5: Give it to the customer. Not a mockup. Not a wireframe. Not a concept drawing. A working prototype they can actually use. “Here, try this. Does this solve your problem?” Their reaction to a real thing is worth more than ten interviews about a hypothetical thing.

Step 6: Get feedback, iterate, scale. The customer’s reaction tells you everything. They use it and their eyes light up? Scale it. They use it and shrug? Iterate. They don’t use it at all? Kill it. You’ve spent hours, not months. The cost of being wrong dropped to near zero.

That’s the loop. Signal, prototype, customer, feedback, scale. It runs in days, not months.

Auto-transcribing everything your company generates

Step 1 is where most PMs stop before they start. “I can’t process all that data” is the excuse. You can.

Sales calls. Tools like Gong, Chorus, and Fireflies already transcribe every call. But transcription isn’t the point. The point is extraction. Set up an agent that reads every sales call transcript and pulls out: customer problems mentioned, features requested, competitors compared, objections raised, and commitments made. You don’t read the transcripts. You read the extraction. Five minutes instead of forty-five.

Support tickets. Connect your Zendesk, Intercom, or Freshdesk to an AI agent that runs weekly. It reads every ticket, categorizes by problem type and customer segment, identifies trending issues, and flags anything that spiked versus the prior week. When “export is broken” goes from 3 mentions to 30 in a week, you know about it Monday morning, not when a VP escalates it Friday afternoon.

Customer emails. Your CS team gets hundreds of emails a week. Most contain useful signal buried in polite language. An agent scans them for pain indicators: phrases like “we’re considering alternatives,” “this has been a problem for months,” “our team is frustrated with,” “when will you support.” These get flagged and categorized. You see the patterns across all customer communication, not just the loudest complainers.

NPS and surveys. Open-ended NPS responses are a goldmine that nobody reads. An agent processes every response, extracts the core sentiment and specific product areas mentioned, and groups them by score band. Your detractors are telling you exactly why they're unhappy. Your passives are telling you what would tip them to promoters. Read the synthesis, not the raw responses.

App reviews and social mentions. G2, App Store, Trustpilot, Reddit, Twitter. An agent monitors all of them, extracts product-relevant mentions, and surfaces anything new. When three people on Reddit mention switching from you to a competitor for the same reason in the same week, you want to know.

Product analytics. This one's different because it's behavioral, not verbal. An agent monitors your key flows and flags anomalies. Onboarding completion dropped 12% this week. The export feature usage spiked 40%. Users who hit the billing page are bouncing 3x more than last month. These are signals that no interview would surface because customers don't always know their own behavior.

All of this runs automatically. You set it up once. It runs every day or every week depending on the source. Monday morning, you have a unified view of what your customers are experiencing. Every single one of them, not just the two you talked to.

From signals to opportunities (automatically)

Raw signals are useless without synthesis. This is where the second layer of agents comes in.

A synthesis agent takes the extracted signals from all sources and does what you used to do manually on Friday afternoons: looks for patterns.

It asks: What problems are showing up across multiple channels? If “export is slow” appears in 40 support tickets, 3 sales call objections, and 2 NPS detractor responses, that's not three separate issues. That's one opportunity with strong signal across channels.

It also asks: What's new this week versus last week? Trending problems matter more than chronic ones because trending means something changed. Maybe you shipped a release that broke something. Maybe a competitor launched a feature that's making your users jealous. Maybe a customer segment is growing and hitting a scale wall you didn't anticipate.

And it asks: Which customer segments are affected? “Export is slow” is generic. “Export is slow for enterprise customers processing more than 10,000 records” is actionable. The agent breaks down patterns by segment so you know who cares and how much revenue is at stake.

The output is a weekly opportunity brief. Five to ten opportunities, ranked by signal strength and business impact. Each one includes: what the problem is, who's affected, how we know (which channels surfaced it), how severe it is, and whether it's getting better or worse.

You used to build this picture over weeks of interviews and analysis. Now it arrives in your inbox Monday morning.

Rapid prototyping as the core of discovery

The prototype is no longer the output of discovery. It's the tool of discovery.

In the old model, you'd discover a problem, validate it through more interviews, design a solution, spec it, build it, test it, ship it. Discovery and delivery were sequential phases with a handoff in between.

In the new model, discovery and delivery merge. You discover a signal on Monday. You prototype a solution on Tuesday. You put it in front of a customer on Wednesday. Their reaction is the validation. The prototype did the job of weeks of interviews and specs and designs, in hours.

This works because AI coding tools have collapsed the cost of building a working prototype from weeks to hours. When I say prototype, I don't mean a Figma mockup. I mean a functional thing. Something the customer can click through, enter data into, and react to as if it were real. Something that provokes an honest reaction because it feels real.

Describe the problem and your hypothesis to an AI coding agent. "Build me a dashboard that shows freelancers their top 10 job matches, ranked by fit score, with a one-click apply button." Two hours later, you have something to show.

That prototype is worth more than a spec document, more than a mockup, and more than an interview where you describe the idea. Because people react to real things differently than they react to hypotheticals. They find edge cases. They reveal their actual priorities. They show you, through how they use it, what matters and what doesn't.

I've had customers use a prototype for five minutes and reveal an insight that would have taken me a month of interviews to surface. "Oh, I wouldn't use the fit score. I'd just sort by deadline because I need work this week." That one sentence kills the recommendation algorithm and replaces it with a much simpler deadline-sorted view. Five minutes, one prototype, one insight that saved months.

Agentic discovery: Ideas that will feel radical

Now push further. Here are things I'm building toward or actively experimenting with. Some will sound extreme.

The auto-interviewer. An AI agent that conducts asynchronous interviews with customers via chat. Not a survey. An actual conversational interview that follows up, asks "why," and probes deeper based on responses. It runs 24/7. It can interview 100 customers in a week instead of 2. The PM

reviews the synthesized patterns, not the raw transcripts. You still do live interviews for the hardest problems, but 80% of your discovery volume is handled by the agent.

The opportunity radar. An agent that doesn't wait for you to ask "what should we build?" It monitors all signals continuously and proactively surfaces emerging opportunities. "Three enterprise customers mentioned HIPAA compliance this week. None of them mentioned it last month. Two of them have renewals in Q3. This is a new signal worth investigating." You didn't ask for this analysis. The agent brought it to you because it noticed the pattern.

The prototype factory. When the opportunity radar surfaces a signal, a second agent automatically generates a prototype that addresses it. By the time you read the opportunity brief on Monday morning, there's already a clickable prototype attached. You review the opportunity, open the prototype, and decide: is this worth showing to a customer today? If yes, you're in front of a customer with a working solution before lunch. If no, you've lost nothing.

The feedback loop closer. After a customer uses a prototype, an agent follows up automatically. "You tried the new export feature yesterday. Quick question: did it solve the problem you were having?" The response gets fed back into the signal processing layer. The loop closes itself. Signal, prototype, customer, feedback, back to signal.

The competitive shadow. An agent that monitors competitor product changes, pricing updates, G2 reviews, job postings, and social mentions. When a competitor launches a feature your customers have been requesting, you know within hours. When a competitor posts job listings for ML engineers, you know they're investing in AI before they announce it. The synthesis includes: "Competitor X launched feature Y. 14 of our customers have requested something similar in the past 90 days. Here are the three most relevant customer requests."

The churn predictor. An agent that combines usage data, support ticket sentiment, NPS scores, and engagement trends to predict which customers are at risk of churning. Not after they've left. Before. "Customer ABC's usage dropped 30% this month. They submitted 4 support tickets about the same issue. Their NPS score went from 8 to 5. Their contract renews in 60 days. Recommended action: reach out this week with a prototype that addresses their specific issue." You're not reacting to churn. You're preventing it with a targeted prototype delivered before they even consider leaving.

What your week looks like now

Here's what my actual week looks like.

Monday morning. I open my laptop and review three things: the weekly opportunity brief (auto-generated from all signals), the agent health dashboard (are all my data pipelines working), and the

prototype queue (any auto-generated prototypes worth reviewing). This takes 30 minutes. I now have a clearer picture of my customers than I used to get from a month of manual work.

Monday afternoon. I pick the top two opportunities from the brief and decide how to validate them. For one, I schedule a customer call. For the other, I build a prototype. The prototype takes two hours. I share it with two customers via async message: “Hey, we’re exploring a solution for the issue you mentioned. Mind taking a look?”

Tuesday. I do my live customer call. But unlike the old model, I’m not exploring blindly. I already know, from the signal analysis, exactly what this customer is struggling with. The interview is targeted. I show them the prototype. Their reaction tells me more in 10 minutes than an hour of open-ended questions would.

Wednesday. Customer feedback is in from the async prototype shares. One loved it. One said “close but I’d need X instead of Y.” I iterate on the prototype in an hour. I share the updated version. Meanwhile, the feedback loop agent is processing new signals from this week’s support tickets and sales calls.

Thursday. Decision day. Based on the prototype feedback and signal data, I decide: we’re building this. Not the original version. The iterated version that customers shaped. I brief engineering with the prototype as the spec. They can see exactly what we’re building because it already works.

Friday. Engineering starts building the production version. I review next week’s opportunity brief draft. I check the assumption tracker to see which hypotheses are still unvalidated. I plan what to prototype next week.

Five days. From signal to validated solution to engineering kickoff. That used to take eight weeks.

The human parts that matter more, not less

I need to say this clearly because it’s easy to misread what I’m arguing: AI makes the human parts of discovery more important, not less.

The reason is simple. AI handles volume. It processes thousands of signals. It generates prototypes. It tracks patterns. But it can’t do the things that actually make a great PM.

It can’t sit across from a customer and feel the frustration in their voice when they describe a workaround. It can’t read body language. It can’t build the relationship that makes a customer honest with you instead of polite. It can’t make the judgment call about which opportunity matters most when the data is ambiguous.

It can't ask the follow-up question that nobody thought to ask. "You said you use Slack to find work instead of our search. What is it about Slack that works better?" That question, and the answer, led to one of our best features. No agent would have asked it because it requires human curiosity and context.

The new model doesn't eliminate customer conversations. It makes them 10x more powerful. You show up to every conversation with full context. You know what this customer's been struggling with because the agents already analyzed their support tickets, usage patterns, and NPS scores. You bring a prototype to react to instead of hypotheticals to discuss.

Your interviews go from "Tell me about your experience" to "I noticed you've been exporting large datasets three times a week and hitting timeout errors. We built something that might fix that. Want to try it?" That's a different caliber of conversation.

Setting up your discovery engine

Here's the practical path. You don't need to build all of this at once.

Week 1: Turn on transcription. If your sales and CS calls aren't being transcribed, start there. Gong, Fireflies, or even Otter.ai. Get every customer conversation into text. This is the raw material for everything else.

Week 2: Build your first signal agent. Pick your highest-volume channel, probably support tickets. Set up an agent that reads last week's tickets, extracts the top 10 problems, and sends you a summary every Monday. This alone will change how informed you are. See the [setup guide](#) for the technical details.

Week 3: Add a second channel. Expand to sales call analysis or NPS responses. Now you have two signal streams feeding your Monday brief. Patterns that span both channels are gold.

Week 4: Build your first prototype in response to a signal. When Monday's brief surfaces an opportunity, don't schedule more research. Build a prototype. Show it to a customer. Get feedback. Close the loop in a week.

Month 2: Add the synthesis layer. Connect your signal agents to a synthesis agent that cross-references patterns across channels. Add the opportunity radar. Start getting proactive alerts instead of weekly summaries.

Month 3: Scale the loop. Add the prototype factory. Add the feedback loop closer. By now, you're running 3-5 discovery cycles per week instead of 1-2 per month. Your learning velocity has increased by an order of magnitude.

For the full agent fleet and scheduling details, see [Your AI Agent Fleet](#).

Start this week

Pick one.

Turn on transcription. If you're not transcribing every customer call, that's step one. You're sitting on thousands of insights locked in audio files nobody will ever replay.

Build one signal agent. Support tickets or NPS responses. Whichever you have the most of. Get a weekly summary of what customers are telling you. Follow the [setup guide](#).

Prototype something today. Take the most common customer complaint you know about. Don't research it further. Build a working prototype that addresses it. Show it to one customer. See what happens.

Run the full loop once. Signal on Monday. Prototype on Tuesday. Customer on Wednesday. Feedback on Thursday. Decision on Friday. Do it once and you'll never go back to the old way.

The discovery fundamentals haven't changed. Understanding customers, finding real problems, testing before building. What changed is the speed at which you can do all of it. And in a world where the PM who learns fastest wins, speed is everything.

Build Your First Opportunity Solution Tree

Build your first Opportunity Solution Tree. AI populates the opportunity layer from 500+ signals. Prototype and test a solution in one week instead of eight.

Web version: <https://falkster.com/handbook/your-first-ost> · last updated 2026-06-12

The short version

An Opportunity Solution Tree connects a business outcome to customer problems, candidate solutions, and experiments. Teresa Torres created the framework. The structure has not changed. What has changed is speed: AI can populate the opportunity layer from hundreds of support tickets, sales calls, and NPS responses in minutes, where interviews used to take weeks. The new one-week loop is: Monday, review the AI-generated opportunity brief; Tuesday, build a working prototype for the top opportunity; Wednesday and Thursday, show it to five customers; Friday, decide and update the tree. At Smartcat I went from customer signal to shipped feature in four weeks using this loop. The old model took three to four months.

What an Opportunity Solution Tree actually is

An Opportunity Solution Tree (OST) is not a roadmap. It's not a feature list. It's a living map that connects a business outcome to the customer problems that drive it, the solutions you could build, and the experiments you're running to learn what works.

[Teresa Torres](#) created this framework and it's still one of the most useful thinking tools in product management. The structure is simple: outcome at the top, opportunities below it, solutions below those, experiments at the bottom. Each layer answers a different question. What are we trying to achieve? What customer problems could we solve to get there? What could we build? How do we know it works?

The power of the OST is that it separates three thinking modes most PMs conflate. Discovery mode: what problems do customers actually have? Solution mode: what are different ways to address each problem? Validation mode: which solution is worth the engineering time?

What's changed is how fast you can populate, test, and iterate on each layer. The structure is the same. The speed is radically different.

The old way vs. the new way

The old way to build an OST:

Week 1-2: Run 4-6 customer interviews to identify opportunities. Week 3: Synthesize interview notes into opportunity themes. Week 4: Brainstorm solutions with your trio. Week 5-6: Design and run assumption tests. Week 7-8: Analyze results, update the tree, decide what to build.

Eight weeks from “we need to learn” to “we know what to build.” That was considered fast.

The new way:

Monday: Review your AI-generated opportunity brief. It synthesized signals from 500 support tickets, 30 sales calls, 200 NPS responses, and last week’s usage data. Five clear opportunities emerged, each with signal strength and segment data.

Tuesday: Pick the top opportunity. Build a working prototype that addresses it. Two hours.

Wednesday: Show the prototype to three customers. Watch them use it. Get reactions.

Thursday: Iterate on the prototype based on feedback. Show the updated version to two more customers.

Friday: Decide. Build it, iterate more, or kill it. Update the tree.

One week. Same quality of learning. Fraction of the time.

Step 1: Start with your outcome (this hasn’t changed)

The root of your tree is still a business outcome. Make it specific and measurable.

Not “make users happy.” Not “improve the product.” Something like: “Reduce monthly churn from 8% to 5% in 6 months.” Or “Increase free-to-paid conversion from 3% to 6% this quarter.” Or “Grow enterprise ARR by \$2M this year.”

This part is pure strategy and judgment. No AI helps you pick the right outcome to pursue. That’s your job as a PM. Pick the outcome that matters most to your business right now and put it at the top.

At Smartcat, my outcome last quarter was “Increase freelancer activation by 50%.” Getting new marketplace users from signup to first paid project. Everything below it was a hypothesis about how to get there.

Step 2: Let AI populate your opportunities

This is where the new model diverges sharply.

In the old model, you’d spend weeks interviewing customers to identify opportunities. You’d schedule calls, prepare questions, take notes, synthesize patterns. It was necessary work but

incredibly slow.

Now, your opportunity layer populates from AI signal processing. If you've set up the discovery engine described in [Continuous Discovery on Autopilot](#), you already have an agent synthesizing customer signals across channels every week.

That synthesis tells you things like:

“42 support tickets this month mention difficulty finding relevant projects. 8 of those are from enterprise accounts. 3 sales calls this week included objections about job discovery. NPS detractor comments mention ‘search’ 2x more than last quarter.”

That's an opportunity: Freelancers struggle to find relevant projects because search doesn't match how they think about their skills.

You didn't need to interview 10 people to find this. The signal was already there, scattered across your support queue, sales recordings, and NPS responses. AI collected and connected it.

But you still need human judgment to evaluate whether the AI-identified opportunity is real. Signal strength tells you customers care about this. It doesn't tell you whether solving it moves your outcome metric. It doesn't tell you whether it's the highest-leverage opportunity compared to others. That's your job.

Here's how I evaluate AI-surfaced opportunities:

Signal strength. How many data points support this? Across how many channels? A problem that shows up in support tickets, sales calls, and NPS responses is stronger than one that only shows up in one place.

Segment relevance. Does this affect the customers who matter most for your outcome? If you're trying to reduce churn, opportunities from churned or at-risk customers matter more than feature requests from your happiest power users.

Business impact. If you solved this, would it move the outcome metric? Some problems are real and painful but don't connect to the metric you're optimizing.

Feasibility signal. Can you even imagine a solution? Sometimes AI surfaces a problem that's real but not something you can address. “Customers want us to be cheaper” is a valid signal but might not be an actionable opportunity.

Pick three to five opportunities. Not fifteen. Focus is what makes the tree useful.

Step 3: Prototype solutions (don't brainstorm them)

Here's the biggest shift from the traditional OST process.

The old model said: brainstorm 5-8 solutions per opportunity. Have a whiteboard session. Generate ideas. Then design experiments to test them over weeks.

The new model: build the solutions. Not as production features. As prototypes. Today.

For our job discovery opportunity, instead of listing “personalized recommendations, better search filters, curated digest, onboarding tutorial” on a whiteboard, I do this:

Prototype A (2 hours): Build a working search interface with skill-based matching. Freelancers enter their specialization in their own words, and the prototype shows matched projects. Clickable, functional, real data.

Prototype B (2 hours): Build a curated weekly digest. A page that shows “Top 10 projects for Python developers this week,” generated from actual listings, with one-click apply.

Prototype C (1 hour): Build a simple “skills profile” flow where freelancers describe what they do, and the prototype shows how many matching projects exist. “Based on your skills, there are 47 projects that match you right now.”

Three prototypes. Five hours of work. Each one is a testable solution that a customer can actually react to.

Why this is better than brainstorming: customers react to real things differently than they react to descriptions. Show someone a whiteboard sketch and they say “sure, looks good.” Show them a working prototype and they say “oh, I wouldn't use it this way, I'd want to sort by deadline because I need work this week.” That one sentence of honest reaction is worth more than a two-hour brainstorm session.

Step 4: Test with customers (this week, not next month)

You have three prototypes. Now show them to customers. Not in a month. This week.

The rapid prototype test:

Pick 5 customers in your target segment. Reach out: “Hey, we're exploring some ideas for helping you find relevant projects. Would you spend 15 minutes looking at something we built?”

Share the prototype. Watch them use it. Don't explain. Don't guide. Just watch.

What you're looking for:

Do they understand what it does without explanation? If not, the concept is too complicated.

Do they engage with it, or do they click around politely and stop? Engagement is signal. Politeness is noise.

Do they say something that reveals a deeper insight? “Oh, I’d use this but only on Mondays when I’m looking for new work.” That tells you frequency. That tells you context. That tells you where to put it in the product.

Do they compare it to something they already do? “This is kind of like what I do manually in Slack.” That validates the opportunity and tells you the current workaround.

The before/after test:

Even faster. Show customers the current experience, then the prototype. Ask: “Which one helps you more? Why?” The comparison forces specificity. They can’t just say “it’s nice.” They have to say what’s better and why.

The leave-it-with-them test:

For prototypes that are functional enough, share it and let them use it for a few days. Follow up: “Did you end up using it? What happened?” If they used it without being asked, that’s the strongest signal. If they forgot about it, that tells you something too.

You’re not looking for statistical significance. You’re looking for strong directional signal from 5-10 people. Is this solving a real problem? Would they use this? What’s missing?

Step 5: Update the tree weekly

The traditional OST was a quarterly artifact. You’d build it once, maybe update it after a big research cycle.

In the new model, the tree updates every week. Because you’re running the discovery loop every week.

Every Friday, I spend 30 minutes updating my OST:

What did I learn this week? Which prototype got the strongest reaction? What surprised me? Did any opportunity get stronger or weaker?

What do I cut? If a prototype got zero engagement from 5 customers, I remove that solution branch. If an opportunity turned out to be a niche concern (1-2 customers, not a pattern), I deprioritize it.

What’s next? Which opportunity or solution should I prototype next week? Which assumption is still untested?

The tree is alive. If it looks the same three weeks in a row, you're not learning fast enough.

A real OST that changed a quarter

Let me walk you through how this played out at Smartcat.

Outcome: Increase freelancer activation by 50%.

Monday: My weekly opportunity brief surfaced four patterns from signals across channels: 1. New freelancers can't find relevant projects (40+ support tickets, 3 sales call objections) 2. Profile completion is too complex (usage data showed 55% abandon at step 3) 3. First-project anxiety, freelancers self-filter out of jobs they're qualified for (8 interview mentions over 3 weeks) 4. No feedback after rejection, freelancers stop applying (CS email pattern)

Tuesday: I built two prototypes. One: a simplified profile flow that took 3 minutes instead of 15. Two: a "match score" that showed freelancers how well they fit each project, reducing self-filtering.

Wednesday: I showed both prototypes to 6 new freelancers. The simplified profile got polite nods. The match score got excitement. One freelancer said: "If I could see I'm an 85% match, I'd definitely apply. Right now I just guess and give up."

Thursday: I iterated on the match score prototype. Added a "what's missing" view that showed freelancers exactly what to add to their profile to improve their score. Showed it to 4 more freelancers. Three of them immediately started updating their profiles without being asked.

Friday: Decision. The match score prototype had strong signal. Profile simplification was nice but not the lever. I updated the OST: the match score solution moved to "validated, ready for engineering." The profile simplification moved to "park for now."

The following week: Engineering started building the production version of match score. They used the prototype as the spec. Three weeks later, it shipped. Activation improved 28% in the first month.

Total time from signal to shipped feature: 4 weeks. Old model would have been 3-4 months.

The OST wasn't a planning document. It was a learning machine that ran every week. The prototype did the job of months of research in days.

Common mistakes that still apply

Even with AI acceleration, these mistakes will tank your tree.

Too many branches. Your tree should have 3-5 opportunities, max. More than that means you haven't prioritized. AI will surface dozens of potential opportunities. Your job is to pick the ones that

connect to your outcome metric. The rest get parked.

Confusing opportunities with solutions. “Improve search” is a solution. “Freelancers can’t find relevant projects because search doesn’t match their self-described skills” is an opportunity. The opportunity describes the customer problem. The solution is what you build. Keep them separate, even when AI is generating both.

Trusting AI signals without judgment. AI surfaces patterns. It can’t tell you which patterns matter. If AI says “export issues mentioned 200 times,” but 180 of those are from free-tier users and your outcome metric is enterprise retention, that signal might not be your top priority. Apply human judgment to AI output.

Skipping the prototype. Some PMs will use AI to populate the opportunity layer and then go straight to building production features. Don’t. The prototype step exists because your understanding of the opportunity is always incomplete. The prototype reveals what you’re missing. Skip it and you’re back to building on assumptions.

Not killing solutions. The point of rapid prototyping is that it’s cheap to be wrong. But you have to actually kill solutions that don’t work. If 5 customers shrug at your prototype, kill it. Don’t rationalize. Don’t iterate endlessly hoping it clicks. Move to the next solution.

Build your first AI-powered OST this week

Monday (30 min): Pick your outcome. One metric, measurable, bounded by time.

Monday (30 min): Review your signal sources. What customer data do you already have access to? Support tickets, NPS, sales call recordings? If you have an agent running, review the opportunity brief. If not, manually scan last week’s support tickets for the top 5 complaints.

Tuesday (3 hours): Pick your top opportunity. Build a prototype that addresses it. Not a mockup. A working thing.

Wednesday (1 hour): Share it with 3 customers. Watch them use it. Take notes.

Thursday (1 hour): Iterate or pivot based on feedback. Share updated version with 2 more customers.

Friday (30 min): Update your tree. What did you learn? What’s next?

You just ran a full discovery cycle in one week. The tree isn’t perfect. It doesn’t need to be. It needs to be a learning engine that runs every week.

Next week, do it again. New opportunity, new prototype, new learning. After a month, you'll have tested more solutions than most teams test in a quarter.

That's the new speed of product discovery. For the assumption testing layer that sits below the prototype step, see [Assumption Testing](#). For how the interview guide fits into discovery, see [Interview Guide](#).

The Interview Guide That Actually Works

Customer interviews matter more than ever. Show up with full signal context, a working prototype, and AI that synthesizes the conversation in minutes.

Web version: <https://falkster.com/handbook/interview-guide> · last updated 2026-06-12

The short version

Customer interview technique has not changed, but everything around it has. AI agents prepare you before the call by pulling the customer's support history, usage patterns, and NPS score so you walk in already past the surface. During the call, real-time transcription frees you to listen fully instead of splitting attention with note-taking. After the call, a synthesis agent compares the transcript against hundreds of other data points in minutes. The prototype interview format, 30 minutes instead of 45, confirms an agent-identified signal, goes deep on the customer's workaround, and shows a working prototype to get a concrete reaction. Three to five interviews with AI prep and synthesis outproduce ten interviews done the old way.

Why interviews matter more now, not less

You might think that AI signal processing replaces customer interviews. It doesn't. It makes them essential in a different way.

When your agents are processing thousands of support tickets, sales calls, and NPS responses, you have breadth. You know what customers are saying at scale. What you don't have is depth. You don't know why they feel that way. You don't know the context behind the complaint. You don't know the workaround they've built, the emotion behind the frustration, or the thing they haven't said yet because nobody asked.

That's what interviews are for. Depth. Understanding. The human insight that no amount of data processing can replace.

But here's what's changed: you no longer walk into an interview blind. You walk in knowing exactly what this customer has been struggling with. You've seen their support tickets. You've read the sentiment from their NPS score. You've reviewed their usage patterns. And you're carrying a prototype that might solve their problem.

That's a different caliber of conversation.

The AI-powered interview: before, during, after

The interview itself is still a human conversation. But everything around it has changed.

Before the interview.

In the old model, you'd prepare a few questions and hope the conversation went somewhere useful. Maybe you'd reviewed one or two past interactions with this customer.

Now, your prep agent pulls everything relevant about this customer before the call. Their support history. Their usage patterns. Features they use and don't use. How long they've been a customer. Their NPS score and comments. Any mentions in sales notes or CS logs.

You show up knowing: "This customer submitted 3 tickets about export speed in the last month. Their usage dropped 20% after our last release. They rated us a 6 on NPS and wrote 'exports are painful for our team.'"

That changes your opening question from "Tell me about your experience" to "I noticed your team has been running into export issues. Walk me through what happened last time." This is [continuous discovery on autopilot](#) in practice: the agent does the pre-work, you do the human work.

You're already past the surface. You're in the problem from the first minute.

During the interview.

You're still asking questions, listening, probing. That hasn't changed. What's changed is that you have a prototype ready. More on this in a moment.

AI transcription runs in real-time (Gong, Fireflies, Otter, or even the built-in tools in Zoom and Meet). You don't take notes. You listen. You're fully present in the conversation instead of splitting attention between listening and writing.

After the interview.

The old model: spend 30 minutes writing up your notes. Compare manually with other interviews. Hope you remember the important parts.

Now: your synthesis agent processes the transcript within minutes. It extracts: key problems mentioned, emotional intensity, workarounds described, features discussed, competitive mentions, and any commitments or expectations. It compares this interview against the patterns from your signal data. "This customer's export frustration matches a pattern seen in 42 support tickets this month. Their workaround, exporting in small batches, is mentioned by 15% of enterprise users."

You spend 5 minutes reviewing the synthesis instead of 30 minutes creating it. And the synthesis is connected to your broader signal picture, not isolated in a notebook.

The prototype interview: a new format

This is the format I use most now. It's different from the traditional discovery interview and it produces dramatically better insights.

The structure is 30 minutes, not 45.

The old 45-minute interview spent most of its time in exploration. You were trying to find the problem. With signal data, you already have a hypothesis about the problem. So you spend less time exploring and more time validating and deepening.

Minutes 1-3: Context and connection. “Thanks for making time. I’m Falk, I work on [product]. We’ve been digging into how our export experience works for teams like yours. Want to make sure we’re solving the right problems.”

One personal question. Nothing about the product yet.

Minutes 3-10: Confirm the signal. “I know your team has been working with large exports. Walk me through what that looks like for you. What happens when you need to get data out of the system?”

You’re not asking blind. You know from the signal data that this customer has export problems. But you need to hear it in their words. Let them tell the story. Don’t lead. Don’t mention the support tickets. Let them describe the problem from their perspective.

This serves two purposes: it confirms the AI-identified signal is real for this person (not just a pattern artifact), and it gives you the context and emotion that data can’t capture.

Minutes 10-15: Go deep on the workaround. “How do you handle it now when exports time out? What’s your workaround?”

Workarounds are gold. They tell you what the customer values enough to build a manual process around. They show you the shape of the solution from the customer’s perspective.

“We export in batches of 500 because anything bigger crashes. My analyst spends two hours a week combining the batches in Excel.”

That tells you: they need bulk export. The current limit is around 500 records. It’s costing 2 hours per week of analyst time. The solution isn’t just “faster exports,” it’s “eliminate the need to batch.”

Minutes 15-22: Show the prototype. “Based on what we’ve been hearing, we built something. It’s rough, but I’d love your reaction. Here, take a look.”

Share your screen or send the link. Let them interact with it. Don’t explain. Don’t guide. Just watch.

What you're observing: - Do they understand what it does without explanation? - Where do they click first? - Do they try to do something the prototype doesn't support? That's a feature insight. - Do they say "oh nice" politely or do they lean forward and start exploring? - Do they immediately connect it to their problem? "Oh, so I could export everything at once?"

Minutes 22-27: Get the honest reaction. "What's your first reaction? Would this change how your team handles exports?"

Then the critical follow-up: "What's missing? What would make this actually useful for your team?"

This question, asked about a real prototype, produces 10x better answers than "What features would you want?" asked about a hypothetical. Because they've just used the thing. They know what's missing because they tried to do something and couldn't.

"I'd need it to export directly to our data warehouse. Right now we go through CSV and then upload. If this could push straight to Snowflake, that would save us even more."

You just learned that the real solution isn't faster CSV export. It's a direct integration with their data infrastructure. One prototype. One interview. One insight that reframes the entire opportunity.

Minutes 27-30: Close with context. "How important is this for your team? Is this a 'nice to have' or a 'we need this to stay'?"

"Anything else about how your team works that I should understand?"

"Thanks. Super helpful."

That's it. Thirty minutes. You confirmed the signal, understood the context, got a prototype reaction, and uncovered a deeper insight. In the old model, it would take three interviews just to get to the point where you could describe the problem clearly.

Finding the right people to talk to (AI-assisted)

The old model: email a batch of customers and hope the right ones respond. Or ask your CS team for introductions.

The new model: your signal data tells you exactly who to talk to.

For validating opportunities: talk to customers whose signals match the opportunity you're investigating. If you're exploring export problems, your agent can identify the 10 customers who submitted the most export-related tickets, have the highest usage of the export feature, or mentioned export in their NPS response. These people feel the pain most acutely. Their interviews will be the most informative.

For testing prototypes: talk to customers in the target segment for the solution you've built. If your prototype is for enterprise teams, don't test it with solo users. If it's for new users, don't test it with power users who've adapted to the current workaround.

For understanding churn risk: talk to customers the churn predictor has flagged. The [Red Flag detection agent](#) surfaces these accounts daily so you always have a warm list. These are people whose usage dropped, whose NPS went down, or who submitted frustrated support tickets. Their interviews are urgent because they might leave, and valuable because they'll tell you exactly why.

For exploring new markets: talk to prospects who didn't convert. Sales call analysis can tell you which prospects mentioned specific objections or competitor advantages. Those conversations reveal where your product falls short for segments you want to win.

You're not interviewing randomly. You're interviewing surgically. Every conversation is targeted at a specific learning goal, with a specific customer who has demonstrated relevant behavior.

Synthesis at scale: connecting interviews to signals

Here's where the new model really pulls ahead. In the old model, you'd synthesize interviews individually and then manually look for patterns across 8-10 conversations over a month.

Now, every interview is synthesized within minutes of ending and immediately connected to your broader signal picture.

Your synthesis agent processes the transcript and outputs:

Key problems identified. Not paraphrased. Extracted with context. "Customer described spending 2 hours weekly on batch exports because single exports time out above 500 records."

Emotional intensity. Did they describe this as mildly annoying or deeply frustrating? Were they matter-of-fact or animated? This matters for prioritization.

Match to existing signals. "This matches a pattern seen in 42 support tickets and 3 other interviews this quarter. The batch export workaround is mentioned by 15% of enterprise users in ticket data."

New insights not in signal data. "Customer mentioned they'd want direct Snowflake integration. This is new, not mentioned in any previous signal source. Suggest exploring with 3-5 more enterprise customers."

Prototype reaction summary. "Customer engaged with prototype for 4 minutes. Attempted to export a large dataset. Found the flow intuitive but asked about data warehouse integration. Overall reaction: positive with a specific gap."

After 5 interviews over two weeks, you have a synthesis that spans individual conversations and connects them to your full signal picture. Patterns become obvious. And when you spot something new that didn't appear in the signal data, you know it's worth investigating because a human revealed it.

The questions that still work (and one new one)

The core interview questions haven't changed. Story-based questions are still the best way to surface real behavior and real friction.

“Walk me through the last time you did X.” Still the best opening. Forces specificity. Avoids hypotheticals.

“How did you solve that problem?” Surfaces workarounds. Tells you what they value.

“Why?” Asked three times, each time peeling back another layer.

“What would change for you if this was solved?” Tests severity. If the answer is “not much,” the opportunity is weak. If the answer is specific and emotional, it's strong.

“What's your workaround now?” The most underrated question in product discovery. Workarounds are prototypes your customers built for themselves. They show you the shape of the solution.

And the new question, specific to prototype interviews:

“What did you try to do that it didn't let you?” After they use the prototype, this question reveals the gap between what you built and what they need. It's more specific than “what's missing” because it's grounded in something they just tried to do.

The mistakes that still kill interviews

AI doesn't fix bad interview technique. If you lead the witness, you'll get biased data faster. If you pitch instead of listen, you'll validate your assumptions instead of testing them.

Leading with the signal. “We know exports are slow. How bad is that for you?” You've told them the answer. Instead: “Walk me through what happens when your team needs to get data out.”

Pitching the prototype. “We built this amazing new export tool that can handle 10x more records.” Now they feel social pressure to be positive. Instead: “We built something rough. I want your honest reaction. If it's not useful, that's the most helpful thing you can tell me.”

Only interviewing people who match the signal. If all your interviewees have export problems, you'll confirm that export is the top priority. But you might miss that onboarding is actually more important. Mix in some customers from different segments or with different usage patterns. Let the interviews surprise you.

Interviewing too many, learning too little. In the old model, you needed volume because each interview was expensive to set up and synthesize. Now, with AI synthesis and targeted selection, 3-5 interviews on a focused topic are enough to get directional signal. Don't run 20 interviews when 5 will tell you what you need to know.

Ignoring what the prototype reveals. Some PMs show the prototype and then keep asking questions about the problem. The prototype is doing the work. Watch how they use it. That's your data. The questions are follow-up, not the main event.

Your interview plan this week

Monday: Identify your targets. Review your signal data or recent support tickets. Pick one opportunity to explore. Identify 3-5 customers who've demonstrated relevant behavior (submitted related tickets, churned recently, or are in the target segment).

Monday: Build a quick prototype. If you have a hypothesis about the solution, prototype it. Even if it's rough. Having something to show transforms the conversation.

Tuesday: Run two 30-minute calls. Use the prototype interview format. Context, confirm signal, go deep on workaround, show prototype, get reaction, close.

Wednesday: Review AI synthesis. Your transcription tool and synthesis agent should have processed both interviews. Review the synthesis. What confirmed your hypothesis? What surprised you? What's new?

Thursday: One more call if needed. If the first two conversations pointed in different directions, run one more. If they converged, you probably have enough signal. Show the iterated prototype if you've had time to update it.

Friday: Update your OST. Based on this week's interviews and prototype reactions, what did you learn? The [Your First OST](#) chapter walks through how to structure and update the opportunity solution tree from weekly interview signal. Which solutions are stronger? Which should you kill? What should you prototype next week?

Three to four hours of interview time. AI handles the prep and synthesis. The conversations themselves are deeper and more productive than they've ever been because you're not starting from zero.

Interviews aren't the bottleneck anymore. They're the force multiplier. Every other part of discovery can be accelerated or automated. The interview is where human insight happens. Make it count.

The Assumption Testing Playbook

Stop debating what to build. Prototype it in hours, test with customers, and let their reaction decide. Assumption testing just went from weeks to 3-5 days.

Web version: <https://falkster.com/handbook/assumption-testing> · last updated 2026-06-12

The short version

Assumption testing is the practice of identifying the riskiest bets underneath a feature idea and using a prototype to test them in days, not weeks. The prototype is the test. Instead of designing separate experiments for desirability, usability, and viability, you build one working thing that tests all three at once. The one-week cycle: map assumptions Monday morning, build the prototype Monday afternoon, test with five customers Tuesday and Wednesday, synthesize Thursday, decide Friday. Red assumptions (the ones that kill the project if wrong) go first. A failed test is a win. It is the fastest way to kill a bad idea before engineering touches it. At Smartcat, one prototype test saved eight weeks of building the wrong recommendation system and led directly to a shipped feature that moved activation by 28%.

Every decision is still a bet

You want to build a feature. Before you write a single line of production code, you're betting on a stack of assumptions. That hasn't changed. What's changed is how fast you can test them.

Say you want to build an AI-powered job recommendation system for freelancers. Here are the assumptions you're betting on:

Desirability: Freelancers want personalized recommendations. They trust algorithmic curation. They'd rather see fewer, targeted options than browse everything.

Viability: You can build this in 8 weeks. The recommendation engine will be accurate enough to be useful. It moves the activation metric enough to justify the cost.

Feasibility: Your team has ML expertise. Your infrastructure supports it. You can integrate it without a major redesign.

Usability: Freelancers understand "recommended for you." They know why a job was recommended. They use recommendations in their workflow.

Any one of those assumptions being wrong kills the project. The only question is how fast you test them.

The old testing model vs. the new one

The old model: Map your assumptions. Design an experiment for each one. Run experiments over 2-6 weeks. Analyze results. Decide.

Total time: 4-8 weeks of testing before you've built anything.

That was the right approach when experiments were expensive to set up. When building a prototype took weeks. When getting customer feedback required scheduling calls and waiting for availability.

The new model: Map your assumptions. Build a prototype that tests the most critical ones simultaneously. Put it in front of customers this week. Their reaction is the experiment.

Total time: 3-5 days.

The prototype is the test. That's the shift. Instead of designing separate experiments for desirability, usability, and viability, you build a working thing that tests all three at once.

When a freelancer uses your recommendation prototype and says "I wouldn't use this, I prefer browsing everything," you just tested desirability, and it failed. When they use it and say "these recommendations don't match my skills at all," you just tested feasibility (can you build an accurate engine), and it failed. When they use it and immediately apply to three jobs, you just tested desirability and usability, and they passed.

One prototype. One week. Multiple assumptions tested.

The assumption map (still do this)

Even with faster testing, you still need to know what you're testing. Skipping the assumption map and jumping straight to prototyping is a common mistake. You build something, show it to customers, they say "cool," and you've validated nothing specific.

Spend 20 minutes with your team mapping assumptions. Here's the exercise:

Step 1: Write down every assumption (10 minutes). Don't filter. Get them all out.

For the recommendation system: - Freelancers want personalized recommendations - They prefer recommendations to search - They trust algorithmic curation - They'll check recommendations regularly - We can build an accurate recommendation engine - It doesn't require new infrastructure - The UI will be intuitive - Recommendations will increase apply rate - Apply rate increase will improve activation

Step 2: Identify the killers (10 minutes). For each assumption, ask: if this is false, does the whole project fail?

Red (kills the project): “Freelancers want personalized recommendations” and “Recommendations increase apply rate.” If freelancers don’t want them or if recommendations don’t lead to more applications, everything else is irrelevant.

Yellow (important but solvable): “We can build an accurate engine” and “The UI is intuitive.” These might slow you down but won’t kill the concept.

Green (nice to have): “They’ll check recommendations daily” and “It doesn’t need new infrastructure.” Details you’ll figure out.

Step 3: Design the prototype to test the red assumptions.

This is the key. Your prototype isn’t a demo of the solution. It’s a test of the riskiest assumptions. Design it to answer: do they want this, and does it lead to the behavior we need?

The prototype as experiment

Here’s how I design prototypes to test assumptions rather than just showcase solutions.

For testing desirability (“do they want this?”):

Build the simplest version of the solution that a customer can react to honestly. For recommendations: show 10 recommended jobs based on the freelancer’s profile. Make them clickable. Watch what happens.

If they browse the recommendations and apply to 2-3, desirability is validated. If they glance at the recommendations and go back to search, desirability failed. You didn’t need a survey. You didn’t need a focus group. You watched them choose between your solution and their current behavior.

For testing usability (“do they understand it?”):

Don’t explain the prototype. Just share it. “Here, take a look at this. What do you think it does?” If they understand it in 10 seconds, usability passes. If they stare at it confused, it fails. The prototype reveals usability problems instantly because confused people look confused.

For testing feasibility (“can we build it well enough?”):

Use real data in the prototype. Don’t use dummy jobs. Pull actual listings and run them through a simple matching algorithm. If the matches are obviously wrong (“you’re a Python developer, here are 10 graphic design jobs”), the feasibility assumption is shaky. If the matches are reasonable, you have signal that the technical approach works.

For testing viability (“does it move the metric?”):

This one's harder to test with a prototype alone. But you can get a proxy. After the customer uses the prototype, ask: "If this existed in the product, would it change how often you apply?" or "Would this have kept you from leaving?" Their answer alone won't prove anything. Combined with the behavioral signal from how they used the prototype, it points in a direction.

Running the one-week test cycle

Here's the actual schedule.

Monday: Map and build.

Morning: 20-minute assumption mapping with your team. Identify the red assumptions.

Afternoon: Build the prototype. Focus on testing the riskiest assumption. Use AI coding tools to get something functional in 2-3 hours. It doesn't need to be polished. It needs to work well enough for a customer to react honestly.

Tuesday-Wednesday: Test with customers.

Show the prototype to 5 customers. Use the [prototype interview format](#). Each conversation is 30 minutes. Watch them use it. Ask the follow-up questions. Note which assumptions their behavior confirms or challenges.

If you can't get 5 live calls, do 3 live and 2 async. Send the prototype link with a Loom walkthrough and ask for their reaction via email or Slack.

Thursday: Synthesize and iterate.

Review your notes and the AI-generated synthesis from the interview transcripts. For each red assumption, what did you learn?

If the assumption passed (customers engaged, understood it, connected it to their problem), move forward.

If the assumption failed (customers didn't engage, were confused, or went back to their current behavior), you have a decision to make.

If the results are ambiguous, iterate on the prototype and test with 2-3 more customers on Friday.

Friday: Decide.

Three possible outcomes:

Green: Build it. The critical assumptions held up. Customer reactions were strong. Move to production. Use the prototype as the spec.

Yellow: Iterate. The concept is right but the execution missed. Customers engaged but had specific feedback (“I’d use this if it had X”). Iterate on the prototype next week, test again.

Red: Kill it or pivot. The critical assumption failed. Customers don’t want this, or the underlying approach doesn’t work. Kill the solution and go back to the opportunity. Is there a different solution worth prototyping?

When prototypes aren’t enough

Prototypes test desirability and usability well. They’re weaker at testing some types of assumptions. Here’s when you need a different approach.

Testing retention assumptions (“will they keep using it?”): A prototype test tells you if someone is interested in the moment. It doesn’t tell you if they’ll use it next week. For retention assumptions, you need the “leave it with them” test. Give 10-20 customers access to the prototype for a week. Check back. If they used it without being prompted, the retention assumption has signal. If they forgot about it, it doesn’t matter how excited they were in the demo.

Testing scale assumptions (“does this work at volume?”): Your prototype might work beautifully for 10 users. But if it depends on manual curation, personalized content, or high-touch support, it won’t scale. For scale assumptions, run a concierge test first (manually do the thing for 50 users), measure the impact, then ask: can we automate what the concierge did?

Testing pricing assumptions (“will they pay for this?”): Prototypes test value, not willingness to pay. For pricing, use a smoke test. Put a “premium” badge on the prototype with a price. “This feature is available on our Pro plan for \$X/month.” Measure how many people click through versus bounce. It’s not perfect, but it’s directional.

Testing technical assumptions (“can we build this at production quality?”): Your prototype is a hack. Can your team actually build the production version in the estimated time? For technical assumptions, have your engineer spend a day on a spike. Not building the feature, just building the hardest part. If the spike works, the assumption holds. If it surfaces unexpected complexity, adjust your timeline or approach.

The kill decision (the hardest part)

Everything I’ve described makes testing fast. But the hard part was never testing. It was acting on the results.

When a test fails, most PMs rationalize. “The sample was too small.” “The prototype wasn’t polished enough.” “We just need to iterate more.” “Let’s build the real version and see.”

Don't do this. If 5 customers don't engage with your prototype, adding polish won't fix it. The concept didn't land. The assumption was wrong.

I killed a recommendation project at Smartcat based on prototype testing. We showed personalized job recommendations to 8 new freelancers. Two engaged. Six went back to search. When I asked why, the pattern was clear: they wanted to see all their options, not a curated set. They valued visibility over optimization.

We could have rationalized. We could have said the algorithm needed tuning. We could have built the full version hoping adoption would be different at scale.

Instead, we killed it. Pivoted in a week. Built better search filters and a "browse all projects in your category" view. Shipped in three weeks. Activation improved 28%.

The kill saved us 8 weeks of building the wrong thing. The pivot took 3 weeks to build and ship. Total time from "bad idea" to "working solution that moved the metric": 4 weeks. In the old model, we'd have spent 8 weeks building recommendations, shipped them, measured 5% adoption, spent 4 more weeks iterating, and eventually killed it anyway. That's 12 weeks wasted instead of 1.

The kill decision is the highest-leverage decision you make. Testing just makes it cheaper and faster to get there.

AI-powered assumption tracking

One thing AI does better than any human: keeping track of what you've tested and what you haven't.

Set up an assumption tracking agent. It maintains a living document of every assumption across every active project. For each assumption, it tracks: current status (untested, testing, validated, invalidated), the test method used, the result, and the date.

Every Friday, the agent generates a report: "You have 12 active assumptions across 3 projects. 7 are validated. 2 are invalidated. 3 are untested. The untested assumptions are: [list]. Recommended: test assumption X next week because it's a red-risk assumption on your highest-priority project."

This sounds simple but it solves a real problem. In practice, teams test the easy assumptions and avoid the scary ones. The ones that might kill the project. The agent doesn't have that bias. It flags the untested red assumptions every week until you test them. The build is written up as [the Testable Assumptions Tracker Agent](#).

It also connects to your signal data. "Assumption: Enterprise customers want bulk export. Signal update: 15 more support tickets about export this week, 60% from enterprise accounts. This assumption is getting stronger based on signal data alone, but has not been prototype-tested yet."

The assumption tracker turns testing from an ad-hoc practice into a system. Every assumption gets tracked. Nothing falls through the cracks.

For the full agent setup, see [Your AI Agent Fleet](#).

Building a testing culture (faster now)

In the old model, building a testing culture took months. You had to convince your team that spending 4-6 weeks on experiments before building was worth the delay. That was a hard sell when leadership was pushing for shipped features.

The new model makes this sell easier because the “delay” is measured in days, not weeks.

Week 1: Pick one project. Map assumptions. Build a prototype. Test it with 5 customers. Show your team the results. Total investment: one week.

If the test validated the approach, you saved your team from building blindly. If the test killed the approach, you saved 8 weeks of wasted engineering. Either way, the ROI is obvious.

Week 2: Do it again with the next project. This time, invite your engineer and designer to the customer prototype sessions. Let them see the reactions firsthand. When an engineer watches a customer struggle with the prototype and say “I’d need it to do X instead,” the engineer is now invested in building the right thing.

Week 3: Your team starts asking for it. “Did we test this assumption?” “What did customers say about the prototype?” “Can we show this to a few more people before we commit?” That’s the culture shift. It happened in three weeks instead of three months because the cost of testing dropped to near zero.

Month 2: Testing is the default. No one starts building a production feature without showing a prototype to customers first. Not because you mandated it. Because they saw the difference between building confident and building blind.

Your test this week

You have a project in mind. Stop reading and do this.

Right now (10 minutes): Write down the top three assumptions your project sits on. Which one, if wrong, kills the whole thing?

Today (2-3 hours): Build a prototype that tests the riskiest assumption. Not a mockup. A working thing. Use AI coding tools. Describe what you want and iterate until it’s functional enough for a customer to react to.

Tomorrow (2 hours): Show it to 3 customers. Use the prototype interview format. Watch them use it. Listen to their reaction.

End of week (30 minutes): Decide. Did the assumption hold? Build, iterate, or kill?

You just compressed a month of testing into a week. The prototype did the work of multiple experiments. The customer reactions gave you real data, not hypothetical answers.

The companies that win are the ones that get to the right decision fastest. A prototype in front of a customer tests more assumptions in 30 minutes than a month of surveys and data analysis. Build it, show it, learn, decide.

Start today.

Continuous Listening: Every Customer, Every Day

Weekly customer calls are the floor, not the ceiling. Pipe every support ticket, call, and churn survey into a daily digest and stop scheduling discovery.

Web version: <https://falkster.com/handbook/continuous-listening> · last updated 2026-07-22

The short version

Continuous listening is a daily pipeline that ingests every customer signal source (support tickets, call transcripts, NPS surveys, churn reasons, product rage clicks), synthesizes overnight by theme, and surfaces the top clusters every morning in a five-minute digest. Weekly 1:1 interviews with Teresa Torres-style continuous discovery are still essential, but they now serve a different purpose: understanding the why behind a cluster you already detected, not finding the cluster in the first place. A PM with Claude Code can wire the v1 pipeline in two weeks. The old model, 5 calls a week as the only signal channel, gave you 250 data points per year from your most cooperative customers. The pipeline gives you thousands per day from everyone, including the customers about to churn who never take your calls.

The weekly cadence is no longer the bar

The Continuous Discovery chapter raised the bar from quarterly to weekly. Most PMs are still trying to get to weekly. That's table stakes now.

The actual bar in 2026 is continuous. Every support ticket, every sales call recording, every churn survey, every in-product rage click, every email response, synthesized and clustered overnight by agents, surfaced as a daily signal digest. The weekly customer interview is the floor. Not the ceiling.

I wake up every morning to a list of what roughly 3,000 Smartcat customers said yesterday, clustered by theme, ranked by impact. I read it in five minutes before my first meeting. I know what's happening in my product before any stakeholder does. That's not a research cadence. That's a pipeline. The [AI Agent Fleet](#) gives you the full setup for the overnight synthesis and clustering agents that make this work.

Why “weekly customer conversations” hits a ceiling

Weekly conversations were a giant leap over quarterly research. [Teresa Torres](#) was right, and the [Continuous Discovery Autopilot](#) chapter covers that foundation in depth. The discipline separates great PMs from good ones. But the practice has hard limits.

Sample size. 3 to 5 calls per week is 150 to 250 customers per year. My product has 10x or 100x more customers than that. The 5 I spoke to are not a representative sample of the system. They're a representative sample of *customers willing to take a 30-minute meeting with me*, which is a different population.

Selection bias. Customers who say yes to research calls are disproportionately my power users, my most engaged, my friendliest. Customers about to churn don't take the meeting. Customers who silently moved to a competitor don't take it. The signal I need is in the silence, and the silence is invisible to a weekly call cadence.

Synthesis lag. Even the disciplined PM batches synthesis. I'd take notes, let them pile up, do a Friday review. The customer who said something critical on Monday got acted on at best the following Tuesday. In an AI product where context shifts daily, that's an eternity.

Single-channel listening. A 1:1 call is one channel. Customers are also writing support tickets, replying to NPS surveys, leaving reviews, posting on Reddit, complaining in community Slack, churning silently. Every one of those is signal. Most PMs read none of them systematically.

The five-component system I run

Built once, runs forever.

1. Every signal source, piped in.

List every place customers leave evidence of how they feel about your product:

- Support tickets and email
- Sales call recordings ([Gong](#), [Chorus](#), [Fireflies](#))
- Customer success call recordings
- In-product feedback widgets
- NPS and CSAT survey responses
- Churn cancellation reasons
- App store reviews
- Product Hunt, G2, Capterra reviews
- Public social mentions
- Community Slack, Discord, forum posts
- In-product rage clicks, dead clicks, behavioral anomalies

Most companies have 6 to 10 of these flowing somewhere. Most PMs are connected to 1 or 2. The system starts by piping all of them to a single ingestion point.

2. Overnight synthesis.

An agent runs every night against the previous day's haul. For each source it:

- Extracts key statements (verbatim quotes, with source links).
- Tags by surface (which part of the product).
- Tags by sentiment (negative, neutral, positive, or finer-grained).
- Tags by intent (bug, feature request, confusion, praise, churn signal).

The output is structured: a list of statements, each with metadata, all linked back to the original source.

3. Clustering.

A second agent groups statements into themes. “47 customers mentioned the new sync flow being slow.” “23 customers asked for a way to undo the bulk action.” “12 customers mentioned the same wording in onboarding being confusing.”

Each cluster has a count, a representative quote, source links, and a trend (is this growing day-over-day?). A single comment is anecdote. Fifty comments saying the same thing is a product bug.

Cluster the outcome, not the feature request. This is the discipline that makes the pipeline worth having. A customer who asks for a “CSV export button” is handing you a feature. The outcome underneath is “get this data into my board deck without re-keying it,” and a dozen different builds could serve it. Tag clusters by the outcome the customer is reaching for, not the solution they happened to name. Feature requests fragment into a hundred one-off asks. Outcomes collapse into ten or fifteen jobs, and a job is something a prototype can actually go after.

4. The morning digest.

Every morning I (and my team, and anyone who wants to sign up) get a one-page digest:

- Top 5 clusters, ranked by impact (count times severity times velocity).
- New clusters that emerged in the last 24 hours.
- Clusters that grew week-over-week by more than X.
- A short narrative paragraph the agent writes summarizing what changed.
- Direct links into the underlying conversations.

I read it with coffee in five minutes. I know what my customers said yesterday before my first meeting starts.

5. The action loop.

Every cluster has a status: new, acknowledged, investigating, scheduled, fixed. I review new clusters daily. I make a call: act on this, or note-and-watch? Acting becomes a bet on the portfolio. Original

conversations stay linked so when I ship the fix I can reach back and tell those customers individually: “You mentioned this. Here’s what we did.”

That last move is the one most teams skip. It’s also the one customers remember. The cluster of 47 customers who complained about slow sync becomes 47 personalized “we heard you, we shipped this” emails when the fix lands. That’s where retention is built.

What 1:1 conversations are for now

Continuous listening doesn’t replace 1:1 conversations. It changes their purpose.

Old purpose: discover what customers want. **New purpose:** understand the *why* behind a cluster you already detected.

I no longer need the conversation to surface the issue. The digest already did. I need the conversation to deeply understand it, test framings of the solution, uncover upstream causes the digest can’t infer. The conversation becomes high-leverage interpretive work, not detective work.

This also changes who I talk to. The digest tells me which customers are inside a cluster. I call those customers specifically, the ones whose silent behavior or written words flagged them. Not the ones who happened to be free for a recurring research slot. Better sample. Better signal.

The objections

“This sounds expensive to build.” It isn’t in 2026. Open-source agent frameworks. A connector to your support tool. A connector to your call-recording tool. An LLM with a clustering prompt. A Slack output. A capable PM with Claude Code can wire v1 in a week. V2 (cleaner UI, better dedup, customer linking) takes another week. Two weeks of work for a permanent listening pipeline is the highest-ROI project on your team’s docket.

“Won’t agents miss things?” Yes. They’ll miss subtleties, sarcasm, multilingual nuance, regional context. They get the gist of a conversation right and the texture wrong. Two responses: first, texture is exactly what your weekly 1:1s are for now. Second, missing some texture across 3,000 conversations is still infinitely more signal than perfectly capturing 5 conversations.

“What about privacy?” Real concern. Audit the data flow. Mask PII before it enters the synthesis pipeline. Use vendor terms that prohibit training on your data. Document the practice in your trust center. Not a reason to skip the system. A reason to build it carefully.

“My company won’t let me touch this data.” Then your company doesn’t have a customer feedback culture. It has a customer feedback bottleneck. Find the smallest data source you do have

access to (in-product widget, app store reviews) and start there. The proof of concept will eventually get you the rest.

Pick one thing this week

Don't wait for the full pipeline. Start tomorrow morning. For a practical walkthrough of building your first discovery agent stack, see [Build Your Discovery Agent Stack](#).

1. Identify one customer signal source you have access to right now (support tickets, sales call transcripts, NPS responses, whatever).
2. Pipe the last 30 days into Claude. Ask it to cluster by theme and rank by count.
3. Read the output. Notice three things you didn't know.
4. Pick the cluster that's most actionable and bring it to Monday's team meeting.
5. Next week, automate it. A small script, a cron job, a Slack output. Make it run daily.

That's your v1. One source. One cluster output. Daily. Within a quarter you're adding sources until you have the full pipeline. Stop scheduling time to talk to customers. They're talking to you all day. Open the channel.

Part 3: Build & Ship

Prototypes, not specs. Evals, not reviews..

Prototype Before You Spec

Why the fastest way to get alignment, test ideas, and advance your career is to build something people can touch - and exactly how to do it in 2 hours.

Web version: <https://falkster.com/handbook/instant-prototyping> · last updated 2026-07-22

The short version

Prototype before you spec. The PM who walks into a meeting with a working prototype beats the PM with slides every time. The 2-hour prototyping method: frame the problem in three sentences (who, what they're trying to do, how you'll know it's solved), build the core interaction in Claude Code or Cursor using experience-first descriptions, test immediately with one real person without explaining anything, then decide to kill or iterate. Vibe coding is not writing production code. It's translating a customer problem into a working artifact using clear English and conversational iteration. If you can write a clear email, you can do this. The barrier is artificial. Pick something you're genuinely unsure about and block two hours this week.

The power move nobody teaches you

Here's something they don't tell you in PM school: the PM who walks into a meeting with a working prototype gets more leverage, more influence, and more credibility than the PM who shows slides.

I watched this play out firsthand. Two PMs, same company, competing for a \$2M feature budget. The first PM brought a 30-slide deck with detailed user research, competitive analysis, and a fully specified PRD. Impressive. It took him three weeks.

The second PM brought a 10-minute prototype. You could click through it, try it yourself, feel the friction. It took her two hours.

Guess who got the budget? Guess who got promoted six months later?

The person with the prototype.

There's something about working software that short-circuits the part of people's brains that wants to debate abstract concepts. You can't argue with a prototype. You can only experience it. And once someone experiences your idea, it becomes real to them. The outcome becomes inevitable instead of theoretical.

This is the actual job of product management in 2026. Not writing specs. Rapidly converting ideas into something tangible. The spec is your old playbook. The prototype is your new superpower.

Why specs are the enemy of speed (said with kindness)

Let me be direct: PRDs are theater. Well-intentioned theater, but theater nonetheless.

Here's what happens. You spend a week writing a 40-page PRD. You nail the success metrics. You anticipate edge cases. You feel rigorous and professional. It looks good on the project tracker.

Then engineering reads the first three pages, skims page 4, and asks the same questions they would have asked anyway. Design comes back with mockups that don't match your vision because they interpreted the spec differently. Marketing wants to see it and suddenly we're in a 90-minute "spec review" that creates four new conflicting requirements.

By the time you're done clarifying, you've spent 20 hours on a document that bought you nothing except the illusion of certainty. The document is now wrong because the world changed while you were writing it. Nobody reads it again.

But worse: you've trained your team to believe that talking about building something is the same as building it. That you can design your way out of uncertainty. You can't. Uncertainty doesn't resolve until you build and test.

The spec creates false precision. It makes everyone feel like a problem is solved when really you've just written down your guesses in formal language. And then when reality doesn't match the spec, the team wastes time arguing about who misunderstood instead of adapting.

Prototyping is the antidote. A prototype is honest. It shows you exactly what works and what doesn't. It can't be misinterpreted, can't be partially read. It forces you to make tradeoffs visible instead of hiding them in strategic ambiguity.

The best part: prototyping is faster. You'll have a working version before you could even finish section 3 of your requirements doc.

The 2-hour prototyping method

This is a system, not a suggestion. Follow it exactly once and you'll see what I mean. Then you'll adapt it to your style.

Frame the problem (15 minutes)

Before you touch any tool, answer three questions in the clearest language you can find:

1. **Who has this problem?** Not "users" or "the market." Be specific. "Sales reps at mid-market SaaS companies who manage 50+ accounts" beats "salespeople." Specificity matters because different people have different problems.

2. What are they trying to do? What's the job they're trying to get done? Not what you think they should do. What are they actually trying to accomplish in the moment when this problem hurts? "I need to find the top 5 accounts most likely to churn in the next quarter so I can proactively reach out" beats "improve retention."

3. How will we know if we solved it? What changes if this works? Is it faster? Cheaper? Less stressful? "A sales rep can identify churn-risk accounts in under 5 minutes instead of 45 minutes of manual analysis" beats "better retention" because it's falsifiable.

Write your answers down in three sentences total. One sentence per question. If you need more than three sentences, you don't understand the problem yet. Keep going until you can say it simply. This is the hardest part and also the most important part.

Show these three sentences to someone. Anyone. If they look confused, you're not clear. Revise. This clarity compounds through the rest of the process.

Build the core interaction (30 to 90 minutes)

Now open Claude Code, Cursor, or Claude.ai - your choice based on what you're building. (For web-based prototypes, start with Claude in the web interface. For more complex things, use Cursor.)

Paste in your three sentences. Then describe what you want to build with the same clarity.

Don't describe the interface. Describe the user experience. The difference matters.

Bad: "Build a dashboard with charts showing account health metrics and a filter for churn risk."

Good: "A user opens their account list, clicks an account name, and instantly sees a red/yellow/green churn risk score. They can drill into the factors driving that score (payment delays, declining usage, etc.) and see which accounts have dropped off the most in the last 30 days."

The second version tells the AI what moment of value you're trying to create. It focuses on experience, not implementation. The AI will figure out the how.

A few things that actually work here:

- Use real scenarios. "A sales rep named Alex has 67 accounts. She wants to know which 5 are most likely to fire her this quarter." Real names, real numbers, real friction. This creates better prototypes than abstract descriptions.
- Describe the happy path first. What happens when everything goes right? Build that. Ignore error states, edge cases, and nice-to-haves. You can add those in iteration 2.
- Tell the AI to use real data if you have it. Fake data creates fake confidence. If you can grab a CSV of actual customer accounts, account health metrics, or whatever real data exists, feed it

into the prototype. Real data surfaces real problems faster.

- Iterate conversationally. “The chart is good but I need it to show percentages too” beats starting over. Each tiny improvement takes 30 seconds. Have a conversation with your tool. That conversation is where the actual design work happens.
- Don’t polish. This is not a design exercise. The prototype’s job is to answer a question, not look beautiful. Ugly works. Shipped ugly beats unpublished perfect. If your prototype makes someone uncomfortable because the design is rough, great. That’s the point. The question isn’t “do people like how this looks?” It’s “does this solve the problem?”
- Deploy it somewhere real people can access it. A prototype on your laptop is just a demo. Host it on Replit, Vercel, a simple Google Sheet, a Figma prototype, whatever. Make it live. Send them a link. Watch what happens when they don’t have you sitting next to them explaining.

The whole phase should take 30 to 90 minutes depending on complexity. A simple interactive prototype: 30 minutes. Something with data and multiple screens: 60-90 minutes.

Test immediately (30 minutes)

Here’s the non-negotiable rule. Watch someone use it without explaining anything.

Find a customer if you can. If you can’t, find someone on your team who wasn’t involved in building it. Show them the live prototype and ask them one question: “What would you do first?”

Then step back. Watch them click. Watch them hesitate. Watch them look for something that doesn’t exist. Watch where they get confused.

Don’t help. Don’t explain. Don’t defend. Just watch.

You will learn more in 30 minutes of observation than in 30 days of meetings and surveys combined. You’ll see what people actually do vs. what they say they’d do. You’ll spot the moment when your idea doesn’t work because they’ll look lost. You’ll see the accidental feature nobody planned for because someone will instinctively try it.

Take notes. Lots of notes. Not on what they say but on what they do. Did they understand the core idea immediately or did it take explanation? Did they find the churn risk score or did they look for it in the wrong place? Did they try to do something the prototype doesn’t support?

That’s your raw data. That’s your market feedback in its purest form.

Decide: kill, iterate, or graduate (15 minutes)

After you’ve tested, you have exactly three decisions:

Option 1: Kill it. The core idea doesn’t work. Users don’t understand it, or they understand it and don’t want it, or something deeper is broken. That’s okay. You just spent 2 hours instead of 2 months

finding out. Celebrate this. A killed prototype saves 10 weeks of engineering time. Kill often.

Option 2: Iterate. The idea has legs but needs refinement. Users got the core concept but needed clearer labeling. Or they wanted it but with one key feature you didn't build. Go back to Step 2. Have the AI add that feature. Build iteration 2. Test iteration 2. Decide again.

Option 3: Graduate it. The idea works and the reaction was strong enough to build for real. Graduating does not mean shipping the prototype. The prototype was disposable, a two-hour argument you made in working code. What graduates is the decision, plus an eval that defines what good looks like. You hand engineering a working prototype to harden and an eval to gate it against, not a spec to interpret. Most prototypes never reach this option, and that is the point: the eval bar decides what earns engineering time, not the loudest voice in the meeting. See [Old PM vs Product Builder](#) for the handoff and [The Eval Is The Spec](#) for the gate.

Repeat the kill-or-iterate loop three times maximum. By the third iteration, you should know if this is real or not. If you're still uncertain after three cycles, the problem isn't the prototype. The problem is you haven't identified who the real user is or what their real problem is. Go back and talk to more customers.

Vibe coding is not coding

Let me be clear about what "vibe coding" actually means. You don't become a developer. You become fluent in the language of building.

You learn to translate a feeling, a hunch, a market insight, into working software without the traditional handoff ceremony. You stop writing specs to engineers. You stop losing meaning in translation. You build a direct pipeline from intuition to artifact.

This requires exactly four skills:

1. Describe what you want in clear English. Not pseudo-technical language. Not bulleted lists of features. Story language. "When Sarah opens the app, she sees her 10 riskiest accounts ranked by likelihood to churn." If you can describe it clearly, Claude can build it.
2. Iterate conversationally. This is easier than you think. "Can you make that interactive so I can click to see more details?" That's it. That's the whole skill. You can improve anything in seconds by asking questions.
3. Know when something works. You don't need to know code to feel when a prototype solves the problem. Your intuition gets better every time you build.
4. Deploy it. Use Vercel for React, Replit for anything, Google Sheets if it's data. You need people to be able to click a link and experience your idea. That's the whole game.

You're becoming a builder, not an engineer. There's a massive difference.

The barrier to entry here is totally artificial. You're not writing production code. You're not managing database architecture. You're describing ideas and refining them through conversation. If you can write an email clearly, you can do this.

When prototyping works (and when it doesn't)

Prototype when:

- You're testing whether an idea resonates with real customers. Before you build for real, see if the real problem is what you think it is.
- You need to align a divided team on vision. Nothing ends a debate faster than a working prototype. People who disagree about abstract concepts agree when they can interact with something.
- You're validating demand before investing engineering time. "Would you use this?" is theoretical. A prototype with a signup link is data.
- You're exploring a UX concept and you're unsure about the optimal flow. Users will show you the right flow instantly.
- You need to demonstrate technical feasibility. Can it be done? Prototype it and answer the question.
- You're onboarding a new engineer and need them to understand the vision. A prototype is faster than any document.

Don't prototype when:

- You already know exactly what to build and engineering is waiting on you. The prototype phase is done. Time to spec and build the real thing.
- The problem is purely backend/infrastructure. You can't prototype database architecture or API design with a user-facing prototype. (Though you might prototype the backend separately with different tools.)
- You're optimizing an existing feature for speed or efficiency. If the feature already exists and works, use analytics and user testing instead. Prototyping a 10% faster version of something people already use is the wrong move.
- You're building something where mistakes are expensive. Medical devices, financial systems, infrastructure - these need rigorous engineering from the start. Prototyping doesn't replace that.

The continuous discovery thread

This is the piece most PMs miss: prototyping doesn't exist in isolation. It connects directly to your discovery practice.

You talk to customers and learn that churn is their #1 problem. You prototype a churn-prediction feature. They love the prototype. You ask them, "If we shipped this, what would you do differently?" They tell you exactly. That answer informs the spec.

Then you ship it. Customers use it. Their usage pattern is slightly different from what they said they'd do. You iterate. That iteration teaches you about the next problem. You prototype that.

This is the discovery loop: **Customer insight** → **Prototype** → **Test** → **Learn** → **Next insight** → **Prototype again**.

The teams that win are the teams that move through this loop fastest. They're not the teams with the most rigorous specs. They're the teams that build, learn, and adapt continuously. Prototyping is how you move fast through that loop. For the discovery practice that feeds this loop with the right problems to prototype, see [Continuous Discovery on Autopilot](#).

The organizational friction point (and how to move through it)

Here's where it gets hard: you work in an organization that loves specs. Maybe it's a mature company. Maybe it's an organization where people have spent 15 years getting really good at writing requirements documents. Maybe your executives think a 40-page PRD means you're doing serious work.

They will resist prototyping. You'll prototype something amazing. You'll show it around. And someone will ask, "Where's the spec?" Or they'll say, "This is interesting but we need a proper requirements document before engineering can start."

This is where most PMs back down. They shouldn't.

Here's your move: don't fight the culture directly. Leapfrog it with results. Prototype. Test with real customers. Get data. Then write a short spec that reflects what you learned from the prototype. You're not replacing the spec. You're replacing the three weeks of speculation with three hours of learning first.

The spec gets better because it's based on reality instead of guesses. The team gets more excited because they saw the prototype work. Engineering starts faster because they understand the problem, not just the requirements.

One successful prototype-first cycle changes the conversation. By the third one, your organization won't ask for specs upfront anymore. They'll ask for prototypes.

How this connects to your career

Here's the real reason you should care about this beyond the cycle time savings:

The PMs who get noticed, who get promoted, who become go-to people at their companies - they're the ones who make things real. Not on paper. Actually real.

They walk into a room and say, "I built something. Let me show you." That's a different energy than "I wrote a spec. Here's what I think we should do." One is theory. One is evidence.

People follow the PMs with evidence. People argue with the PMs with theories.

When you start building before spec'ing, you move faster than everyone else. You accumulate data faster. You learn faster. You look like you have better instincts because you have more data to back them up. You become the PM other teams want to work with because you actually know what you're asking for, not just what you wrote down.

This compounds. The PM who shipped five prototypes in the last three months has more credibility, more seniority, more career momentum than the PM who wrote five PRDs. That's not fair. That's just how influence works.

Your first prototype this week

Pick something small, something you're honestly unsure about, and something that, if it works, matters to your business.

Here are some ideas depending on your context:

- **If you're at a B2B SaaS company:** Pick the #1 problem your most frustrated customer segment complains about. You know it's real. You don't know if a specific solution would work. Prototype it.
- **If you're at a marketplace:** Pick a friction point you suspect is losing transactions (hard to find sellers, hard to coordinate, shipping complexity). Prototype a fix.
- **If you're building internal tools:** Pick a report someone is constantly asking for and prototype a self-serve version so they can build it themselves.
- **If you're at an early-stage startup:** Pick the feature you're least confident about. The one where you've had the most customer conversations and still aren't sure what to build. Prototype that.

Block two hours tomorrow. Or this Friday. Tell no one. Open Claude. Describe what you want to build. Deploy it. Send the link to a customer or a colleague.

Watch what happens.

I promise you: you'll learn something in two hours that a month of meetings wouldn't have taught you. And you'll feel different about your work. You'll feel like a builder instead of a document writer.

That difference is everything.

For a minute-by-minute walkthrough with exact prompts, see [How to Build a Working Prototype in 60 Minutes](#). To see the Instant Prototype Agent that automates this loop for customer feature requests, see the [Instant Prototype Agent](#).

The Impact Loop

The daily rhythm that replaces sprints, stand-ups, and roadmap reviews. Sense what's happening, build a response, measure the impact, amplify what works.

Web version: <https://falkster.com/handbook/impact-loop> · last updated 2026-06-12

The short version

The Impact Loop is a four-beat operating rhythm: Sense (know what is happening), Build (make a working response instead of a plan for one), Measure (quantify what actually changed), Amplify (scale what works and kill what does not). It replaces sprints because sprints optimize for predictability and the Impact Loop optimizes for responsiveness. The loop runs continuously, not on a fixed cadence. AI agents handle the sensing layer automatically, surfacing a two-minute daily brief. Prototyping takes hours, not weeks. Measurement is automatic and daily. A full loop from customer signal to validated, profitable change took eight days in the Smartcat example here. Compare that to the thirty-plus-day waterfall most teams run without noticing.

Why your process is making you slower

I'm going to say something that'll get me hate mail from the Agile certification people: Scrum, SAFe, Kanban, and Shape Up were designed to solve a problem that doesn't exist anymore.

These frameworks all share a core assumption: predictability and control. You estimate work. You batch it into sprints. You commit to deliverables. You measure velocity. The entire system is optimized so that you know, with some accuracy, what will ship in two weeks.

That made sense in 2008, when shipping was expensive and slow. Now? Your customer needs change Tuesday morning. Competitors launch something Friday afternoon. Your analytics tell you something broke or an opportunity opened yesterday. The person who spent two weeks estimating work in a sprint planning meeting is now slowing you down.

I'm not saying Scrum is evil. It served a purpose. It taught the industry that you could ship incrementally instead of hoarding code for six months. That was real progress. But if you're still using a process built for predictability when your actual job is to respond faster than anyone else, you're optimizing for the wrong thing.

The Impact Loop doesn't optimize for predictability. It optimizes for responsiveness. For speed. For actually moving customer behavior, revenue, and retention metrics that matter.

Four beats, continuous rhythm

Think of the loop like breathing. Sense in. Build out. Measure and see. Amplify what works. Then loop back. No waiting for sprint planning. No ceremony. Just rhythm.

SENSE: Know what's actually happening

Before anything else, you need clarity. What's happening with your customers? What's the market doing? What signals are screaming for attention and what's just noise?

Most PMs get this wrong. They look at a dashboard and call that sensing. A dashboard shows you what was configured to show you three months ago. It's a snapshot of yesterday's questions, not today's problems.

Real sensing means: - **Customer signals:** Support tickets, churn reasons, feature requests, but also the subtext. The customer who says "I need better reporting" might actually be saying "I don't trust my data." - **Behavior signals:** How are users actually moving through your product? Where do they get stuck? Where are they succeeding faster than expected? - **Competitive signals:** What did your competitors ship? Did they steal a feature you were building? Did they find a white space you missed? - **Market signals:** Is your ICP growing more cautious? Are budgets shrinking? Is a new regulation coming that impacts your compliance requirements?

In the old world, you'd read spreadsheets and talk to customer success. In the new world, you have AI agents that can monitor all of this continuously and surface only the patterns that matter.

Here's what that looks like in practice: Every morning, instead of digging through Slack or email or dashboards, your sensing layer - powered by four core agents: **Red Flag Detection Agent**, **Competitive Intelligence Agent**, **Support Signal Processing Agent**, and **NPS/CSAT Analysis Agent** running daily - gives you a 2-minute brief. "Trial conversion is down 18% in the Enterprise segment. Support is seeing confusion around column-based access. Competitor X just announced role-based permissions. Customer Y, one of your top 5 accounts, had 6 support tickets this week."

Your job isn't to notice these signals. Your job is to judge them. Which one matters most right now? The conversion dip is a revenue problem. The access confusion is a retention problem. The competitor announcement is a 6-week headache. Your sensing agent found all three. You decide which one demands attention today.

That judgment - which signal is worth your focus - is irreplaceable. AI is excellent at pattern detection. You're excellent at knowing what patterns move the needle.

BUILD: Make the response real, not a plan for it

This is where a lot of PMs break down. You sense a problem, and then you do this: - Write a requirements document (3 days) - Get design to mock it up (5 days) - Present to engineering (2 days of async feedback, 1 meeting) - Engineering estimates it (1 day) - 2 weeks waiting for the sprint to start - 2 weeks of building (hopefully) - 1 day of QA

You just spent 30+ days deciding whether a hypothesis was true. By then, customer behavior may have shifted. Competitors moved. Your sense of urgency faded.

The Impact Loop flips this. You don't plan. You build. You make a prototype of your idea and test it with real customers and real data.

Most features don't need the full engineering treatment to validate. Call it 80%. They need a prototype. Something clickable. Something you can show a user. Something you can run an A/B test on.

Your job in the BUILD phase: 1. Define the hypothesis clearly: "If we simplify the onboarding for Enterprise customers, their setup completion rate will increase." 2. Work with your AI development partner to build a prototype. Not a sketch. Not a figma file. Something real. Something running. You might spend 4-6 hours on this. You describe the problem, you iterate: "Make the onboarding simpler" → "Add a progress indicator" → "Show smart defaults instead of blank fields" → "Add a short video for the hardest step." 3. Review it yourself. Does it actually address the problem you sensed?

For most ideas, you'll prototype, measure it, kill it, and learn something. Killing it fast is the win. That's 5 days instead of 30 days. That's the compounding advantage right there.

For the 15% of ideas that show real promise, you hand off the prototype to engineering. Now they're looking at a working prototype and a mountain of data saying "customers respond to this." No 40-page spec that may not even be correct. Their job gets faster, easier, and more informed.

MEASURE: Quantify what actually happened

Here's where most impact loops break: PMs measure the wrong things.

Vanity metrics feel good but they lie. Users visited your new page? Great. Did they convert? Did they stay? Did they tell a friend? Did you make money? Those are the only metrics that matter.

An impact metric answers one question: Did customer behavior change the way we hoped? The **Feature Adoption Agent** and **OKR Tracker Agent** run daily at 4pm, giving you real outcome data automatically without waiting for manual analysis.

If you shipped a simplified onboarding, the impact metrics are: - Setup completion rate (the behavioral change you wanted) - Time-to-first-value (does it happen faster now?) - Trial-to-paid

conversion (does simplicity actually drive revenue?) - Onboarding support tickets (did you reduce confusion?)

You don't measure one of these. You measure all of them. Context matters. Maybe completion went up but conversion went flat - that tells you something important. People get through faster but aren't convinced. The problem isn't speed, it's clarity of value.

In your loop, measurement is automatic and daily. Your analyst agent: - Sets up tracking for every feature you ship (no manual instrumentation, no waiting on engineers) - Runs the experiment with proper controls (A/B test or staged rollout) - Analyzes results with statistical rigor (not "we shipped it and it feels good") - Reports daily until you have enough data to decide

You don't wait for "the results." Results flow in continuously.

Your job is to interpret them. Is a 12% increase in completion meaningful? Depends on sample size, depends on the business impact, depends on how much effort this cost. Your analyst tells you the number. You tell it what it means.

AMPLIFY: Scale wins, kill losers, apply learnings

This is the easiest beat and yet most PMs skip it. If something works, you expand it. If something doesn't, you kill it fast and move on.

Amplification looks like: - **Double down on winners:** The simplified onboarding worked for Enterprise. Test it for mid-market. Expand it to all customers. Brief engineering to build the production version so you're not running a prototype forever. - **Kill losers without guilt:** You shipped the "suggest improvements" feature. Nobody uses it. Three people asked for it. Delete it. Get that code out of your codebase. That's a clean win - you learned it wasn't a real need. - **Apply learnings elsewhere:** Enterprise customers were confused by the original onboarding. Are they confused by anything else? Your research agent can scan support tickets for similar patterns. Maybe the pricing page is confusing in the same way. Maybe the payment setup is. You've learned something about how your customers think. Apply it.

Amplification at scale is where your compounding advantage explodes. You're doing more than shipping features. You're building a feedback loop that makes each subsequent iteration better. You learn from one loop and immediately apply it to the next.

A complete loop from sensing to amplifying

Here's how it works in practice. The scenario is painfully real.

Tuesday morning, 8:15am - SENSE

Your monitoring agent alerts you: trial-to-paid conversion for your “Starter” product tier dropped 8% last week. That’s 24 fewer paying customers than expected. At \$29/month, that’s \$700/month in ARR, or about \$8,400 annualized. Not catastrophic, but directional.

Your research agent digs. It finds: - 14 support tickets in the last week, all from trial users in their second week - Common theme: “I don’t understand how to use the custom fields feature” - Your main competitor just released a “quick setup assistant” in a blog post you saw yesterday - Your user data shows trial users hit the custom fields screen on day 3, spend 8 minutes there (usually it’s 2 minutes), and 23% bounce without configuring anything

The pattern is clear: your most powerful feature is a barrier to adoption. New users see it, get intimidated, and bail.

Tuesday, 10:30am - BUILD

You open Claude Code. You describe the problem: “When trial users hit the custom fields screen, they’re getting lost. I want to build an interactive guide that walks them through creating their first custom field. It should be simple, friendly, show them the result immediately, and only ask about the options they actually need.”

You iterate. Back and forth. By 1pm, you have something real: - A step-by-step wizard instead of a form - It asks 3 questions, not 12 - Shows them the result in real-time - Has a “recommended” option they can use without thinking

You test it yourself. You create a test account, go through the wizard, and it works. You feel 60% confident (which is high for a prototype).

Tuesday, 2pm - MEASURE

You deploy the wizard as an A/B test, but only for new trials starting Tuesday afternoon. Your analyst agent sets up the tracking: - % of users who see the custom fields screen - % of users who complete the wizard - % of users who configure at least one custom field - Time spent on the screen - Trial-to-paid conversion rate

The experiment is live. You’re comparing new behavior (wizard) against old behavior (form).

Thursday - WAIT AND OBSERVE

By Thursday, you have 60 trial users in each group. Your analyst agent sends you a daily update:

Control group (old form): - 62% hit custom fields screen - 34% configure something - Average time: 7.2 minutes

Variant group (wizard): - 64% hit custom fields screen - 71% configure something - Average time: 3.1 minutes

The wizard is crushing it. More people complete it. They spend less time. But conversion data takes longer (it's about whether they upgrade, and trials are 14 days).

Next Tuesday - AMPLIFY

You now have 10 days of data. The numbers are even more dramatic:

Conversion rate for users who configured a custom field: - Control: 34% converted to paid - Variant: 52% converted to paid

That's an 18-point difference. In a cohort of 200 trial users, that's 36 additional conversions. At \$29/month, that's \$1,044 in additional monthly revenue. You're paying for your own salary with one feature.

You also notice something interesting: users who went through the wizard and configured a field are staying longer (their retention is higher too).

Now you amplify: 1. **Expand the test** to 100% of trial users (not just a test group anymore) 2. **Brief engineering** on building this in production. Show them the prototype. Show them the data. The wizard moves from prototype to roadmap priority. 3. **Look for similar problems** with your research agent. Where else do new users get stuck and bounce?

From problem detection to validated, profitable solution: **8 days.**

That's the compounding advantage. You didn't wait for a sprint. You didn't spend a week in meetings. You didn't estimate something that might not work. You just: sensed, built, measured, amplified.

How this connects to what you actually care about: outcomes

You probably know about OKRs. Maybe you use them. Maybe you've sat through a planning meeting where everyone wrote objectives and key results. And maybe you've noticed that nothing really changed. You still shipped what you were going to ship. Metrics still moved like they always moved. The OKRs felt like a reporting layer, not a thinking layer.

That's because OKRs without the Impact Loop are just a dashboard. But OKRs plus the Impact Loop? That's a decision system.

Here's how it works:

You set a quarterly outcome, not an output. Outputs are what you make. “Ship role-based permissions.” Outcomes are what happens in the world. “Increase trial-to-paid conversion from 28% to 35%.”

Now your Impact Loop is no longer random. It’s focused. Every sense, build, measure, amplify cycle is aimed at moving that needle. Your sensing agent is looking for signals that point to that outcome. “What’s stopping people from converting?” Your build cycles are experiments on that question. Your measurement is ruthlessly focused on conversion metrics.

A practical example: Let’s say your outcome is “Increase retention from 85% to 89% for enterprise customers by June 30.” That’s a 4-point improvement. On a base of 100 customers, that’s 4 more customers staying every month.

Now you run the Impact Loop against that outcome:

Week 1 - SENSE: Your research agent analyzes all churn conversations for enterprise customers over the last 90 days. The top reasons: (1) they don’t know how to do X with your product, (2) a key person left and nobody knew how to onboard the replacement, (3) they built a workflow that broke when you shipped a change.

Week 1-2 - BUILD: You prototype three experiments: - An in-app tutorial that shows common workflows - An “org admin” role that can manage users and permissions without help - A change log and compatibility guide so users don’t get surprised

Week 2-3 - MEASURE: You roll out all three to customers who churned in the past. You measure whether they come back. You measure whether active enterprise customers engage with each feature.

Week 4 - AMPLIFY: One of the three (let’s say the org admin feature) is showing 2x engagement and higher retention for customers who use it. You expand that test to all enterprise customers. The other two didn’t move the needle, so you kill them.

At the end of the quarter, you’ve run 16+ complete loops, each feeding into the outcome. Some loops moved the needle a quarter-point. Some moved nothing. But collectively, you hit the 4-point improvement. And you know exactly which features, which changes, which customer segments drove that improvement.

Compare that to: “We’ll work on enterprise retention this quarter. Here are the things we might build that seem important.” That’s theater.

The continuous discovery connection

There's something important about how sensing feeds discovery, which feeds building, which feeds more sensing.

Discovery - really understanding what your customers need, not what they say they want - is continuous. It's not a phase. It's not something you do before building. It's parallel to building. The [Continuous Discovery on Autopilot](#) chapter covers the sensing infrastructure in detail.

Here's the loop: 1. You sense a problem (trial users are bouncing on the custom fields screen) 2. You discover the root cause by talking to bounced users, reading support tickets, analyzing behavior (they're intimidated by power) 3. You build a response based on what you learned (a guided wizard) 4. You measure the response (wizard works) 5. You sense new patterns in the data (users who go through the wizard are also more likely to adopt notifications) 6. You discover a new insight (simplification increases feature adoption across the board) 7. You build another experiment (add a wizard to notifications) 8. Loop...

Most companies separate discovery from building. They have researchers and designers and PMs in discovery, then engineers in building. That's a waterfall with research on top. The Impact Loop has discovery embedded in every beat. You sense, you immediately start learning why. You build while learning. You measure to learn more.

This is only possible if building is fast. If prototyping takes three weeks, you can't afford to have discovery in parallel. But if prototyping takes three hours, discovery becomes part of the normal rhythm.

Why this works when sprints don't

Scrum has a specific failure mode in a fast-moving market.

Sprints force batching. You commit to work on Monday. Reality doesn't change until Friday. But reality is changing. Customer behavior is changing. Competitors are moving. Your learning from last week's launch is informing what you should do this week, not what you committed to two weeks ago.

The Impact Loop is continuous. You sense a problem Tuesday morning. You build Tuesday. You measure Wednesday through Friday. You amplify Monday. By the following Tuesday, you're sensing again, and maybe the problem has changed. Or maybe the measurement showed something unexpected. You respond.

This is about responding in the right cadence. Speed for its own sake misses the point. Some problems need a same-day loop. A competitor launches a critical feature, you need to know what

your response is by EOD. Some bets need a multi-week loop. You're testing a major positioning change, you want 4 weeks of data before you decide.

Sprints force everything into the same cadence. That's the death of speed.

Four concrete actions to start this week

You don't need your organization to adopt this. You don't need a change management project or training or buy-in from engineering. You can start running the Impact Loop tomorrow morning. For the prototype step specifically, [Instant Prototyping](#) covers the two-hour build technique in full. For the opportunity layer that feeds the BUILD beat, see [Build Your First Opportunity Solution Tree](#).

Action 1: Set up your sensing layer (Wednesday, 30 minutes)

Stop waiting for weekly metrics reviews. Build a simple daily brief. Use Claude or an AI agent to:

- Pull yesterday's conversion metrics
- Scan your last 20 support tickets and summarize patterns
- Check if any major competitor shipped something
- Identify the top metric mover (what changed the most?)

This should be a 2-minute read every morning. You're not building a complex system. You're just automating the thing you're already doing (checking email, dashboards, Slack) and putting it in one place.

Action 2: Prototype one idea this week (Thursday, 3-4 hours)

Take the problem you've been thinking about and build a prototype. Not a design doc. Something interactive. If you can code (or have Claude Code), build it yourself. If not, work with your designer or engineer to mock something clickable.

Don't overthink it. The goal is to have something you can show a user by Friday.

Action 3: Test it with real customers (Friday, 1 hour)

Send your prototype to three customers. Real customers. The ones you know are feeling the pain. Ask them to use it for 5 minutes. Watch them. Ask one question: "Does this solve the problem I told you about?"

Write down what happens. Did they understand it? Did they get stuck? Did they ask for something different?

Action 4: Measure something (Monday, 2 hours)

Deploy the prototype to a cohort. Even if it's small. 100 users. 500 users. Whatever your traffic looks like. Run it alongside the old version. Measure one outcome: "Does this change user behavior in the

way I hypothesized?”

You’re not measuring perfection. You’re measuring signal. Did the change move the needle? Even a little? That’s all you need.

Then loop. By next Wednesday, you have another sensing moment. What did you learn? What does it suggest you should build next?

Start this week. Start small. One prototype. One test. One loop. Within a month, this will be how you work. And you’ll be shipping features faster than anyone else in your organization.

Because you’re responding now, instead of waiting.

The Eval Is The Spec

Kill the PRD. Ship against a test set. The eval is the contract, the changelog, and the definition of done.

Web version: <https://falkster.com/handbook/the-eval-is-the-spec> · last updated 2026-06-12

The short version

The eval set replaces the PRD as the primary specification artifact for AI features. An eval set is 30 to 200 real input/output pairs that define what good looks like, built from actual production inputs (support tickets, customer messages, usage logs), labeled with the correct output, spiked with 10 to 20 adversarial examples the system should refuse or handle carefully, and scored daily against a rubric (exact match, semantic similarity, LLM-as-judge, or human review, depending on the slice). Engineering builds against the eval set. Done means the score went up, not that someone approved a spec. Feature reviews shrink from 40 minutes of opinions to 20 minutes of score diffs and regression analysis. The first eval set takes an afternoon. By the third one, you have a template and wonder why you were ever writing PRDs.

The PRD was a coping mechanism

The PRD existed because the person who knew what “good” looked like couldn’t test the thing themselves. So they wrote it down, at length, hoping someone else would read it carefully enough to build the right thing. They never did. They built a thing. You reviewed it. You said “not quite,” and the cycle repeated until everyone was tired enough to ship.

That was a reasonable deal in 2018. It’s a bad deal now. In 2026, I can test the thing myself. I can run it against real inputs, score the outputs, see exactly where it breaks, and know within an hour whether we’re close. I don’t need a 10-page spec to adjudicate opinions about what the product should do. I can run the spec.

So I stopped writing PRDs. I write eval sets now. The eval is the spec. It’s also the changelog. It’s also the definition of done. For a practical look at running an eval-first product org at scale, see [Eval-First Product Org](#). For the cost dimension of every prompt change, eval work connects directly to [Gross Margin Is Your Job Now](#).

What an eval set looks like

For any AI surface I'm building, the eval set is 30 to 200 real input/output pairs that define "good." It's the thing I'd use to verify the product works. In 2026 it's also the thing I use to build the product in the first place.

Five moves to build one.

1. Gather real inputs. 30 actual customer messages, requests, transcripts, whatever the feature will handle. Not hypothetical ones. Real. I usually pull these from sales call transcripts, support tickets, or old usage logs. If I don't have real inputs yet, the feature isn't ready for an eval set, and honestly isn't ready for a PRD either.

2. Label the desired output. For each input, I write what "good" looks like. This is the actual PM work. If I can't write it, I don't understand the problem yet. Back to discovery.

3. Include the failure modes. I add 10 to 20 examples specifically designed to trip the system. Adversarial prompts, edge cases, data the system shouldn't know, requests it should refuse. Each one gets a labeled "right" behavior too. These are often more informative than the happy-path examples.

4. Score with a rubric. Decide what I'm measuring per slice. Exact match for structured output. Semantic similarity for generative. LLM-as-judge for judgment calls. Programmatic checks for format. Human review for taste. Different rubric per slice, all in the same file.

5. Run the set every day. Not every sprint. Every day. Every prompt change. Every model swap. Every deploy. Stored in the repo, runs in CI, alerts on regressions.

That's the spec. Engineering builds against it. I don't have opinions about whether the thing is ready. I have a number. The number went up or it didn't.

What changes in the review

A typical feature review before I made this shift: 40-minute meeting, five slides of opinions, a decision everyone would reverse by Friday.

Now: one screen showing eval scores by surface, the diff since last week, the top three regressions with the inputs that caused them, and the three bets we're making to close the gap. 20 minutes. A decision that holds.

The eval is the only thing in the room with standing to end the debate.

The objection, and why I don't buy it anymore

A senior engineer or a founder who wrote a lot of PRDs in the 2010s will tell you: "Evals can't capture everything. There's judgment involved. You can't measure quality."

They're right that evals don't capture everything. They're wrong that this is an argument for the PRD. The PRD captures less. The PRD captures exactly one person's opinion, written once, never re-run, never scored. An eval captures 50 concrete examples, re-run every day, scored against labeled truth. Pretending the PRD was the more rigorous artifact is the sunk cost talking.

For the things evals actually miss (taste, craft, emotional resonance), the answer is human review as a named step in the rubric. You don't throw out the whole approach because one slice needs eyes. You build a rubric with an "eyes" column and keep moving.

A specific example

I was running a feature at Smartcat that needed to decide whether a translation request should route to a human linguist or stay with AI. Classic AI-vs-human decision, high stakes (cost, quality, turnaround).

The old way: write a PRD specifying routing rules, hand to engineering, watch them implement something different from what I meant, iterate for a month.

The new way: I spent an afternoon pulling 150 real translation requests from the last quarter. I labeled each one: should it have gone to AI, to a human, or been flagged for escalation? I wrote out the rubric (correctness of routing, cost efficiency, SLA compliance). Eng took the eval set and had a working router in three days. We ran the eval daily. Every time the score regressed, we could see exactly which inputs were failing and why.

No PRD. No spec review meeting. The eval was the spec. The review was the score.

Get over the "I don't know how"

If you just thought "I don't know how to write an eval set," that's the exact gap this site is trying to close. Claude Code will walk you through it in 20 minutes. The [Instant Prototyping](#) chapter covers the parallel skill of building something testable before writing a spec. The mechanics are: put inputs and expected outputs in a file, write a small script that runs each input through your system, compare outputs to expected, store the score.

That's it. There's a lot of commercial tooling for this (Braintrust, LangSmith, Weave, and others), but you don't need them to start. A CSV and a Python script is a fine v1. The discipline is what matters, not the framework.

The first time you do it you'll feel clumsy. The second time you'll wonder why you were ever writing PRDs. By the third eval set, you'll have a template.

Pick one thing this week

You're probably about to write a PRD. Don't. Pick that same feature and build its eval set instead.

1. Open a file. Call it `evals/feature-name.md`.
2. Write down 30 real inputs the feature will see. Pull from your actual product data.
3. For each one, write what the output should be.
4. Add 10 adversarial cases (edge cases, things the feature should refuse, weird formats).
5. Share the file with your engineer and say: "This is the spec. Build against it."

That's your first eval-as-spec. Notice what changes in the conversation with engineering. Notice how much faster you align on what "done" means. Notice that you already know, before anyone writes code, exactly what the first test will be on launch day.

If you can't write the eval, you don't yet know what you're building. If you can write the eval, you don't need the spec.

Building Evals: Error Analysis, Eval Types, the Loop, the Gate

Teresa Torres's three steps as a weekly routine, plus the fourth step that turns an eval from a scoreboard into what decides what an agent may do alone.

Web version: <https://falkster.com/handbook/building-evals> · last updated 2026-09-19

Most teams build the judge first. They write a rubric from imagination, point a model at it, and get a score that measures nothing anyone cares about. Then they wonder why the number goes up while the customers keep complaining.

Teresa Torres's [hands-on guide to AI evals](#) fixes the order, and this chapter is her three steps written as a routine you can run every week, with one step added at the end. I wrote [why the eval is the spec](#) two waves ago. This is how you build one.

The short version

Four steps. One, error analysis by hand: read fifty outputs, write down the two or three mistakes that recur, rank them by what they cost a customer. Two, match each error to an eval type: code assertion for anything deterministic, golden dataset for one-right-answer questions, LLM-as-a-judge for semantic judgment, customer feedback signals for real usage, always cheapest first as a filter for the expensive one. Three, the loop: fixed inputs, baseline, one change, compare across every eval, log it. Four, the gate: for each action, decide whether it is reversible, and let the eval's pass history buy autonomy only on the reversible side. Correctness is the product team's definition, never the vendor's. Credit for steps one through three goes to Torres. The fourth is mine, and it is the step that makes the first three matter for an agent.

Step one: error analysis, by hand

Set aside two hours. Pull fifty real outputs from the feature or workflow. Not curated, not the best ones. The last fifty.

Read each one and write a note. Not a score, a note: what is wrong with this one, in a sentence. After fifty you will have a list of maybe fifteen distinct complaints, and three of them will have shown up ten times each. Those three are your errors. Everything else is noise for now.

For a production product, the same thing at scale: log every failure a customer reports or a reviewer catches, categorize, and rank by cost. The ranking matters. An error that embarrasses you once a

month is not the same as an error that costs a customer an hour once a day, and the eval budget goes to the second one.

Torres is exact about why this cannot be skipped or delegated. Correctness is context-dependent. The same question gets a different right answer for a student and for a professor. The person doing the error analysis is the person deciding what correct means for this product, and no vendor's benchmark makes that decision for you.

Step two: match the eval to the error

Now, for each of the three errors, pick the cheapest measurement that catches it.

If the check is deterministic, write a code assertion. Does every quoted string appear verbatim in the source? Did a banned phrase appear? Is the output within a length or count range? These run in milliseconds and cost nothing, and they catch a surprising share of what matters.

If the input is small and there is one right answer, build a golden dataset. Thirty to a hundred input and expected-output pairs. Classification, routing, extraction, factual lookups. Score is exact match or close to it.

If the judgment is semantic, and only then, build an LLM-as-a-judge. Write the rubric from the disagreements you had with yourself in step one, not from a blank page. The [rubric answer](#) covers the extraction. Keep a rubric dimension only if failing it maps to a real cost.

And for real usage, instrument the implicit signals: regeneration, editing, follow-up questions, abandonment. These are not a fourth eval you build. They are the observation record the product generates on its own, and they are the only eval that improves without you.

The pattern to steal from Torres's opportunity-tree example: chain them. Code counts the children; the judge is only called when the count exceeds a threshold. Filter with the free thing, spend on what survives. That is also the margin discipline this handbook argues for in [gross margin is your job now](#), because every judge call is a model call.

Step three: the loop

Fix the inputs. Fifty to a hundred cases that look like production, including the ugly ones. Run every eval and record the baseline.

Change one thing. A prompt edit, a model swap, a retrieval change, a reordering of the orchestration. One.

Run every eval again and compare all of them, not the one you meant to move. This is the discipline most teams lose first. A prompt change that fixes hallucinated quotes routinely makes the tone worse, and if the tone eval was not run you find out from a customer.

Log the experiment: what changed, every score before and after, and whether you kept it. The tenth experiment gets compared to the baseline and to the best-so-far, not to last week, and the log is what makes that possible. The starter kit has the template.

Iterate until the error you ranked first in step one is at a level you can defend. Then go back to step one, because the next fifty outputs have a new top three.

Step four: the gate

Torres's loop answers whether a change worked. For an agent that acts, there is a second question, and it decides whether the thing is safe to run: what is the system allowed to do on its own?

For every action the system can take, write down one thing. Can a person undo it cheaply? Draft an email, yes. Update a record with a receipt, yes. Send money, message a customer under the company's name, delete anything, no.

Then the rule. A passing eval on a reversible action buys autonomy: let it run, keep the receipt, watch the observation record. A passing eval on an irreversible action buys a draft and a human, until the eval has a long pass history and a person has confirmed the labels in the golden dataset. And model confidence is never the gate. Confidence is a number the model produces about itself. It says nothing about what happens when the number is wrong. The eval is the evidence input; the reversibility column makes the call. [The harness argument](#) has the full version of this gate.

One warning that belongs here because it breaks the gate silently. A golden dataset labeled by a cheap model, or by the model you are evaluating, measures agreement with itself. A cheap tier passes things a frontier model catches. Record which tier produced every label, human, frontier, or cheap, and put a person on every label that gates an irreversible action. Where the judge and the human disagree, keep the row. It is the most valuable case in the set.

What this looks like as a week

Monday, an hour: read the last fifty outputs, update the error list. Tuesday, two hours: build or extend the eval for the top error, cheapest type first. Wednesday, run the loop on one change. Thursday, read the observation record: what did users regenerate, edit, abandon. Friday, review the gate: did any eval earn enough history to move an action from draft-and-human to autonomous, and did anything get reversed that should move the other way.

Five hours. It replaces the review meeting where six people argue about whether the output looks right.

This chapter sits in the running argument on [AI Product Management](#): deciding what correct means, and what a correct answer is allowed to do, is the part of the job that did not get cheap.

Start this week

Do step one and nothing else. Fifty outputs, a notes column, three recurring errors ranked by cost. Do not build a judge. Do not write a rubric.

If you cannot name the three errors by Friday, you do not know what correct means for your product yet, and no eval will tell you.

Sources: [Teresa Torres, “AI Evals: A Hands-On Guide for Product Teams,” Product Talk \(Sept 2, 2026\)](#) · [AI Eval Starter Kit, falkster toolkit](#)

Ship With Observability or Don't Ship

No feature leaves staging without the traces, metrics, and evals that will tell you whether it's working. Before your first customer hits it.

Web version: <https://falkster.com/handbook/ship-with-observability> · last updated 2026-06-12

The short version

No feature leaves staging without the traces, metrics, and evals that will tell you whether it's working, before your first customer hits it. The instrumentation contract is a one-page document written before any code: success metric (the one number proving the feature works), leading indicators (early signals predicting whether the success metric will land), cost meter (real-time cost per successful action), eval set (named set with a pass threshold), trace points (which actions get logged), dashboard URL (exists before launch even if empty), and kill condition (metric below threshold for time period means the feature is reviewed for deprecation). If any of the seven is missing, the feature is code complete, not done. Observability and evals are the same loop from two angles. Watch both on the same page. A feature without observability is a feature you shipped on faith.

The gap where PMs fall off

The space between your prototype and your production is where most PMs fall off the Builder path.

You ship a beautiful prototype. Hand it to engineering. They wire it up. It goes live. A week later you ask “how's it doing” and get a shrug. Nobody set up instrumentation. Nobody knows the adoption curve. Nobody can tell you what's broken because nothing's being measured. By the time dashboards are stitched together, it's three weeks post-launch, half the customers who tried it have already bounced, and the learning window is closed.

I've watched this happen at every company I've worked at. It's preventable. It's also non-negotiable.

The rule: no feature leaves staging without the traces, metrics, and evals that will tell you whether it's working, before your first customer hits it.

Why “we'll add metrics later” keeps failing

Three predictable failures, in order of severity.

First, you can't tell if it's working. Launch. Some customers use it. Some don't. Some use it once and never again. Without instrumentation, no idea which is which. You guess. You guess wrong. You invest in the wrong things or kill the feature based on vibes. The feedback loop closes weeks later, when sales escalates a complaint or finance flags a cost spike.

Second, regressions go undetected. Without an eval set running against production, the silent regression (the one that happens when a model updates, a prompt drifts, a downstream tool changes) finds your customer before it finds you. The other half of shipping is telling people what changed, which is the job of [the release documentation agent](#).

Third, you can't justify the next investment. Stakeholder asks "should we double down on this?" No data. Takes three weeks to instrument what you should have instrumented at launch. The moment has passed.

"Adding metrics later" isn't a slight delay. It's a multi-week feedback loop being absent during the most learnable period of the feature's life. A strategic loss disguised as tactical convenience.

The instrumentation contract

Before any code is written, PM and engineer write a one-page contract.

- 1. The success metric.** What's the one number that tells us this feature works? Not engagement. Not click-through. The actual outcome. Specific. Quantifiable. Tied to the user job.
- 2. The leading indicators.** What earlier signals tell us if the success metric will land? Adoption rate in week 1. Time-to-first-success. Percent of users who reach the "aha" moment. Leading indicators are what I watch in the first two weeks, before the lagging metric is reliable.
- 3. The cost meter.** Cost per successful action for this feature. Real-time. Per user, per workspace, per surface. If this number goes wrong, the rest of the metrics don't matter. Feature is unprofitable and either gets repriced or killed.
- 4. The eval set.** Named eval set for this feature, current score, threshold below which the feature shouldn't ship. Runs daily against production once live.
- 5. The trace points.** What user actions, system calls, model calls, tool invocations get traced? Sample rate? Retention period? I don't need to instrument everything. I need to instrument the path that matters. I write down which path that is.
- 6. The dashboard URL.** Where will all of this be visible, in one place, refreshed automatically? Exists before launch, even if the data is empty.

7. The kill condition. If metric X stays below threshold Y for time Z, feature is reviewed for deprecation. Written down before launch. Makes the future deprecation conversation 10x easier (see [The Deprecation Playbook](#)).

One page. Negotiated up front. Engineering builds against it. PM signs off. Nothing ships without all seven.

The Definition of Done upgrade

Most teams' Definition of Done includes "QA passed," "code reviewed," "docs updated." Add four lines:

- Instrumentation contract written and approved.
- Eval set in place, current score above threshold.
- Dashboard live, populated with real data from a staging cohort.
- Cost-per-action measurement validated end-to-end.

If any of these four is missing, the feature is not done. It is "code complete." Different state. Different decision.

This is the move that hardens the new practice into team culture. It's not a PM preference. It's a checklist item in the merge process.

The "good enough" rule on instrumentation

The trap on the other side is over-instrumentation. PMs new to this discipline ask for everything to be measured, everywhere, all the time. Result: a dashboard nobody can read, a data pipeline that costs more than the feature, and instrumentation work that delays launch by weeks.

The right amount is the minimum that lets you make the next decision. That's it. You can always add more later if a question emerges the current data can't answer. You can rarely walk back the cost of over-instrumentation.

A useful test: imagine looking at the dashboard a week post-launch. What's the first decision you'll need to make? Instrument exactly enough to support that decision. Cut everything else. You'll be surprised how spare that is.

What the PM actually owns here

The PM doesn't do the instrumentation work. The PM owns what gets instrumented and why. The engineer implements.

PM job is: - Writing the contract before work starts. - Pushing back when “we’ll add it later” appears in the plan. - Owning the dashboard after launch. - Making the kill-or-double-down call when the data points clearly.

That last one is where it matters most. Most PMs have an instinct to keep features alive past their data-supported lifespan because killing them is awkward. The contract removes the awkwardness. The kill condition was agreed up front. You’re not killing a baby. You’re executing the plan.

Observability plus evals: the same loop

Observability and evals are the same loop from two angles. Evals tell you what the system is *doing*. Observability tells you what the users are *experiencing*. Both must run continuously. Both must be visible on the same page.

When the offline eval score is high but production usage is flat, you have a discovery problem (people aren’t finding or trying the feature). When the eval score is low but usage is high, you have a quality problem (people are using it but it’s failing them). When both are good, you have a winner. When both are bad, you have a learning.

Most teams in 2026 only watch one or the other. The builder PM watches both. The pair is much more informative than either alone.

Pick one thing this week

You’re about to ship something. Write the contract. For how the contract fits into the broader Build phase alongside prototyping, see [Prototype Before You Spec](#) and [The Impact Loop](#).

1. Open a page. Title it “[Feature Name] Instrumentation Contract.”
2. Write the seven items: success metric, leading indicators, cost meter, eval set, trace points, dashboard URL, kill condition.
3. Share with your engineer and your data person. Ask them to push back on any item that’s vague.
4. Once signed, commit it to the feature’s ticket in Jira or Linear. Make it part of the “definition of done” checklist.
5. Don’t ship the feature until all seven are in place.

Do this for one feature this week. Do it for every feature next month. Within a quarter, no feature ships on your team without observability, and you stop losing the learning window on every launch.

A feature without observability is a feature you’ve shipped on faith. Faith is not a strategy.

The Deprecation Playbook

Feature death is the most under-written topic in PM. Kill on signal, not politics, and your team ships faster than the team that hopes politely.

Web version: <https://falkster.com/handbook/the-deprecation-playbook> · last updated 2026-06-12

The short version

Deprecate on signal, not politics. Every feature ships with an explicit kill condition written at launch: if weekly active usage stays below 2% of MAU for 8 weeks, the feature is reviewed for deprecation. That condition turns deprecation from a political debate into the execution of a decision already made. The three tiers: soft sunset (hidden from defaults but still accessible), hard sunset with migration (feature removed by a date with proactive customer comms), and immediate kill (broken or dangerous features only). The killer move on customer comms is individual notice to the specific users who used the feature in the last 90 days. And the cultural move that makes it all work: publicly celebrate what gets killed. A page showing features killed in the last 12 months, what the data showed, and what you learned reframes deprecation from failure to discipline.

The topic every PM book skipped

Every PM book in the 2010s spent 200 pages on how to launch and 0 pages on how to retire. Result: a generation of products bloated with features that never worked, never reached scale, and never got removed because removal required someone to admit the launch failed.

I've lived inside these products. At Adobe, at Salesforce, at three startups. Everyone on the team knew which features were dead. Nobody would say it. So the bloat compounded. The [Anti-Backlog](#) chapter covers the upstream discipline of not adding features that will need killing later.

In 2026 the cost of carrying a dead feature has gone up. Each surface has its own model costs, its own eval maintenance, its own support burden, its own attack surface, its own cognitive load on every customer who sees a menu item that doesn't work the way they hoped. Carrying ten dead features in an AI product is materially expensive in a way that carrying ten dead features in a CRUD app was not.

So the playbook for killing features has to get better. This is it.

The thesis

Deprecate on signal, not politics. A feature whose data clears the kill threshold gets deprecated, with the PM who launched it as the executor. The decision is not a debate. The debate happened when the kill condition was written into the launch contract. Deprecation is the execution of a decision that was already made.

What's broken about how features die today

The default is “let it linger.” Nobody actively chooses to keep a dead feature. Nobody actively chooses to kill it. It stays. Engineering complains occasionally that they have to maintain it. Customers complain occasionally that it doesn't work right. The PM nods and moves on. After a year, it's part of the furniture.

Killing is socially expensive. The PM who launched it is still in the role, often with a promotion built on the launch. Killing the feature feels like criticizing them. So the conversation gets avoided, and the longer it's avoided, the more entrenched the feature becomes.

There are no triggers. No specific metric, no specific timeframe, no specific decision date forces the conversation. So the conversation never happens. Inertia wins.

And customer comms feel scary. Some unknown number of customers might use it. Killing it might create a churn event. The conservative move is to do nothing. But nothing has its own cost, invisible until it's huge.

The kill condition, set at launch

Every feature ships with an explicit kill condition. Written in the instrumentation contract (see [Ship With Observability](#)). Stated as: *if metric X stays below threshold Y for duration Z, the feature is reviewed for deprecation.*

Examples:

- “If weekly active usage stays below 2 percent of MAU for 8 weeks, the feature is reviewed for deprecation.”
- “If cost per successful action stays above \$0.50 for 4 weeks without a viable optimization path, the feature is reviewed for deprecation.”
- “If the eval score regresses below 0.7 and can't be recovered in 2 weeks, the feature is reviewed for deprecation.”

“Reviewed for deprecation” is not “automatically killed.” It's a forced conversation at a specific moment, with the data already on the table. The decision can still be: invest more, reposition, limit, or kill. The point is that *the decision happens* instead of being avoided.

Without a written kill condition, no feature crosses an objective threshold for the deprecation conversation. With one, every feature has its own clock.

The three deprecation tiers

Not every kill is equal.

Tier 1: Soft sunset. Feature stays accessible to existing users but is removed from default UI, docs, and onboarding. New users don't discover it. Existing users keep using it. No customer harm. Often right for features with a small but loyal user base. Engineering maintains it but stops investing.

Tier 2: Hard sunset with migration. Feature is being removed by date X. Active users get N weeks of notice, a migration path, and proactive support. Used when the feature is incompatible with where the product is going. High-touch comms. Rare but sometimes correct.

Tier 3: Immediate kill. Used only when the feature is broken, dangerous, or unprofitable to a degree that makes further runtime irresponsible. Comms is short and direct: "This feature is being removed because [specific reason]. Here's what to use instead." Burn the bandage in one move. Painful for two days. Better than three months of slow death.

The PM picks the tier based on usage, customer impact, and strategic reason. Tier 1 is default. Tier 3 is rare. Tier 2 is the one most teams under-use because it requires real comms work, but it's often right for retiring meaningful surfaces gracefully.

Customer comms that actually work

The killer move: individual notice to active users, not a generic email blast.

Pull the list of users who used the feature in the last 90 days. For each, send a personal-feeling note (templated but customized): "We're sunseting [feature] on [date]. We saw you used it [N] times in the last quarter. Here's what we recommend instead, and here's how to get help if you need it."

This costs 30 minutes of agent setup and converts more goodwill than the launch announcement did. It also forces you to confront the actual scale of use. Sometimes you'll find the "tiny" feature has 200 paying customers, which changes the kill decision. Better to find out before the announcement than after.

Internal comms that make the culture easier

The single move that makes deprecation culturally easier: publicly tracking what gets killed.

A page in your team space, updated continuously: "Features killed in the last 12 months." For each: what it was, why, what the data showed, what we learned. Make it celebratory rather than apologetic.

Reframe deprecation as evidence the team is operating with discipline, not failure.

When the next feature crosses its kill threshold, the conversation isn't "do we kill this?" It's "how do we add this to the killed-features page?" Frame change does most of the work.

The mortality dashboard (the spicier version)

For the truly committed: a public-internally page where every feature has a mortality risk score visible to the whole company.

The score is a function of: weeks since kill condition was last evaluated, distance from threshold, recent trend, cost per action, support ticket volume. Updated daily. Visible to engineering, support, sales, and the C-suite.

This will be deeply uncomfortable for people who launched the high-mortality features. That's the point. Discomfort is the friction that makes deprecation inevitable. Without it, hope masquerades as strategy and dead features stay in the product forever.

A more diplomatic version: keep the dashboard internal to product. A more cowardly version: private to the PM. Pick the level your culture can handle, but don't skip it entirely.

Pick one thing this week

Pick the feature you've been quietly hoping would die. Write its kill condition now, in retrospect, and see where it lands.

1. Open the feature's page.
2. Write: "If weekly active usage stays below X for Y weeks, this feature will be reviewed for deprecation." Pick numbers you can defend.
3. Pull the actual data for the last 90 days. Does the feature already meet the kill condition?
4. If yes, schedule the deprecation conversation. If no, commit to the kill condition going forward and put a reminder on your calendar for the review date.
5. Tell your manager you did this.

Do it for three features this quarter. By the end of the year, your team has killed more dead code than it shipped in the prior year, and your product is meaningfully less bloated.

Kill the feature on signal. The team that kills bravely outships the team that hopes politely, every time. The [Impact Loop](#) gives you the outcome metrics that determine whether a kill condition has been met.

Incident Response Is a PM Ritual

An incident is a customer telling you the truth about your product, loudly, all at once. Stop letting engineering listen alone.

Web version: <https://falkster.com/handbook/incident-response> · last updated 2026-06-12

The cheapest discovery your company already does

Most PMs treat incidents as engineering's problem. They show up only when sales escalates, only to perform concern. They don't read the post-mortem. They don't extract a product lesson. They go back to the roadmap meeting and the cycle repeats.

This is the most expensive habit I see in PM practice. Every incident is a bundle of compressed truth: a latent assumption that was wrong, a missing guardrail, a mis-scoped user segment, a feature that worked in the demo but not in the wild, an integration nobody mapped. The infra fix is the engineer's job. The product lesson is the PM's job.

Almost nobody does the second job. That's where this chapter starts.

The short version

Incidents are the cheapest discovery your company already does. Every Sev-2 or worse exposes a latent product assumption that was wrong, a missing eval, and a mis-scoped workflow. The engineering post-mortem covers the infra fix; the PM post-mortem covers the product lesson and they are different documents that both need to happen. The PM-owned post-mortem has six sections: user-facing description, wrong assumption, eval gap, what customers said during the incident, product-level fix, and updated eval set. It is written within 48 hours, shared at the weekly team review, and the eval update is committed alongside it. Customer comms during the live incident are also the PM's job: acknowledge fast, estimate honestly, send the post-incident note explaining what changed. After three months of this practice, incident patterns become the leading indicator of product health that NPS misses by quarters.

What changes when the PM owns the post-mortem

Engineering owns the engineering post-mortem. PM owns the product post-mortem. They're different documents. They cover different ground. They both happen, in parallel, after every Sev-2 or worse.

The PM-owned post-mortem is a one-page doc, written within 48 hours of incident close. Six sections.

1. What broke from the user’s perspective. Not “service X returned 503s.” User-facing description: “users in workflow Y could not complete step Z; they saw [specific behavior]; they tried [specific workarounds].”

2. The product assumption that was wrong. Every incident exposes one. “We assumed users would only upload documents under 10MB.” “We assumed the model would handle ambiguous tool calls gracefully.” “We assumed cost per action wouldn’t spike when usage doubled.” Name the assumption.

3. The eval gap. What test in the eval set should have caught this and didn’t? Either the eval set was missing this case, or the rubric didn’t penalize this behavior, or the eval was correct and production diverged from it. Name which.

4. What customers told us during the incident. Direct quotes. Patterns from the support flood. The 5 things customers said that we wouldn’t have heard otherwise. This is the most valuable section. Most teams don’t write it because they’re too busy fighting the fire to listen.

5. The product-level fix. Separate from the infra fix. Often involves changing default behavior, adding a fallback experience, reshaping a workflow, tightening a guardrail, removing a foot-gun feature, or adjusting pricing for a high-cost flow that’s now exposed.

6. The updated eval set. The eval set is updated to include the case that was missed. This change is committed alongside the post-mortem. Next time this pattern emerges, it’s caught in CI before reaching production.

That’s the document. Read aloud at the team’s weekly review. Action items have owners and dates. Six weeks later, the team verifies the fixes landed.

Customer comms during the live incident

The PM owns the customer comms during a live incident. Not marketing alone. Not support alone. The PM, because the customer comms shape the experience of the incident more than the technical resolution does.

Three rules.

Acknowledge fast, even if you don’t know the cause. Silence reads as ignorance or indifference. A short “we’re seeing X, we’re investigating, we’ll update in 30 minutes” is the move.

Estimate honestly. Bad estimates erode trust faster than the incident does. “We don’t know yet” is better than “should be fixed in 15 minutes” repeated for two hours.

Tell them what you learned, after. The post-incident note explaining what happened, what you’re changing, what you’ll do differently next time, is the single most trust-building artifact a product team produces. Most teams skip it. Send it.

This is also the moment to capture which customers were materially affected and follow up individually. The customer who reports a bug during an incident and gets a personal note a week later, explaining what was fixed and why, is the customer who upgrades next quarter.

Incident frequency as product signal

After three months of running PM-owned post-mortems, you’ll have a pattern visible in the incident log. Patterns to watch:

Same surface, repeated. Same feature, same workflow, keeps producing incidents. Infra fixes have layered up. Underlying product design is wrong. Stop fixing the symptom. Redesign the surface.

Same root assumption, different surfaces. “We assumed users would tolerate latency over 5 seconds.” This was wrong in feature A in March, in feature B in May, in feature C in July. The assumption is wrong globally, not locally. Change the global default.

Cost-driven incidents. A growing share of incidents are about cost spikes: runaway usage on a flow nobody priced for, infinite loops in agent calls, retry storms after a minor outage. These are not bugs. They’re unit-economics design flaws. Address them as pricing or architecture issues.

Customer-facing escalations dominating. If a growing share of incidents reach customer-visible status before alerts catch them, your observability is weaker than your customers’ patience. Invest in detection, not in PR.

These patterns are leading indicators of product health. NPS lags by quarters. Incident patterns lead by weeks. The PM who watches the incident log carefully knows what’s about to break in a quarter the way a sailor knows what the weather will be in six hours.

Incidents I’ve actually learned the most from

The Sev-1 I learned the most from at Smartcat involved a translation flow that hit a model timeout under specific document structures, returned an empty result, and the workflow accepted the empty result as valid output. Engineering’s fix was a retry plus a non-empty validator.

My product post-mortem found three things engineering’s didn’t:

1. We had assumed workflow steps would either succeed or fail loudly. They could also succeed silently with empty output. This assumption was wrong across at least four other surfaces.
2. Customers had been working around the silent-empty failure for weeks before the incident, never reporting it because they didn't realize it was a bug. Support had 23 tickets about "translations that look empty," all marked "user error."
3. Our eval set tested for translation correctness but not for non-empty output. Trivial gap. Easy fix.

Three product changes from one incident. Engineering's post-mortem was about the timeout. Mine was about the system that hid the problem from us. Different documents. Both needed.

Pick one thing this week

You probably have an incident in the last 30 days where you didn't write a product post-mortem. Write it now.

1. Pick the most recent Sev-2-or-worse incident.
2. Open a doc. Write the six sections (user-facing description, wrong assumption, eval gap, what customers said, product fix, updated eval).
3. Be honest. The whole point is the assumption that was wrong, not the heroic engineering response.
4. Share with your team. Schedule a 20-minute walkthrough at the next weekly review.
5. Add at least one new test case to the eval set based on what you found.

Do this for every incident going forward. Within a quarter you have a library of compressed truths about your product that you can mine for the next several roadmap decisions. An incident is a customer telling you the truth about your product, loudly, all at once. Don't let engineering listen alone.

The eval set you build here feeds directly into [The Eval Is the Spec](#), the chapter on treating evaluations as the primary spec artifact. For the agent that monitors production quality and surfaces drift before an incident fires, see [the Red Flag Detection Agent](#). For the continuous listening system that catches pre-incident signals, see [Continuous Listening](#).

Agent-to-Agent Dispatch

The listening agent does not file a ticket. It hands the outcome to a prototyping agent, and a working prototype is waiting before standup.

Web version: <https://falkster.com/handbook/agent-to-agent-dispatch> · last updated 2026-07-22

The short version

Agent-to-agent dispatch is what happens when you stop putting a ticket between the customer signal and the build. A listening agent sits on your calls, tickets, and churn surveys and extracts the outcome a customer is reaching for. Instead of writing that up for a human to triage, it hands the brief straight to a prototyping agent, which builds a working prototype the same day. By the time you read the morning digest, the prototype is already attached. You review the extraction and the prototype, then decide: show it to the customer, iterate, or kill it. No backlog, no planning meeting, no handoff. The PM stops initiating work and starts editing it. The control that keeps this sane is not the ticket queue, it is the eval bar: most dispatched prototypes die, and the survivors graduate to engineering hardening.

The relay you didn't notice you were running

Walk the path a customer signal takes in most companies. A customer says something on a call. Someone clips it into a note. The note becomes a Jira ticket. The ticket sits in a backlog. Weeks later it gets groomed, estimated, prioritized against forty other tickets, and maybe scheduled. Then someone builds a first version, and only then does a customer see anything.

Every arrow in that chain is a human relay, and every relay adds latency and loses signal. By the time anyone builds, the outcome the customer actually wanted has been flattened into the feature someone wrote on the ticket. You optimized the queue and lost the point.

The [anti-backlog](#) already argues for burning the queue. Dispatch is what you run in its place. The relay collapses because the agents talk to each other.

How the handoff works

Two agents, one contract between them.

The first is the listening agent from [Continuous Listening](#). It doesn't log feature requests. It extracts the outcome underneath them. A customer asking for a "CSV export button" is handing you a solution guess. The outcome is "get this data into my board deck without re-keying it." The listening

agent records the outcome, tagged with the customer, the evidence, and how many others are reaching for the same thing.

The second is a prototyping agent. It takes the outcome brief and does what a builder PM would do in the [two-hour prototype method](#): frames the problem, builds the core interaction, and deploys something a customer could actually click. It does not wait for you to ask.

The contract between them is the outcome, not the feature. That single rule is what makes the handoff worth automating. Feature requests fragment into a hundred one-off asks that no agent can build coherently. Outcomes collapse into a handful of jobs, and a job is something a prototype can go after in a dozen different ways.

What arrives on your desk

You do not wake up to a to-do list. You wake up to drafts.

The digest says: three enterprise customers are reaching for the same outcome, here is the evidence, and here is a working prototype that takes a run at it. Your job is to review, not to originate. Did the listening agent extract the real outcome, or did it capture the surface request? Does the prototype serve the outcome, or does it just implement the button somebody named? That review is the whole job now, and it is the same skill described in [PM as Editor](#): reading the output the way a senior PM reads a team member's work, cutting what does not serve the purpose, and shipping the version that does.

Most mornings, the right call on a given prototype is to kill it. That is not a failure of the system. That is the system working. You spent an agent's time, not a sprint.

The eval bar is the control, not the queue

The obvious objection is that skipping the backlog invites chaos. It would, if the backlog were the thing keeping you honest. It never was. A four-hundred-item backlog is not control, it is institutional amnesia in tabular format.

The real control is the eval. A dispatched prototype does not ship because it looks good in a meeting. It graduates because it clears an eval bar that defines what good looks like, the same contract described in [The Eval Is The Spec](#). Most prototypes never clear it, and that is the point. The eval decides what earns scarce engineering time, and the survivors get handed to engineers to harden, per the ledger in [Old PM vs Product Builder](#). Dispatch makes the front of the funnel fast. The eval keeps the back of the funnel disciplined.

You cannot run this on bare ground

Dispatch is a loop, and a loop that writes and deploys code needs somewhere safe to run. An agent building unsupervised against production is not speed, it is a liability. What makes dispatch safe is the substrate underneath it: scaffolded environments an agent can build in, guardrails that bound what it can touch, an eval harness that scores the output, and isolated deploys so a prototype reaches a customer without ever touching the real system. Building that substrate is engineering's job in an AI-native org, and it is the precondition for everything on this page.

Start this week

You do not need the full loop to feel the shift. Wire the smallest version.

Take one listening agent you already trust and one outcome it reliably surfaces. When it fires, have it trigger a single prototyping run automatically, into a sandbox, and drop the result into your morning review next to the signal. Do not wire it to production. Do not let it ship anything. Just let one agent hand one brief to another, and put the result in front of your own eyes before your first meeting.

The first time a working prototype is waiting for you that you never asked anyone to build, you will understand why the ticket was the bottleneck all along.

Part 4: AI Craft

The technical discipline the role now demands.

PM AI Agent Fleet, Mapped to the 7-Stage Operating System

A live index of every AI agent for product managers, mapped to the 7 stages of the PM Operating System: Sense, Discover, Decide, Build, Ship, Measure, Amplify.

Web version: <https://falkster.com/handbook/ai-agent-army> · last updated 2026-06-12

The short version

The PM AI agent fleet is a set of autonomous agents that cover every repeated decision and output a product manager makes, mapped to the seven stages of the PM Operating System: Sense, Discover, Decide, Build, Ship, Measure, Amplify. Each agent runs on a schedule, pulls from connected data sources via [Model Context Protocol \(MCP\)](#), and delivers reports where you already work. Setup takes about an hour. Deploy them incrementally starting with Red Flag Detection. By month two the whole fleet is running. This page is the live index, auto-generated from the blog, and it updates every time I publish a new agent post.

For the honest meta-look at the fleet, what stuck, what died, what I'd do differently, see [39 PM AI Agents Deployed: What Stuck, What Died, and Why](#), the field-research piece behind this index.

Why a full fleet, and why full automation is a PM superpower

I started with 18 agents. They saved me 10+ hours a week. Then I realized the architecture was incomplete.

The agents were only covering parts of the PM job. I had good visibility into what was broken (Sense), but no agents orchestrating research into customer insights (Discover). I could spot priorities (Decide) but couldn't automatically generate PRDs or coordinate releases. The infrastructure was there, MCP wired up, vector DB running, but the playbook was thin.

So I mapped the entire PM Operating System across 7 stages and kept adding agents until each major decision or output a PM makes repeatedly had one. The result: I'm not doing PM work anymore. I'm doing PM *thinking*. The data flows, analysis happens, decisions get shaped, and I focus on judgment, strategy, and leadership.

This is what full PM automation looks like.

The data foundation

Your agent stack sits on three layers:

Structured data via MCP (Model Context Protocol) Connected data sources are everything. Agents pull from [Jira](#), [Slack](#), [Zendesk](#), [Salesforce](#), [GitHub](#), Google Calendar, [Notion](#), Google Drive, and analytics ([Amplitude](#), [Mixpanel](#)). MCP standardizes the access, write once, agents read across all systems.

Unstructured data via vector database (Weaviate) Call transcripts from [Gong](#). Support ticket descriptions. Meeting notes. Emails. These live in a vector database so agents can semantically search, finding patterns that share meaning, not just keywords.

Delivery and scheduling Agents output via Slack webhooks (reports land where you check them). They run on cron schedules (daily, weekly, bi-weekly). Automated but you control the cadence.

To get started: - [Claude setup guide](#), MCP connections for Claude Desktop, Claude Code, or Claude Projects - [OpenClaw setup guide](#), self-hosted alternative with the same data sources

Setup takes one hour. Then you have the foundation.

The 7-stage PM Operating System

Every agent I've written about slots into one of seven stages. The list below is generated from the blog at build time, so when I publish a new agent - *.mdx post and tag it with a stage, it appears here automatically. No article edit required.

Try the live sandbox first. [Open the Agent Sandbox \(PM Version\)](#) to poke at an interactive version of the fleet before reading every blueprint. You can click through agents, see sample outputs, and feel the cadence in about five minutes.

The live agent fleet, grouped by stage with every blueprint linked, is at <https://falkster.com/handbook/ai-agent-army>. It updates as agents ship, so it is not reproduced here.

Your daily and weekly rhythm

Across the fleet, agents fire on a staggered schedule so information arrives when it's useful, not when it's fresh off the keyboard.

7:00 AM (daily, weekdays) Daily Focus, PM Issues, Documentation Gaps, GTM Monitoring fire. You know your priorities, what's broken, what's missing, and what's happening in market before you

open email.

8:00 AM (daily, weekdays) Support Ticket Signals and NPS/CSAT Analysis run. Signal detection deepens, customer sentiment and issue patterns become visible.

Monday, 8:00 AM Weekly Ops Digest, Engineering Capacity, Customer Segmentation, and Sprint Planning kick off. This is your week-setup moment, patterns from last week, capacity reality, customer shifts, and your sprint crystallizes.

9:00 AM (daily, weekdays) Red Flag Detection, Product Ops, Roadmap Tracker, Team Triage, and Signal-to-Ship Cycle Time fire. Second wave of daily signals, catches what slipped through morning scan, and refreshes the in-flight portfolio view.

Monday, 9:00 AM Signal-to-Ship Cycle Time digest lands. The weekly meta-view of the whole fleet's output: where time is going across the seven stages (Signal, Prototype, Design partner, Production code, Rollout, Launch, Measure), which stage is the bottleneck, and how cycle time is trending. The agent that tells you whether the PM transformation is real.

Monday, 1:00 PM Executive Report compiles after morning agents finish. Prep for leadership conversations.

Monday, 4:00 PM Opportunity Prioritization and Assumption Tracker run. Your week's prioritized bets and validation plan are ready.

Tuesday, 9:00 AM Product Dashboard updates metric views before any stakeholder conversations.

Wednesday, 7:00 AM Interview Synthesis processes this week's customer calls into structured insights.

Wednesday, 8:00 AM Release Readiness Review and Release Documentation assess what's ready to ship.

Thursday, 10:00 AM Release Checker validates post-release stability.

Thursday, 2:00 PM Customer Journey Mapping updates your understanding of where friction lives.

Friday, 10:00 AM OKR Tracker and Retrospective Synthesis wrap the week. Did you hit outcomes? What did you learn?

Friday, 2:00 PM Opportunity Prioritization finalizes next week's top bets. Stakeholder Communication generates tailored updates for every team.

4:00 PM (daily, weekdays) Product Health and Feature Adoption run as your closing scan. OKR Tracker checks progress.

Hourly (continuous) KPI Watchdog watches your core metrics and pages you the moment a KPI drops outside its band, with a prototype fix attached.

Bi-weekly (rotating) Competitive Intelligence, Market Intelligence, Customer Commitments, Journey Mapping, Win/Loss Analysis cycle through. Strategic slower-burn agents that feed your monthly planning.

Building this yourself

The full agent fleet runs on: - **Claude (Desktop, Code, or Projects)** with **Model Context Protocol (MCP)** connected to your data sources - **Weaviate vector database** for semantic search across unstructured data - **Slack webhooks** for report delivery - **Cron scheduling** for agent execution

Setup guides walk you through all of it: Jira, Slack, Zendesk, Salesforce, GitHub, Google Calendar, Gong, analytics, PagerDuty, Sentry connections. One hour to wire. Then deploy agents incrementally.

Don't try to run them all at once. Start with SENSE (Red Flag Detection, Support Signals, NPS Analysis). Then add DISCOVER (Weekly Ops Digest, Interview Synthesis). Build from there.

By month two, your entire PM function is instrumented. By month three, you're operating at a completely different level.

What's still human

Agents handle data gathering, analysis, synthesis, and pattern detection. But PM judgment still lives here:

- **Customer empathy.** An agent tells you 14 customers mentioned a pain. Only you sit across from them and feel the real problem.
- **Strategic trade-offs.** The agent can prioritize by impact and effort. You decide whether market timing overrides the math.
- **Organizational persuasion.** Your agent can't convince your CEO that the roadmap shifted. You can.
- **Hypothesis design.** The agent flags assumptions. You decide which ones matter and how to test them.
- **Creative problem-solving.** The agent finds the constraint. You figure out how to design around it.

This is the PM job as it should always have been. Data gathering automated. Analysis automated. Thinking and judgment remain yours.

The honest limitations

They hallucinate. Claude is confident when it's wrong. Every output needs your eye. Always verify critical findings.

They need real data. A DISCOVER agent is useless if your Gong transcripts aren't in the vector DB. Garbage in, garbage out.

They need tuning. Deploy an agent with bad thresholds and it alerts you 20 times a day. Best agents are the ones you've trained over 2-3 weeks.

They miss context. An agent can tell you which customers churned. It can't explain the political situation that drove the churn.

They need management. Set up the foundation once. But manage the agents like a small team, review outputs weekly, adjust parameters, add new data sources, prune noise.

Start this week

Don't read about agents. Build them.

Today: - [Claude setup guide](#), wire MCP connections - Start with Jira + Slack + Zendesk. One hour.

Tomorrow: Deploy Red Flag Detection and Daily Focus. These two reclaim 60+ minutes per day.

This week: Add Product Health and Support Signals. You now have basic signal detection.

Next week: Add Weekly Ops Digest, Interview Synthesis, and Market Intelligence. DISCOVER stage activates.

Week 3-4: Opportunity Prioritization, Assumption Tracker, Sprint Planning. DECIDE stage runs.

Month 2: Full fleet deployed and running. Your entire PM Operating System. You're operating at a different level entirely.

Your team will notice. Your manager will notice. Your customers will notice. That's the real point.

When Not to Use AI

The senior PM move in 2026 isn't using AI everywhere. It's knowing when a regex, a query, or a form beats a model.

Web version: <https://falkster.com/handbook/when-not-to-use-ai> · last updated 2026-06-12

The instinct everyone has right now

In 2026, every PM's instinct is: throw an LLM at it. Mine was too for about a year. I watched us wrap regexes in chatbots, turn dropdowns into “chat experiences,” replace working SQL queries with RAG pipelines, and generally add a model call to every workflow we touched.

Most of it was worse than what it replaced. Slower. More expensive. Less predictable. Users had to type out what a dropdown would have collected in one click. The model had to infer what the form would have constrained. Both sides did more work for a worse outcome.

The senior move in 2026 isn't adding AI everywhere. It's knowing when to take AI out of a flow. That's the skill that separates product builders from hype builders.

The short version

Before wrapping any surface in an LLM, walk a five-step decision tree in order and stop at the first yes: can a rule do it, can a query do it, can a form do it, can a heuristic and lookup do it, and only then use a model. The most powerful pattern in 2026 is the hybrid: use a rule for 80% of cases, escalate to a model for the 20% that actually need it. That hybrid cuts costs in half with no quality drop. The cost math is real: a 1,500-token prompt plus 500-token completion at 100 actions per user per day across 50,000 users is 1.5 million dollars a month for a feature a regex could do for free. The PM who can name why a model is necessary (rather than just convenient) is the one who earns credibility when the model genuinely matters.

The decision tree I run now

Before wrapping any surface in an LLM, I walk through these questions in order. I stop at the first “yes.”

1. Can a rule do it?

If the input space is small or the output is well-defined (validation, parsing, format conversion, classification with under ten categories), a rule or regex or state machine will outperform a model on

cost, latency, and reliability. I don't use a model to check if an email is valid. I don't use a model to extract a phone number. I don't use a model to route between three workflows when an if/else does it cleanly.

2. Can a query do it?

If the answer lives in the database, I query for it. RAG is a query with a creative writing flourish on top. When a user asks "what's my MRR last month," that's a SQL query rendered as a number, not a 200K-context-window model call. Give them a chart.

3. Can a form do it?

This one breaks most PMs' brains. Many "chat experiences" shipping in 2026 are worse than the form they replaced. Chat is the right interface when the user is exploring, the inputs are open-ended, and the value comes from synthesis. Form is the right interface when the user knows what they want, the inputs are enumerable, and the cost of error is non-zero.

Most agentic SaaS products are forms in chatbot clothing. Take the chatbot off. Ship the form. Ship faster.

4. Can a heuristic and a lookup do it?

For ranking, prioritization, or scoring problems, a weighted heuristic with a lookup table beats an LLM on cost, latency, predictability, and explainability. You can debug a heuristic. You can A/B test a weight. You cannot debug "the model decided." Save the model for the hard cases.

5. Now use a model.

By the time you get here, you've earned it. The remaining cases are: open-ended generation, synthesis across many sources, judgment calls under ambiguity, conversational interfaces where the value is in the dialogue. These are real LLM use cases. They're a smaller slice of your product than you think.

The hybrid pattern (the one most teams miss)

The most powerful pattern in 2026 isn't "use AI" or "don't use AI." It's "use a rule for 80%, escalate to a model for 20%."

A few examples from products I've worked on:

- **Support ticket routing.** Rule-based on keywords for the obvious cases. Model only for ambiguous tickets. About 80% cost reduction with no quality drop.

- **Document extraction.** Regex for structured fields (dates, amounts, IDs). Model for the unstructured remainder. About 90% cost reduction.
- **Search.** Lexical first. Vector as fallback when lexical returns under five results. Cuts retrieval cost in half.
- **Content moderation.** Rule-based for slurs and known bad patterns. Model only for ambiguous edge cases. Faster, cheaper, more legally defensible.

The PM job is identifying the 80%. Most PMs assume the whole thing is the 20%, ship a model-only solution, and discover they're losing money on every transaction.

The "form versus chat" gut check

For any chat interface in your product, ask three questions:

1. Could a dropdown collect the same input?
2. Could a 3-5 step wizard replace the conversation?
3. Are users typing roughly the same phrases repeatedly?

If yes to any of these, the chat is wrong. Every PM eventually learns this by shipping a chatbot, watching the analytics, and discovering 70% of the user inputs are the same five phrases. That's a form.

The cost case

A 1,500-token prompt with a 500-token completion on a flagship model costs about a cent. Sounds like nothing. Now multiply: 100 actions per user per day, 50,000 active users, 30 days. That's \$1.5M a month for a feature a regex could have done for free.

Your CFO will eventually find this. Better that you find it first.

The reverse is also true. A feature that really needs a model (like summarizing a 50-page legal document into a one-paragraph brief) is worth the cost ten times over and shouldn't be cheaped out by a rule that produces garbage. The PM job is knowing which is which. The way you know is by running both and looking at the eval scores against the cost.

The cultural fight

A peer (sometimes a founder, sometimes the CEO) will want the product to be "AI everywhere" because that's what the market is rewarding. They're confusing a marketing position with a product decision.

The market rewards companies that use AI thoughtfully. It doesn't reward companies that wrap a regex in a chatbot for the demo. The customer eventually figures out which is which. So does the bill payer. Credibility comes from the LLM working brilliantly where it matters and being absent where it doesn't. That's the AI-native posture.

Pick one thing this week

Pick one AI feature in your current roadmap. Apply the decision tree to it.

1. Could a rule do it? If yes, what's the rule?
2. Could a query do it? If yes, what's the SQL?
3. Could a form do it? If yes, sketch the form.
4. Could a heuristic do it? If yes, what's the formula?
5. If it truly needs a model, write one sentence explaining why a rule wouldn't work.

If you can't make it past step 5 with confidence, the feature is an AI feature for marketing reasons, not product reasons. Either kill it or ship the simpler version first and see if users notice.

AI is a tool. Putting it down when something cheaper does the job is the move that actually makes you look senior.

Once you know where a model is actually required, [Prompt Ops](#) covers how to version, test, and deploy those prompts safely. For the eval harness that validates model quality before any prompt reaches production, see [The Eval Is the Spec](#). And for the cost-side argument you will need to make to your CFO, see [Gross Margin Is Your Job Now](#).

Gross Margin Is Your Job Now

Cost per successful action is the new primary PM metric. If you don't own it, your CFO will kill your product before your customers do.

Web version: <https://falkster.com/handbook/gross-margin-is-your-job-now> · last updated 2026-06-12

The short version

Gross margin is now a PM decision, not a finance one. AI-first SaaS runs 55 to 70 percent gross margin against traditional SaaS at 78 to 85 percent, and the gap is entirely driven by product choices: model routing, prompt hygiene, caching, and early-exit logic. Three levers every PM can pull without becoming an ML engineer. First, build a routing table and send classification, extraction, and format conversion tasks to models 10x cheaper than your flagship. Second, cut production prompts ruthlessly: most are 3 to 10x longer than needed and cutting a prompt from 3,000 tokens to 600 typically saves 80 percent on that surface with no eval regression. Third, turn on prompt caching (50 to 90 percent cost reduction on repeated prefixes) and design early-exit rules for the top 20 percent of inputs by volume. Watch one metric: cost per successful action by surface. If you don't have it, finding out why not is your week.

The metric nobody taught us to care about

For thirty years of my career, marginal cost on a software feature was basically zero. My job was demand-side: engagement, retention, conversion, NPS. The CFO worried about the supply side. We stayed in our lanes. That deal is described more broadly in [The Product Operating Model](#), but this chapter is where the AI-era cost reality lands.

That deal is over. In 2026, every AI feature I ship has a cost line that moves with usage. Token costs. Retrieval costs. Model routing. Vendor markup. Latency-driven retries. These variables move gross margin by 20 to 40 percentage points on a product. That's the difference between a venture-backed business and a venture-killed one.

And if the PM doesn't own cost, nobody does. The CFO sees a number at month end. The engineer sees a prompt at the character level. Only the PM sees the full customer job end-to-end. Only the PM can say "this entire flow costs too much and needs to be rebuilt."

So gross margin is my job now. Welcome to the role.

The numbers that should ruin your afternoon

If any of these surprise you, we have work to do.

- AI-first SaaS gross margins are running 55 to 70 percent in 2026. Traditional SaaS runs 78 to 85 percent. That's 20-plus points of margin I have to earn back through product decisions.
- A B2B app processing 50M tokens per enterprise customer per month spends \$500 to \$2,000 in raw inference, before retrieval, before tools, before any other COGS.
- Companies still pricing per-seat in AI are running 40 points lower gross margin than companies that moved to hybrid usage pricing. Same product, different pricing, different business.
- [Anthropic](#) Sonnet 4.5 doubles in price above the 200K token input threshold. The PM who doesn't know that threshold ships a feature that silently halves margin the first time a real customer tries it with a large document.

These are not finance problems. They are product decisions showing up on the finance ledger. The person making them should be you.

The three levers every PM can pull

You don't need to become an ML engineer. You need to make three decisions well.

Lever 1: Model routing.

Not every task needs your flagship model. Classification, structured extraction, simple rewrites, and format conversion often work on a 10x cheaper model with comparable quality. My job is to know which surfaces in my product can be down-routed without eval scores dropping.

The artifact: a routing table. For each surface: what model, what fallback, what eval threshold triggers a swap. I review this quarterly. About once a quarter I find a surface that was over-routed to the flagship for no reason other than we shipped it that way.

Lever 2: Prompt and context hygiene.

Most production prompts in 2026 are three to ten times longer than they need to be. PMs over-specify because they don't trust the model. Engineers inherit the prompt and don't prune it. Context windows get packed with "just in case" instructions that you pay for per token.

Cutting a prompt from 3,000 tokens to 600 (which is usually possible without quality loss) cuts cost on that surface by about 80 percent. I don't need an engineer for this. I open the prompt, cut ruthlessly, run the eval, keep the cuts that didn't drop the score. Most of my cuts survive.

Lever 3: Caching and early exit.

Prompt caching on [Anthropic](#), [OpenAI](#), and Gemini cuts cost 50 to 90 percent on repeated prefixes (system prompts, tool definitions, long instructions). Most teams aren't using it. Early-exit logic (don't call the LLM if a rule or a cached answer will do) cuts cost 100 percent on the hit. My job is to identify the top 20 percent of inputs by volume and design for cache hits or rule shortcuts on them.

The dashboard I actually watch

Every AI product I own has a dashboard with these metrics, visible at all times:

- Cost per successful action (by surface).
- 7-day trend of that number.
- Latency p50 / p95 / p99 (by surface).
- Token volume (input vs output, by surface).
- Cache hit rate.
- Model distribution (how much traffic hits which model).
- Failed actions and their cost (yes, failures still cost money).

If this isn't on your wall, you're guessing. I had the ugly version of this as a Notion page for six months before anyone built a real dashboard. The ugly version was still better than not having it.

The pricing interaction

Cost ownership is pointless if pricing doesn't flex with it. This is why [Pricing for AI Products](#) is the natural next chapter. Cost and pricing are two sides of the same decision: is the customer paying enough for the specific flow they just ran?

In a world where a single “answer this from my docs” query can range from 5 cents to 5 dollars depending on document size, per-seat pricing is an arbitrage your customers will win every time. Your pricing has to map to value units. You can't design that without knowing the cost units first.

What the old playbook got wrong

The 2018 playbook was: ship the best product, optimize cost later, scale compresses unit costs. The [Eval Is the Spec](#) chapter explains why you also can't separate cost from quality: every prompt change that cuts cost should run through an eval to confirm it didn't regress quality. That worked when unit cost was deterministic, had steep scale curves, and was fractions of a cent per action.

None of that holds now. Unit cost is stochastic (depends on prompt length, tool calls, retries). Scale curves are flat (you pay per token whether you have 10 users or 10 million). Cost per action is 1 to 20 cents.

“Optimize later” means “rewrite the product later.” That’s not a runway you have.

Pick one thing this week

Here’s a 90-minute exercise that will embarrass you if you’ve never done it before.

1. Pick the most-used AI surface in your product.
2. Find the cost per successful action over the last 30 days.
3. If you don’t have that number, stop and figure out why you don’t have it. That’s your week.
4. If you do have it, find the top 10 percent of requests by cost. What do they have in common?
(Long context? Specific customer? A loop?)
5. Apply one of the three levers to that slice: route to a cheaper model, cut the prompt, add a cache.
6. Ship the change. Measure again in 7 days.

My first time doing this on a surface at Smartcat, I found a prompt that was 4,200 tokens because three different PMs had added instructions over six months. I cut it to 800. Same eval score. The feature got 70 percent cheaper overnight.

In an AI product, every feature has a cost line. If you don’t know which of yours is bleeding out, one of them is, and your board will find it before you do.

Pricing for AI Products

Per-seat is dead for AI. Price the work the seat is no longer doing: outcomes, usage, value units.

Web version: <https://falkster.com/handbook/pricing-for-ai-products> · last updated 2026-06-12

The short version

Per-seat pricing is broken for AI products because cost-to-serve now scales with usage, not with license count. Power users at a \$50 seat can cost \$200 to serve. Companies still on per-seat pricing in 2026 run gross margins 40 points below those on hybrid or outcome-based models. The four models that work: hybrid (base fee plus usage), outcome-based (pay per resolved ticket or successful action), tiered consumption (buy a bucket upfront), and pure usage (pay for what you use). The hardest decision is picking the right value unit: customers must be able to tell you what 100 units will do for them before signing. If they can't, you will churn them on their first big bill. Migrate without losing the book by grandfathering existing customers, selling hybrid to all new accounts, and re-pricing expansion on the new model only.

Per-seat is dead and you can stop pretending

The data is clear and has been for about a year. Companies still pricing AI per-seat in 2026 are running gross margins 40 percentage points below companies that moved to hybrid or outcome-based pricing. The market figured it out. Customers figured it out. Most PMs are still building roadmaps assuming the seat count is the lever, because that's what the 2018 SaaS playbook said.

The 2018 playbook was written for a world where marginal cost was zero. That world is over. The compute cost shift is part of a broader planning change described in [The Product Budget Is a Compute Budget Now](#).

Why per-seat breaks

Per-seat pricing optimized for predictability. Customer knew what they'd spend. Sales knew what to forecast. Finance knew what to model. It priced *access to a tool* in a world where the tool's cost-to-serve was negligible.

Now cost-to-serve is the largest line item on my COGS, and it scales with what the user does, not whether they have a license. A power user at \$50 a seat might cost me \$200 a month to serve. A dormant seat costs me \$50 and earns me the same. I'm subsidizing my power users with my dormant

ones, until the power users add a few more, the subsidy collapses, and my gross margin disappears in a single quarter.

The customer figured this out faster than I did. They're buying fewer seats and using them harder. Seat count goes down as usage goes up. That's the death spiral.

The four models that work now

Hybrid is a base fee plus usage. A floor that covers fixed costs and basic access. Usage on top that scales with a value-aligned metric. Adoption jumped from 27 percent to 41 percent of AI software companies in a single year. This is the safest move from per-seat. Customers tolerate it because the base is predictable and usage is "fair."

Outcome-based means you charge when the agent succeeds. Customer pays per resolved support ticket, per successful translation, per document processed, per qualified lead. Highest alignment between price and value, and the hardest to instrument. Highest pricing power too, if you can pull it off, because the customer is paying for the result they would have hired a human to deliver, at a fraction of that cost while you still earn a software margin.

Tiered consumption is a bucket of usage units bought up front, with overage at a higher rate or a forced tier upgrade. Familiar to anyone who's bought a cell phone plan. Predictable bill, predictable revenue, easy to sell. Mediocre value alignment but still better than seats.

Pure usage is pay for what you use, no minimum. Strongest alignment with cost-to-serve, worst predictability. Use this only when you have a genuine self-serve product where churn is low because value is concrete every time. Most B2B SaaS doesn't have this, and pure usage will burn you on evaluator customers.

Picking the value unit is the actual hard work

Most teams pick the wrong unit and discover six months later that customers are optimizing against it.

Bad value units: - Tokens. Customers don't think in tokens. Bills feel arbitrary. - API calls. Customers batch to avoid charges and break the product. - Compute time. Punishes customers for using your product.

Good value units: - Successful outcome. A resolved ticket, a closed deal, a published article, a translated document. - Active workflow run. The agent did the thing. Customer paid. Failed runs don't bill (non-negotiable; bill on failure once and the customer never trusts you again). - Document, conversation, or session. Discrete, value-coherent, easy to count, easy to explain.

The litmus test I use: *can the customer tell me, before signing, what 100 of my value units would do for them?* If yes, the unit is right. If no, I'm pricing on something they don't understand and I'll churn them on their first big bill.

The instrumentation requirement

You can't price what you don't measure. Before I change pricing on anything, I need:

- A defined "successful outcome" for every billable surface.
- Real-time tracking of outcomes per customer, per workspace.
- Cost-per-outcome telemetry (so I know margin per unit, per customer).
- A billing system that can ingest usage and reconcile to invoices.
- A customer-facing dashboard so they see usage in real time.

The last one is the trust contract. Customers will accept usage pricing if and only if they can see usage as it happens. Surprise bills are the killer. Build the dashboard before you change the price.

How to migrate without losing the book

You can't just switch pricing. You'll lose half your accounts at renewal. What works:

1. **Grandfather existing customers.** Their pricing stays. Stop selling that pricing to new customers the same day.
2. **Sell hybrid to all new accounts.** Watch the unit economics for a full quarter.
3. **Offer existing customers an opt-in to the new model with a discount.** Many move because the new model is cheaper for them at their usage. The ones who would pay more either stay grandfathered or churn. You learn who was unprofitable.
4. **Re-price expansion seats and new modules on the new model only.** Within 18 months, half your revenue is on the new model without a single forced migration.

Skipping step 1 is how companies make headlines for "raising prices." Be slow on migrations, fast on new-customer pricing.

The outcome-based endgame

The endgame for most AI products is outcome pricing. The economics are unbeatable: perfect alignment with the customer, competitors look like rent-seekers, and I can capture a fraction of the value I create, which is much larger than cost-plus margin allows.

Path I run: start with hybrid (base plus usage on a value-aligned unit). Tighten the value unit until it's an outcome, not an action. Add an SLA on the outcome (success rate). Charge premium pricing on

the SLA tier. Within two product cycles, you've moved a meaningful surface to outcome pricing without a marketing campaign about it. The customer just sees a product that works and a bill that feels fair.

Pick one thing this week

Don't redesign your pricing page. Do this instead. For the field report on what actually happened when one company killed its per-seat tier, see [Field Report: Killing the Per-Seat Tier](#). For the 18-month migration playbook, see [Pricing Migration: 18 Months to Outcome Pricing](#).

1. For one billable surface, write down the value unit the customer actually cares about. Not tokens. Not API calls. The thing they came to get.
2. Instrument that unit. Start tracking it today, even if you don't use it in billing yet.
3. Pull your top 10 customers by revenue. Calculate what they'd have paid on the new unit over the last 90 days versus what they actually paid. Find the outliers.
4. You'll find two patterns: customers you're under-pricing (high usage, low revenue) and customers you're over-pricing (low usage, high revenue). That asymmetry is your pricing redesign, sitting in front of you.
5. Sketch what a hybrid model would look like. Share with your CFO. Watch the conversation get a lot more concrete.

In an AI product, you don't price the seat. You price the work the seat is no longer doing.

Prompt Ops

Your prompts are production code. Version, review, eval, stage, and roll back, or your product is one Notion edit away from breaking.

Web version: <https://falkster.com/handbook/prompt-ops> · last updated 2026-06-12

A bad day I'd like you to avoid

Six months ago I watched a team push a “small prompt tweak” to production on a Friday afternoon. By Monday morning, customer-facing quality scores had dropped 18 percent across three surfaces. Nobody knew exactly when the change went in. The “old” prompt only existed in a Slack thread someone had to scroll back to find. Rollback took six hours.

The team was good. They were not careless. They just hadn't built any operational layer underneath their AI product, so a single intern with edit access to a Notion page could break production for everyone at once.

If your prompts live in a Google Doc and get pasted into the codebase by hand, you don't have a product. You have a liability with a UI on top.

The short version

Prompt Ops means treating prompts with the same lifecycle as production code: version control, pull request review, automated evals on every change, staged rollout (1%, 10%, 100%), monitoring, and one-click rollback. The five-piece stack takes half a day to set up for one prompt. The PM owns the prompt (it is the spec, encoding what the system does, who it's for, and what tone it uses); engineering owns the wiring. Most teams break because prompts live in three places at once, edits are untracked, there is no testing layer, and rollback takes hours during a production outage. Start by moving one prompt into your repo this week, wiring your code to load from the file, and adding an eval set into CI. Build from there.

What I see broken on most teams

Prompts live in three places at once. The “real” one in code, a “draft” in Google Docs, an older version in a Notion page someone forgot existed. When something breaks, nobody knows which prompt is actually running.

Edits go untracked. Someone tweaks a prompt to fix one customer's edge case. The fix breaks three other use cases. The post-mortem reveals “the prompt was changed at some point” and the team

agrees to “be more careful.”

There’s no testing layer. Prompts ship straight to production. The only test is “does it look right when I run it manually three times.” Three runs of a non-deterministic system is a vibe, not a test.

And rollback is a redeploy. If a prompt breaks, the fastest path to recovery is “find the old version, paste it back, redeploy.” That takes hours. During those hours the product is broken for every user.

The five-piece Prompt Ops stack

You don’t need exotic tooling. You need five things, in this order.

1. A prompt repository. Prompts live in your codebase, in version control, in a directory called `prompts/`. Each prompt is a file. Each file has a header with its purpose, owner, and last-evaluated-at timestamp. The prompt that runs in production is the file in `main`. There is one source of truth. There are no Google Docs.

2. A prompt review workflow. Prompt changes go through pull requests, exactly like code. The PR template requires three fields: what changed, what eval set this was run against, what the score delta was. PRs without an eval run can’t merge. Reviewer’s job is to look at the diff, the eval delta, and the failing examples. This is faster than the current “did Slack approve it” process and infinitely more rigorous.

3. An eval harness on every prompt change. Before merge, the prompt is automatically run against its named eval set. Score, deltas per slice, newly failing examples get posted to the PR. You merge when the eval clears the threshold. This is the AI-product version of a CI test suite. Without it, you don’t have a deploy pipeline. You have hope.

4. Staged rollout. New prompts deploy to 1 percent of traffic first. Real-world eval scores get monitored for 24 hours against the same eval set running offline. If live agrees with offline, ramp to 10 percent, then 50 percent, then 100 percent. If they diverge, you’ve found a gap in your eval set. Pause, augment the eval set, try again. You’ll be surprised how often offline and online disagree. That gap is the actual product risk you’re managing.

5. One-click rollback. Every deployed prompt has a one-command revert. The on-call engineer can roll back without a ticket. The PM can roll back without an engineer. Rollback is a routine action used confidently, not a heroic act.

A/B testing prompts in production

You’ll eventually have two viable prompts and want to know which is better. The naive approach is offline evals, but production user behavior is the real test. The right approach:

- Define a primary metric (eval score, completion rate, customer rating, downstream conversion).
 - Split traffic 50/50, or 90/10 if one variant is risky.
 - Run for at least one full weekly cycle to absorb day-of-week effects.
 - Decide. Pick the winner. Delete the loser. Don't carry both prompts forward "just in case."
- That's how you end up with seven prompts for one feature and nobody knows which is which.

Prompt linting

Most production prompts in 2026 have at least one of these bugs:

- Conflicting instructions ("be concise" with a six-paragraph guideline that asks for verbose output).
- Stale references (mentions a tool or capability that no longer exists).
- Untracked dependencies (assumes a variable will be passed but doesn't validate).
- Format-locking that fights newer models (overly rigid format prescriptions newer models handle natively).
- Tone collisions (one paragraph says "be friendly," another says "respond formally").

A simple linter, a small LLM call running over your prompts/ directory once a week, catches most of these. Treat the lint as advisory, not blocking. But read the report. The first time you'll be embarrassed by what you find. That's the point.

Who owns prompts?

The PM owns the prompt. Engineering owns the wiring around it. This is non-negotiable.

The temptation is to "let engineering write the prompt because they understand the system." That's exactly backwards. The prompt is the spec. The prompt encodes the product decision: what the system does, who it's for, when it refuses, what tone it uses, what edge cases it handles. Those are PM decisions. Engineering's job is to make sure the prompt loads cleanly, runs reliably, evaluates automatically, and rolls back fast.

If your engineers are writing your prompts, you haven't yet done the work of translating the product decision into a contract. The prompt is the contract. Write it.

Pick one thing this week

You probably have at least one prompt that lives in a Google Doc or a Notion page right now. Move it.

1. Create a directory in your repo called `prompts/`.
2. Move that one prompt into a file called `prompts/feature-name.md`. Add a header: purpose, owner, last evaluated.
3. Wire your code to load the prompt from the file (one or two lines for most stacks).
4. Make any change to the prompt. Open a PR. Notice the diff is now legible.
5. Add an eval set for that prompt (see [The Eval Is The Spec](#)). Wire it into your CI. Now any prompt change runs the eval.

That's the v1 of Prompt Ops. Five things, half a day of work, and your product gets meaningfully harder to break by accident. Do this for one prompt this week. Add a prompt to the system every week after that. By the end of the quarter, your AI product has the operational maturity of your CRUD product, and the next intern can't break production.

The eval set you wire in step 5 is the foundation of [The Eval Is the Spec](#), the chapter on making evaluations the primary artifact of AI product work. For how to ship AI features with the observability layer that makes staged rollouts measurable, see [Ship With Observability](#). And if you want to understand when to reach for a prompt versus a rule or query, see [When Not to Use AI](#).

The Living Changelog

Your model vendor changed the model on Tuesday and didn't tell you. Run a daily replay against production or your customers will catch it before you do.

Web version: <https://falkster.com/handbook/the-living-changelog> · last updated 2026-06-12

The short version

The Living Changelog is a continuous eval replay against production. You run the same eval set every day against the live system and treat any delta beyond noise as an incident, even before a customer complains. Model vendors change behavior without telling you: snapshots get rerouted, safety training updates silently, quantization changes roll out without a release note. The system has three parts: a replay set (50-300 production examples), a daily automated run that stores scores with timestamps, and a drift alarm set at 2x the noise floor. When the alarm fires, a six-step runbook covers confirmation, vendor check, version pinning, fix decision, customer communication, and eval set update. Build the system for one surface this week and you will catch your first silent vendor regression before any customer does.

Your product changed last Tuesday

A model vendor deprecated a snapshot. Or rerouted a checkpoint behind the scenes. Or pushed a safety-training update that suddenly refuses requests it used to handle. Or fixed a tokenization bug that changed the output format. You didn't get an email. You didn't get a release note. Your customers found out before you did, in the worst possible way: by complaining about a thing that used to work.

This is the new operational reality of building with AI. You shipped a product you don't fully control. Pretending otherwise is the fastest way to lose your customers' trust.

What to do about it

Run a continuous eval replay against production. The same eval set, every day, against the live system. Any delta greater than noise is treated as an incident. Even if no customer has complained yet. You assume the model providers will change things without telling you, because they will, and you build the operational layer that catches it before your customers do.

Why "test once, ship, forget" doesn't work anymore

Most teams test once. Run an eval when the prompt or model changes, get a score, ship it, and never run that eval again. Meanwhile the model behind the API can shift behavior weekly without any version bump on your side.

Vendors don't always tell you. Even when major model versions deprecate, the comms are imperfect. Snapshots get rerouted. Safety training gets updated mid-version. Quantization changes get rolled out silently. Read your model vendor's changelog and assume that 80 percent of what they actually change is not on it.

Your customers will tell you. Eventually. After it's been broken for two weeks. Via a sales escalation. After they've already mentioned it to a competitor.

And without continuous replay, when a customer says "this got worse," you can't reproduce the regression. You can't show what changed. You can't date the regression. You don't have operational ground truth.

The replay system, three components

It's simpler than it sounds.

1. The replay set. Take the production eval set you use as the spec for each surface (see [The Eval Is The Spec](#)). This is your ground truth. Each surface has one. The set is small enough to run cheaply (50 to 300 examples), large enough to detect drift across slices.

2. The daily run. Every day, automatically, the replay set runs against every model your product uses, with the prompts currently live in production. Scores are stored with timestamps. Per-example outputs are stored with timestamps. The store is small (a few MB per surface per day) and it's the only way you'll ever be able to answer "when did this start drifting?"

3. The drift alarm. Define a noise floor for each surface. You'll learn it after a week of data. Any per-day change beyond that floor triggers an alert. The alert goes to the PM and the on-call engineer. It includes: which examples changed, what the previous outputs were, what the new outputs are, and which model and prompt are currently in production.

That's the whole system. A few days to build. It pays for itself the first time it catches a silent vendor change before a customer does.

The runbook when the alarm fires

- 1. Confirm the regression.** Look at the failing examples. Are they actually worse, or did the eval rubric drift? (Yes, the LLM-as-judge can also drift. You may need a second model to judge the judge.)

2. **Check the vendor.** Has the model been updated? Is there a published changelog item? Is there community chatter about the same regression?
3. **Pin the model version.** If the vendor offers pinned snapshots, switch to one immediately. Buy yourself time.
4. **Decide: prompt fix, model swap, or vendor escalation.** A prompt change is fastest. A model swap is most stable. A vendor escalation is the right move if you're a major customer and this is a violation of their stability commitments.
5. **Communicate.** If customer impact is non-zero, send a proactive note. "We detected a regression in [feature]; here's what we're doing." Doing this before customers complain is the trust differentiator. Doing it after is damage control.
6. **Update the eval set.** The fact that this regression made it into production means your eval set was missing something. Add the failing case. Strengthen the rubric. The eval set gets better every time the alarm fires.

The vendor contract conversation

If you're spending real money with a model provider, you have leverage to ask for:

- Pinned model snapshots with stability commitments and notice periods before deprecation.
- Pre-release access to upcoming model versions so you can run your eval set before they go GA.
- A direct support channel for behavior regressions.
- A documented escalation path for production incidents tied to model behavior.

Most vendors will offer these to enterprise customers. Most PMs don't ask. Ask. Even if you're not yet enterprise-tier, asking signals that you operate the way they want their large customers to operate.

Your customer-facing version

Somewhere on your status page or trust center, your customers should be able to see:

- Which models you use, by surface.
- When you last validated each surface.
- How you handle regressions.
- The notice period you give when you change models.

This is the new "trust center" content for the AI era. Most teams haven't published it yet. Be early. Customers buying AI products in 2026 increasingly check this before signing, especially in regulated industries. The absence is a sales blocker more than the presence is a sales asset.

The mental model shift

You used to ship code, and the code did what it did until you changed it.

You now ship a system in which a third party can change part of it, at any time, without your permission. Your product is partially controlled by your model providers. The Living Changelog is how you take that control back at the operational layer. You can't stop the vendor from changing the model. You can detect the change before it hurts your customers.

This is uncomfortable. It's also the job.

Pick one thing this week

Pick the most important AI surface in your product. Set up its replay.

1. Take the eval set you already have for it. (If you don't have one, see [The Eval Is The Spec](#). That's your week.)
2. Schedule it to run daily against production, automatically. A small Python script and a cron job is fine.
3. Store the score in a sheet, with a timestamp.
4. After a week, look at the variance. That's your noise floor. Set an alert at 2x the noise floor.
5. The first time the alert fires, run the runbook above. Notice that you knew about the issue before any customer told you.

Do this for one surface this week. Add another every week. Within a quarter your top 10 surfaces all have continuous replay, and you become the team that catches model drift before any of your competitors. For the customer-facing trust story around model changes, see [Ship with Observability](#). For the prompt-level foundation this builds on, see [Prompt Ops](#).

You shipped a product with a third party in the loop. Either you watch the loop daily, or your customers will tell you when it broke.

Trust, Safety, and the Guardrail as a Product Decision

Every guardrail is a product decision. The PM who outsources it to legal gets a product they didn't design and a customer experience they wouldn't approve.

Web version: <https://falkster.com/handbook/trust-and-safety> · last updated 2026-06-12

The short version

Every guardrail in an AI product is a product decision: what you refuse, what you warn on, what you silently log, what you allow with a disclaimer. Outsourcing that to legal produces a product that is both annoying and unsafe. The guardrail tier system has four levels: Tier 0 (hard block, list under 15 items or you are over-blocking), Tier 1 (soft warning, product proceeds with a flag), Tier 2 (logged only, product proceeds normally), Tier 3 (allowed, the default for most inputs). The most common failure is the overly cautious assistant that refuses legitimate requests and churns customers silently. Track refusal rate as a primary product metric alongside conversion. The real top three risks in most AI products are: the product states something false, the product reveals data it should not, and the product takes an irreversible action incorrectly. Address those before the legal team's list.

The team you accidentally outsourced your product to

Most companies have outsourced “trust and safety” to legal and compliance. That worked when the product was deterministic and rules were enforced by a config file. It doesn't work now. Every guardrail in an AI product is a product decision: what you refuse, what you warn on, what you silently block, what you allow with a disclaimer, what you escalate to a human.

These decisions shape the product experience more than your feature decisions do. And in 2026, the difference between a product customers trust and a product they abandon is mostly in guardrail design.

Take it back from legal. Own it as product.

Why the old model breaks

The old model: legal writes a policy, hands it to engineering, engineering implements a “safety filter,” product team works around the filter when it's annoying. The result is a product where:

- Refusals feel arbitrary and inconsistent.

- The same input gets different responses depending on phrasing.
- The product apologizes for things that aren't problems.
- Real risks slip through because the filter was tuned for the wrong things.
- Engineering builds shadow workarounds because the filter is in the way.
- Customers learn to “jailbreak” past the filter, which works, which means the filter wasn't really doing anything except annoying compliant users.

This is a worst-of-all-worlds outcome. The product is annoying *and* unsafe. You'd be better off with no filter, which is a sentence you don't want to say to your CEO. But it's true.

The guardrail tier system

What I run instead is a structured guardrail design with explicit tiers. Every potentially-risky behavior gets categorized.

Tier 0: Hard block. Product refuses, full stop, with a clear explanation. Reserved for behaviors that are truly dangerous, illegal, or violating to users. The list is short. If your Tier 0 list has more than 15 items, you're over-blocking and your refusal rate will hurt retention.

Tier 1: Soft warning. Product proceeds, flags the user. “I'll do this, but here's something to consider.” For grey areas where context matters. The user is treated like an adult who can make their own decision.

Tier 2: Logged-only. Product proceeds normally; the action is logged for review. For when you want telemetry on edge behavior without a user-facing intervention. Most “trust and safety” instrumentation should live here, not at Tier 0.

Tier 3: Allowed. No filter. The default for the vast majority of inputs.

Every input passes through this hierarchy. Most pass to Tier 3. A small fraction trigger Tier 2 logging. A smaller fraction trigger Tier 1 warning. A tiny fraction trigger Tier 0 refusal.

I define what goes in each tier. Legal reviews. Security audits. I ship.

Refusal rate as a product metric

The move most teams haven't made: track refusal rate as a primary product metric.

Refusal rate up week-over-week is a problem. It means the model is becoming more cautious (often due to silent vendor updates, see The Living Changelog), or your guardrails are over-firing, or your prompts are inadvertently triggering refusals, or your users are testing limits more aggressively. All PM problems.

Refusal rate down week-over-week is also a problem. Maybe your guardrails are eroding, new edge cases are slipping through, or you’ve moved to a less-aligned model. Investigate.

A healthy product has a stable, low refusal rate with explainable spikes. A product with a wandering refusal rate has a guardrail design problem.

Add refusal rate to your dashboard. Treat it like conversion or activation.

The “overly cautious assistant” failure mode

The most common AI product failure I see in 2026 is the overly cautious assistant. The product won’t summarize a news article because it might contain political content. Won’t answer a medical question because it might be advice. Won’t write the email because it might be persuasive. The user concludes: this product is useless. They cancel.

This failure is invisible if you only watch for “harmful outputs.” The product is producing zero harmful outputs. It’s also producing zero useful outputs. The metric you need: per surface, what percent of legitimate requests get refused?

If that number is over a single-digit percent for general-purpose features, your product is broken in the most expensive way: customers churn quietly, blaming themselves, and you never see it in your error logs.

Build the eval set that measures this. Real customer requests, labeled “legitimate” or “actually problematic.” Run the refusal eval daily. Optimize aggressively for legitimate-request approval, with hard bounds on the truly-problematic refusal rate. The shape of the curve matters: high approval on legitimate, near-perfect refusal on problematic, and a small explainable middle ground.

How loud should the caveat be?

When the product proceeds with caveats, how loud is the caveat?

Loud caveats erode trust (“the assistant won’t shut up about its limitations”). Silent compliance is risky (“the assistant just did a thing without acknowledging the risk”). The right answer is contextual:

- For high-stakes domains (health, legal, finance), the caveat is part of the value.
- For everyday tasks, the caveat is friction. Skip it.
- For ambiguous tasks, the caveat is a one-liner offered once per session, not on every turn.

The instinct to over-caveat comes from a defensive crouch, covering yourself in case anything goes wrong. Customers read it as the product being unsure of itself. They lose confidence. They leave. Be calibrated, not defensive.

Map the actual risks

Spend a day mapping the real risks in your product. Not the theoretical ones legal would worry about. The real ones, ranked by probability times impact.

Most products' real top three risks are some version of:

1. The product confidently states something false (hallucination).
2. The product reveals data it shouldn't (data leak across users, or from documents the user shouldn't see).
3. The product takes an irreversible action incorrectly (sends an email, makes a change, completes a transaction).

Legal will worry about other things: copyright, defamation, regulatory exposure. Those are real but usually downstream of the operational risks above. Address operational ones first; legal ones get smaller automatically.

For each risk in your top 5, you should have:

- A specific eval that tests for it.
- A guardrail tier assignment.
- A logged metric.
- A documented response runbook.

If any of these is missing for any of your top 5, that's the gap.

Pick one thing this week

Spend 90 minutes writing your trust and safety design doc.

1. Open a doc. Title it "Trust and Safety Design."
2. List your tiers (Tier 0 through Tier 3) and what behaviors fall into each.
3. List your top 5 product risks, with their evals and runbooks. If you don't have an eval for any of them, mark them as gaps.
4. Calculate your current refusal rate per surface (or note that you don't measure this and add it to the gap list).
5. Schedule a 30-minute review with one person from legal and one from security. Walk them through the doc. Ask: "What am I missing?"

The doc is now the contract between product, legal, and security. With it, the conversation is "do we agree on this design?" Without it, the conversation is "you should have thought of that" in retrospect, after the incident.

A guardrail is a product decision. The PM who outsources it gets a product they didn't design and a customer experience they wouldn't have approved.

For more on the eval systems that enforce guardrail quality automatically, see [The Eval is the Spec](#). For the broader framework on when not to use AI in a product, see [When Not to Use AI](#).

The PM Agent Stack: Open-Source Tools Mapped to PM Work

Build a personal PM agent stack from open-source Claude repos. 18 tool categories mapped to the 7-stage PM operating system, with install order and how-tos.

Web version: <https://falkster.com/handbook/pm-agent-stack> · last updated 2026-05-18

SERIES · THE PM AGENT STACK · PART 5 OF 5 (CANONICAL)

This chapter is the canonical reference for the PM Agent Stack series. The four blog posts are the way in; this is the index they all point back to.

1. 1. [Overview: the destination, the gap, the bridge](#)
2. 2. [Discovery agent stack](#)
3. 3. [Build agent stack](#)
4. 4. [Measure agent stack](#)
5. 5. **The PM Agent Stack handbook chapter** ← you are here

The short version

The destination for every product organization is one AI brain with read access to every system the company runs. Slack, email, calendar, meetings, documents, source code, dashboards, CRM, design files. All of it. Not parts. All. Full stop.

Most companies are 6 to 18 months from that destination because procurement, security review, and data governance move slowly and PMs are not the buyers for the platform layer. This chapter is the bridge: it maps 18 categories of open-source Claude repos to the 7 stages of [the PM operating system](#) (Sense, Discover, Decide, Build, Ship, Measure, Amplify), gives the install order, and points to three concrete how-tos: [Discovery agent stack](#), [Build agent stack](#), and [Measure agent stack](#). The stack works on the inputs a single PM has access to. It does not replace the enterprise brain. It compounds while you wait for the enterprise brain.

This is one connected system, not a set of independent tools. Skim the chapter once. Bookmark the install order. Walk through one trailing how-to. Install five tools this week, not fifty.

The destination and the gap

The destination is an enterprise-wide AI brain. One agent with access to every Slack thread, every meeting transcript, every doc, every PR, every dashboard, every CRM record, every design file.

When the agent has access to everything, it stops feeling like a tool and starts feeling like a colleague who has been at the company for years. The companies that have built this layer first are already pulling ahead in every category. R&D figured it out earliest. Go-to-market is catching up. G&A is right behind.

Most PMs are not there yet. Three structural reasons. Procurement and security cycles take 6 to 18 months for a system that touches this many sources. The decision happens in the office of the CTO, the CIO, or the COO; PMs are not the buyers. And many companies are scarred from earlier knowledge-management projects that ate budget and shipped nothing anyone used. That scar tissue is still there, and it still slows the rollout.

The bridge is what to build while the enterprise version is being procured.

An interactive before-and-after comparison lives on the web version:
<https://falkster.com/handbook/pm-agent-stack>

The bridge is single-PM scope by design. The trade-off is intentional. Standing it up takes a week of evenings, no permission, no procurement.

Credit where due: most of the catalog below is adapted from Divyanshi Sharma's 20-part Instagram carousel mapping the Claude ecosystem. The PM lens, the install order, and the mapping to the 7-stage PM OS are the contributions of this chapter.

The 18 categories of open-source repos

The Claude open-source ecosystem has matured fast. Thousands of community repos extend Claude Code. They cluster into 18 categories. You don't need to know every repo, but you need to know which categories exist so you can find the right one when a problem hits.

The categories, with a one-sentence PM summary for each:

1. **Awesome lists.** Master indexes. Start here when you don't know what you're looking for. Best entry: [hesreallyhim/awesome-claude-code](#).
2. **Meta indexes.** Lists of lists. Use when an awesome list isn't specific enough for your need.
3. **Anthropic repos.** First-party. Trust them. [anthropics/claude-code](#) is the base of the entire stack. Building on a base your competitor also owns has its own consequences, which I worked through in [what competing against Claude taught me about Falkster](#).
4. **Official adjacent.** First-party-maintained workflows like security-review and the github-mcp-server. Treat them as features.

5. **Skill collections.** Bundles of capabilities you bolt onto the agent. [obra/superpowers](#) is the foundation most PM workflows assume.
6. **Markdown skill libraries.** Skills shipped as standalone markdown files. Easy to fork and edit.
7. **Agents.** Roles with goals. A “frontend reviewer” or “user-research synthesizer.” Drop-in expertise.
8. **Subagents.** Agents inside agents. Used for delegation, specialization, and parallel work.
9. **MCP servers.** Connectors that give the agent access to specific systems (GitHub, Postgres, browsers, your codebase).
10. **Connect Claude / more MCPs.** Specific integrations like Playwright, SQLite, browser automation.
11. **Orchestration.** Coordinates multiple agents working in parallel. Skip until single-agent friction is real.
12. **Workflows.** Codified ways of doing work. Brainstorm, spec, plan, TDD, review.
13. **Memory.** Cross-session persistence. The single biggest unlock once you outgrow per-session work.
14. **Context engineering.** Tools like repomix and context-priming that get the right info in front of the agent at the right time.
15. **Slash commands.** Muscle-memory invocations. `/fix-issue 456`, `/security-review`, `/design-review`.
16. **Hooks.** Inject behavior at points in the agent loop: pre-commit, post-edit, on-error, scheduled.
17. **Guides.** Prompt engineering, system-prompt patterns, agentic coding patterns.
18. **Learning.** Other practitioners’ journeys. The most underrated category for PMs because workflow design beats tool count.

This chapter does not list every repo. The framing walk-through is in [the overview post](#), and the three trailing how-tos go deep on the subset of repos that matter most at each PM OS stage.

Mapping the 18 categories to the 7-stage PM OS

This is the mapping that earns its keep. It tells you which categories of tools to reach for at which PM OS stage.

Sense. What’s changing in the world, the market, the codebase, the user base. Tooling that matters: MCP servers (postgres-mcp, github-mcp-server), context engineering (repomix, claude-context), hooks for scheduled digests. The agent’s job here is to flag changes worth your attention before you go looking.

Discover. What problem are we solving and for whom. Tooling that matters: memory (claude-mem, semantic memory), skill collections for synthesis (customer-call-notes, the [continuous-listening](#)

chapter's recipes), Playwright MCP for scraping public reviews and forums, subagents that play the role of researcher, synthesizer, and devil's advocate. Full how-to in [Build a Discovery agent stack](#).

Decide. What to do next, given everything we know. Tooling that matters: subagent collections (wshobson/agents, davepoon's collection) for tradeoff analysis, multi-agent orchestration (claude-flow) for consensus, slash commands for prioritization rituals. The agent runs the analysis; you make the call.

Build. From decision to working software. Tooling that matters: workflows (Superpowers brainstorm-spec-plan-TDD-review), context engineering (repomix, graphify), TDD enforcement (tdd-guard), security review (claude-code-security-review), design review (design-review-workflow), claude-code itself. Full how-to in [Build a prototype-first agent stack](#).

Ship. From working software to in front of users. Tooling that matters: official adjacent (security-review GitHub Action, claude-code-action for PR reviews), hooks (pre-commit, pre-push), the [living changelog](#) practice, [observability hooks](#).

Measure. Did it work. Tooling that matters: MCPs (postgres-mcp for direct DB access, playwright-mcp for dashboard scraping), scheduled hooks for recurring digests, slash commands for variance detection, memory for tracking metric drift over time. Full how-to in [Build a measurement agent stack](#).

Amplify. Telling the story so the org learns. Tooling that matters: writing skills, [the eval is the spec](#) practice, the [living changelog](#), agents that draft stakeholder updates from raw evidence. Falls through to the [PM as editor](#) workflow once you have a fleet doing the synthesis.

The mapping is rough on purpose. The same tool earns its keep at multiple stages. Memory matters everywhere. MCP servers feed Sense and Measure but also keep Discover honest. The point of the mapping is to tell you which categories to install first when you start building for a stage, not which to use exclusively.

The install order

Most PMs who try to build this stack make the same mistake. They install fifteen things over a weekend and then never use any of them because the cognitive overhead is too high to remember what each does. Here is the install order that actually works.

Week one. The base. Install Claude Code. Install the official anthropics/skills repo. Install the github-mcp-server so the agent can read your issues and pull requests. That's it. Use it for a week on real work. Do not install anything else until you have a feel for the base.

Week two. The first capability layer. Install [obra/superpowers](#) and try the brainstorm-spec-plan-TDD-review workflow on one real feature. Install one MCP server that connects to a system you actually use (postgres-mcp if you have a queryable DB, playwright-mcp if you want web automation). Install ccundo on day one so granular undo exists when you need it. The first time the agent does something destructive you will be grateful.

Week three. Memory and context. Install one memory tool (claude-mem is a good default). Install repomix so you can pack a codebase into a single context block when you need to. Pick one subagent collection (wshobson/agents is the default) and install three to five subagents that match your daily roles, not the whole catalog.

Week four. The PM-specific layer. This is where you start applying the trailing how-tos. Pick one of the three (Discover, Build, Measure) based on which stage of your work most needs the leverage right now. Walk through that post end to end. By the end of week four you have a working personal stack you actually use.

Months two and beyond. Add hooks for repeating workflows. Add slash commands for muscle-memory tasks. Read the guides. Subscribe to the awesome lists as feeds and check them monthly for new repos. Build your own skills when the existing ones don't fit your work.

This order is not arbitrary. The friction at each step compounds, and adding tools out of order is how PMs end up with stacks they don't trust.

Critical limitations of the bridge

The bridge has real limits. Naming them honestly is part of the deal. A chapter on a personal agent stack that does not list the limitations is selling, not teaching.

Single-PM scope. The personal stack only sees what one PM can give it access to. It cannot read Slack channels the PM is not in. It cannot see deals in the CRM the PM is not on. It cannot read meeting transcripts from meetings the PM did not attend. Cross-team patterns that span outside the PM's visible work are invisible to it. The enterprise brain solves this. The bridge does not.

No team-wide memory. Cross-session memory works for one PM's filesystem. It does not propagate across teammates. Each PM on the same team rebuilds the memory layer separately. There is no shared "what did we decide as a team" across the team without an enterprise platform.

Limited write access. The personal stack reads. The agent can be configured to write, but write actions outside the PM's own filesystem (sending email, posting in Slack, updating Notion) are fragile, require explicit per-tool setup, and have no audit trail by default. The enterprise version normalizes write actions across systems with auditing and approval flows. The bridge does not.

Security boundary is your laptop. Enterprise agents have audit logs, role-based access, retention policies, and data loss prevention. The personal stack inherits the security posture of the laptop and the source systems being connected. PII handling stays the PM's responsibility. Treat customer transcripts the way they should be treated today, not differently because an agent is reading them. Follow your company's data handling policy.

Tool drift. Open-source repos move fast. A repo that is hot today could be unmaintained in 18 months. The stack requires ongoing curation. Plan for it.

Cognitive overhead is real. Twelve installed tools is a lot to track. The first time the agent surprises a PM (in either direction), the trust drop costs a session of productivity. Add tools only when the friction the new tool solves is something you can name in a sentence.

Won't replace the data team. Or the security team. Or the legal team. The personal stack augments individual PM work. It does not refactor company governance, replace specialist functions, or solve organizational problems. Use it as a tool for a person, not as a substitute for an org.

If any of these limitations is a deal-breaker for your situation, this chapter is the wrong starting point. The right starting point is then to push your CTO or CIO toward the enterprise version directly. Read [when not to use AI](#) for the longer argument about real limits.

What stays human

Building this stack does not change what stays human. You still own the strategic call. You still decide which problems are worth solving and which are not. You still take responsibility for the outcomes the stack helps you ship. The agent is a force multiplier on the parts of PM work that are mechanical, not on the parts that require judgment, taste, and accountability. The [when not to use AI](#) chapter has the longer version of this argument.

What changes is the ratio. A stack like this lets a single PM operate at the throughput of a small team without losing the coherence of one person's vision. That's the prize.

What to do this week

Three concrete actions:

First, read [the overview post](#), which sets up the destination, the gap, and the bridge with concrete outcomes and limitations.

Second, pick one of the three trailing how-tos: [Discovery](#), [Build](#), or [Measure](#). Pick the one whose stage of PM work most needs help right now. Walk through it end to end.

Third, install Claude Code if you haven't, and run one real PM task through it tonight. Not a synthetic example. A real interview transcript, a real PR review, a real weekly digest. The point of the stack is to feel the leverage, and the only way to feel it is on real work.

The shared context layer is coming whether your company is ready or not. Build the personal version now. You'll be the person who already knows what to put on the enterprise version when it arrives. That is not a small thing. That is the difference between adopting AI-native ways of working and being adopted by them.

[Series complete](#) · [Loop back to Part 1](#)

Re-read: The PM Agent Stack overview

When you take this to a teammate, start them at Part 1. The overview is the framing piece (destination, gap, outcomes, limitations) that makes the recipes in Parts 2 to 4 land. Re-read or share.

→

Sources: [hesreallyhim/awesome-claude-code](#), [anthropics/claude-code](#), [anthropics/skills](#), [obra/superpowers](#), [github/github-mcp-server](#), [wshobson/agents](#), [yamadashy/repomix](#), [thedotmack/claude-mem](#), [crystaldba/postgres-mcp](#), [microsoft/playwright-mcp](#). The 18-category map is adapted from Divyanshi Sharma's Instagram carousel on the Claude ecosystem.

Direction Metrics for AI-Native Velocity

Outcomes lag by weeks. Direction moves with each iteration. The seven leading indicators that predict outcomes 4-8 weeks ahead, and the dual-cadence system.

Web version: <https://falkster.com/handbook/direction-metrics> · last updated 2026-06-12

The companion essay [Outcome Accountability Is a Luxury Good](#) makes the case. This chapter is the practice.

The short version

Direction metrics are leading indicators measured on the cadence of the work itself. For AI-native and agent products, outcome metrics like NRR or CSAT lag 4-12 weeks behind your changes. With teams iterating 10-20 times per week, outcomes cannot drive day-to-day decisions. Direction metrics close that gap. The seven leading indicators that predict outcomes 4-8 weeks ahead: eval pass rate, agent quality score, iteration count, design coherence, customer escalation rate, dispute rate on outcome billing, and latency at p95/p99. Run two measurement layers: the seven leading indicators reviewed daily and weekly, and outcome metrics (NRR, CSAT, NPS, expansion) reviewed monthly and quarterly. Prevent gaming by auditing correlation quarterly. If an indicator drops below 0.5 correlation with the outcome it predicts, replace it.

The two-layer measurement system

Layer	Cadence	Indicators	Drives
1 (Direction)	Daily / weekly	The seven leading indicators	Day-to-day decisions: what to ship, what to roll back, what to evaluate further
2 (Outcomes)	Monthly / quarterly	NRR, CSAT, NPS, expansion revenue, churn rate, customer satisfaction by cohort	Strategic decisions: do we keep investing, are we pricing right, is the buyer changing

Both layers are deliberate. Both reviewed in different meetings with different audiences.

The seven leading indicators

One section per indicator with definition, source, healthy band, what it predicts, lag, common pitfalls.

1. **Eval pass rate over the last 7 days.** Predicts: customer CSAT 4 weeks out.
2. **Agent quality score from sampled outputs.** Predicts: NPS on successor 6 weeks out.
3. **Iteration count and shipped changes.** Predicts: feature adoption 8 weeks out.
4. **Design coherence.** Predicts: customer trust 6 weeks out.
5. **Customer escalation rate.** Predicts: churn 12 weeks out.
6. **Dispute rate on outcome billing.** Predicts: NRR 8 weeks out.
7. **Latency at p95 and p99.** Predicts: retention 6 weeks out.

The Goodhart audit

Quarterly process. For each indicator, plot it against the outcome it predicts with the lag. Compute correlation. If above 0.7, healthy. If 0.5 to 0.7, weakening. If below 0.5, dead. Replace dead indicators.

What this changes about how PMs are measured

Two evaluation layers. PMs evaluated on the quality of their leading indicators (are they well-chosen, are they predicting outcomes, is the team's iteration cadence healthy). Plus the strategic outcome layer on annual or semi-annual cadence.

Outcome accountability moves from monthly to annual. Direction accountability becomes the day-to-day.

What to do this week

Build the indicator registry. YAML file. List the seven indicators (or your team's chosen seven), where the data lives, the healthy band, what each predicts. The agent that operationalizes this is at </blog/agent-direction-dashboard>.

Part 5: Lead the Shift

Scale yourself, your team, and your practice.

Your Weekly Playbook

What a week looks like when you run the full handbook. The week is the rhythm, the loop is same-day, and you run it as often as hot signals land.

Web version: <https://falkster.com/handbook/pm-as-team> · last updated 2026-07-22

The short version

The week is the rhythm. The loop is a day. Monday is a strategic reset: read the overnight signal brief, review outcomes, set one focus. Friday is reflection: show working prototypes, tune the agent fleet, capture what you learned. In between, you run the outcome-to-prototype loop as many times as clear signals warrant, and each run collapses into a single day. A listening agent surfaces the outcome a customer is reaching for in the morning, a prototyping agent turns it into something clickable by midday, a real customer touches it in the afternoon, and you decide, kill, iterate, or graduate, by end of day. The slower, high-judgment work, deep interviews about hard problems and trio brainstorming, is the weekly layer that the fast loop does not replace. Most of what used to fill your calendar, standups, sprint planning, backlog grooming, status meetings, quarterly planning, is gone. What replaces it is customer evidence, working prototypes, and outcome data, produced in days instead of quarters.

The capstone: putting it all together

You've learned about the [AI Product Operating Model](#). You understand [Continuous Discovery](#) and [Continuous Listening](#). You know how to [prototype fast](#) and validate ideas in hours, not quarters. You've set up the [Impact Loop](#) to measure outcomes instead of vanity metrics. And you've built AI agents that handle the work that used to bury you.

So what does a week actually look like when you're running all of it? This chapter is the connective tissue, where the frameworks become a coherent cadence.

But first, the shift that changes everything about the shape of the week.

The loop is a day, not a week

The old version of this chapter spread one loop across the week: interview on Tuesday, build on Wednesday, decide on Thursday. That was already faster than a quarter. It is still too slow, and it puts the loop on the wrong clock.

Here is why. In the old shape, Tuesday was “discovery day,” the day you went looking for a problem. But your listening agents already did that. Every support ticket, call, churn survey, and rage click was synthesized overnight, and a ranked set of customer outcomes is waiting in your Monday brief and refreshed every morning. You do not need a day of the week to find the signal. The signal is already on your desk.

So the loop no longer stretches across the week. It collapses into a day, and you run it whenever a clear outcome lands.

A same-day loop looks like this. Morning: the listening agent surfaces an outcome, not a feature request, that several customers are reaching for, with the evidence attached. You read it and make a call: is this clear enough to build against today? If yes, you dispatch it. By midday you have a working prototype, either from an AI coding tool or from the prototyping agent that built it off the outcome brief while you were in your first meeting. Early afternoon, you put it in front of an actual customer, not a colleague standing in for one, through an isolated preview link: “Here, try this. Does this solve your problem?” Their reaction is the validation. By end of day you decide: kill it, iterate on it tomorrow, or graduate it, which means handing engineering a working prototype to harden and an eval to gate it against, not a spec to interpret. Most days, the right call is to kill it, and that is the system working, not failing. You spent an afternoon, not a sprint.

You run this loop as many times in a week as there are clear signals worth chasing. Some weeks that is four times. Some weeks it is once, because the signals that landed were ambiguous and belong to the slower layer below.

The weekly layer: the work that shouldn’t be rushed

Not everything should run same-day. The fast loop is for clear signals with an obvious outcome. The rest, the ambiguous and the high-stakes, is the work the week protects.

Deep interviews about hard problems. The listening agents surface what customers are reaching for. They cannot tell you why a whole segment behaves against its own interest, or what a customer means by a word they keep using. That is a conversation, unscripted, 45 minutes, with the agent-prepped context in hand so you skip the surface and go straight to the problem behind the problem. Book two of these a week. They feed the ambiguous signals the fast loop cannot resolve.

Trio brainstorming. When an outcome has more than one plausible shape, you get your engineer and designer in a room with the transcript and the agent’s synthesis and you sketch. Nothing precious. Some ideas are bad. That is fine. You are choosing which shape is worth a same-day prototype next.

Relationships. An hour a week of genuine customer conversation that is not an interview and not a prototype test. You are building the trust that makes a customer honest with you instead of polite. No agent will ever do this, and it is where the signal that matters most first shows up.

The fast loop gives you speed. The weekly layer gives you judgment. Running everything same-day would flatten the hard problems into shallow ones. Running everything weekly would waste four days on a signal you could have resolved by lunch. You need both clocks.

Monday: set the one focus

Your week starts before you sit down. Your listening and monitoring agents ran overnight, and the brief is waiting: the outcomes customers are reaching for, ranked by impact, the metrics that moved, the anomaly worth a look, the competitive moves. You read it over coffee in eight minutes. This is not a weekend batch. It refreshes every morning. Monday is just when you use it to steer.

Then three things.

Review and reflect. Look at last week's outcomes dashboard. What graduated? What did you kill? What did you learn that changes how you think about the product? Fifteen minutes, data-informed, not process theater.

Set the week's focus. One thing. Not five. What is the single most important thing to figure out this week, the ambiguous problem worth the weekly layer's deep work? Write it down, send it to the team in a two-minute Slack post, make it concrete and falsifiable.

Update the artifacts. Your Opportunity Solution Tree is living. Your dashboards are fresh. Your agents have their marching orders. An auto-generated stakeholder update goes out: what graduated, what you learned, what you're focused on.

Ninety minutes. It replaces the sprint planning meeting, except you are steering with customer evidence and shipped results instead of estimates that slide by Friday.

Friday: align and grow

Friday is not for new work. It is for the work that compounds, and for reflection.

Morning: the meetings that need to be meetings. You show working prototypes and outcome data and ask for the decisions only humans with organizational power can make. You are not reporting on tickets. These meetings are short because you have something to show. You do not need 45 minutes to describe an idea. You need 10 to demo it and 20 to get a decision.

Midday: customer time. An hour of genuine conversation, the relationship layer, the part only you can do.

Afternoon: tune the fleet and reflect. Your agents worked all week. Review their output. Are they extracting the outcome or capturing the feature request? Are the dashboards accurate? Calibrate. Add an agent, retire one. Then sit with your reflection document: what surprised you, what invalidated an assumption, what you want to do differently. This is where the whole system pays off. Most PMs are so buried in process they never think. You have Friday afternoons because you automated everything that does not need human judgment, and because the loop that used to eat your week now takes a day.

What vanishes from your calendar

Let's be explicit about what this replaces.

Daily standups disappear. An agent posts a daily pulse to Slack, two or three bullets on what shipped, what's blocked, what's coming. Everyone has context without spending fifteen people's time.

Sprint planning becomes continuous. You are not committing to work in two-week chunks. You are deciding, day to day, which clear signal is worth a same-day loop.

Backlog grooming is gone. You do not have a backlog. You have a live queue of outcomes and experiments, and an eval bar that decides what graduates. See [The Anti-Backlog](#).

Status meetings are replaced with automated dashboards. Stakeholders see metrics, shipped work, and learnings any time. Nobody needs you to narrate what happened.

Lengthy planning cycles vanish. No quarterly roadmap where you estimate twenty things. Weekly learning loops that test assumptions and amplify what works.

This is not about working less. It is about where your attention goes: the problems that actually matter and won't get solved by process.

Why this makes you more valuable

Let's talk about career, since if you read this far you're thinking about it.

You have customer evidence for every recommendation. Not an opinion, a transcript and a prototype reaction. "I put this in front of eight customers today and six immediately saw the problem" is a different sentence than "I think we should do this."

You show working prototypes while others show mockups and decks. Leadership remembers the prototype they could click. They forget the slide.

You measure outcomes, not outputs, so you know which work moved the needle and which was beautiful and useless, and you can explain the ROI in terms the org cares about.

And you ship faster than anyone expects. Your unit is the day, not the quarter. People notice and start asking why. Then, because the grunt work is automated and the loop is a day, you have space to read, reflect, and make the connections others miss.

Executives promote people who have customer evidence, working systems, outcome data, fast iteration, and strategic insight. You have all five.

Transitioning gradually, the first month

You can't flip a switch. You're in an org with processes and skepticism. Ramp gradually.

Week 1: Turn on listening. Pipe your highest-volume signal source, probably support tickets, into an agent that surfaces the top customer outcomes every morning. Now the signal comes to you.

Week 2: Start weekly deep interviews. Two calls, agent-prepped, recorded with permission. Build the muscle for the weekly layer.

Week 3: Run the loop once, same-day. Take one clear outcome from the brief. Build a prototype in a few hours. Get it in front of a customer that day. Decide by end of day. Do it once and you'll feel the difference.

Week 4: Set up outcome tracking. Measure what you shipped. What metric did you expect to move, did it, by how much? One outcome-driven experiment under your belt.

By the end of month one you have listening, a weekly interview rhythm, one same-day loop run, and outcome data. Not the full playbook, but the foundation. Month two you add pieces. Month three it's normal.

Handling the objections

“My org doesn't support empowered teams.” You don't need permission to listen to customers, build a prototype in your own time, or measure outcomes. That is [the guerrilla version of this playbook](#), and it runs without a single org change. Start small, start solo, prove it. The [Empowered Teams Resistance playbook](#) covers the guerrilla approach.

“I don't have time for the loop.” You don't have time not to. Right now you're in meetings deciding features on incomplete information. Every loop you run is a decision made on a real

customer's reaction instead of a guess.

“My stakeholders want roadmaps.” Show them outcomes. “We shipped this and adoption rose 34% for power users” beats a roadmap. Leaders care about results more than plans.

“AI tools aren't approved here.” Start with what is approved and show the results, then make the case. You can run the spine of this playbook with a spreadsheet and one LLM if you have to. The tool isn't the point. The system is.

Start this week

You don't need everything perfect or your org's buy-in.

Pick one thing this week: turn on one listening source, book two deep interviews, or run the loop once, same-day, from a signal to a prototype to a customer to a decision by end of day.

Just one. Do it. Show it. By next week you'll know something you didn't today. By the end of the quarter, people will start asking why your product is shipping faster and landing better.

The playbook works because it respects how humans actually operate. Humans learn by doing, care about outcomes over activity, and need time to think. Build a system that protects those three things, put the fast loop underneath it, and everything else follows.

Strategy From Signals, Not Slides

The annual strategy deck is a memorial to a meeting. Run a one-page living strategy doc, updated weekly with the signals that could change your beliefs.

Web version: <https://falkster.com/handbook/strategy-from-signals> · last updated 2026-06-12

The short version

The annual strategy deck is a memorial to one Wednesday in February. By the time it is published it is already a fossil, and the re-plan cost is so high that teams execute against assumptions everyone knows are stale. What I run instead: a one-page living strategy doc with two sections. Section one is 5-10 beliefs about the world, each tagged with confidence level and date last reviewed. Section two is, for each belief, the specific signal that would falsify it. Updated every Monday in a 20-minute solo review. Most weeks the note to leadership says “nothing material changed.” Occasionally it says “belief X shifted, here is what that means for our bets.” The annual offsite does not set strategy anymore; it reviews the year’s evolution and identifies the least-confident beliefs. It would be easy to read this as a slower strategy deck. It is a different artifact, one that admits uncertainty instead of performing it.

Strategy is not a deck

For 20 years of my career, every company I worked at performed the same ritual: an annual leadership offsite, a strategy framework, a deck, a rollout, an org-wide email. The execution artifact that sits underneath this strategy layer is the [bet portfolio](#). The deck was the strategy. The strategy was the deck. Everyone aligned. Three months later half of it was dead and the other half was being quietly ignored. The deck still existed in a Drive folder, getting cited in board meetings as if it were operative.

This is the most expensive opinion-hoarding ritual in corporate life. The deck represents what a small group of senior people thought was true on one Wednesday in February. The world has changed every day since. The deck has not.

Strategy needs to update faster than that. The leaders may have been right at the moment. The problem is that the moment ended.

What I run instead

A one-page living strategy doc. Two sections. Updated weekly.

Section 1: What we believe.

A short list of statements about the world. Each one is a *belief*, not a fact. The belief should be consequential: if it's wrong, our actions should change. Each belief is tagged with how strongly we hold it (high, medium, low) and how recently it was last reviewed.

Examples: - "Our buyers will tolerate consumption pricing if they can see usage in real time. (High confidence; reviewed last week.)" - "Mid-market customers will replace incumbents if we can show 10x cost reduction with comparable quality. (Medium confidence; reviewed two weeks ago.)" - "Voice will be the dominant interface for our category within 18 months. (Low confidence; we're watching.)"

Five to ten beliefs. Not more. If you have 30, you don't have a strategy. You have a worldview document.

Section 2: The signals we're watching.

For each belief, the specific signals that would change it. Quantitative where possible, qualitative when not.

Examples: - "If quarterly billing complaints exceed X per Y customers, the consumption-pricing belief is in question." - "If our mid-market win rate against incumbents drops below Z, the cost-reduction belief is in question." - "If voice adoption in our analytics shows fewer than W percent of new sessions starting via voice within 6 months, the voice belief is in question."

The signal is the thing that could falsify the belief. If you can't articulate a signal that would falsify it, the belief is a faith statement dressed as strategy. Remove it.

That's the whole document. One page. Two sections. Updated every Monday.

What's broken about the deck-as-strategy

It rewards conviction theater. The deck must look certain to be respected, so leaders perform certainty. That certainty gets mistaken for accuracy by the rest of the org, who then execute against assumptions the leaders themselves would qualify if asked.

It mistakes the artifact for the practice. The deck is the output, and producing it consumes weeks. The actual strategy work, watching signals, updating beliefs, making bets, happens in the gaps between deck cycles, mostly invisibly.

It's unreviewable. When was the last time your company explicitly revisited a strategy claim and said "this turned out to be wrong"? If the answer is "we don't really do that," your strategy is being performed, not practiced.

And it's a slow artifact in a fast environment. In an AI-product market where competitive position can shift in two weeks (a new model release, a new entrant, a new pricing move), an annual strategy runs on a timescale that doesn't match the reality it claims to describe.

The Monday ritual

Once a week, the strategy doc gets reviewed. Not in a meeting. In a 20-minute solo session by whoever owns it (usually CEO or CPO, with active input from product leadership).

The review:

- For each belief, has the signal changed since last week?
- If yes, what does the new signal tell us?
- If new signal materially shifts the belief, update it. Note the date. Note what changed.
- For each new piece of signal arriving from listening systems, eval dashboards, customer interviews: does it touch any belief? If so, surface it.

The output is a one-paragraph “what changed in our strategy this week” note, sent to the leadership team. Most weeks it says “nothing material; here are the signals we’re watching.” Occasionally it says “we updated belief X based on Y; here’s what this means for our bets.”

This is the practice. Not an annual offsite. A 20-minute weekly review.

What the offsite becomes

Don't kill the offsite. Change what it's for.

The annual offsite, in the new model, isn't where strategy is *set*. Strategy is set continuously. The offsite is where the leadership team:

- Reviews the year's evolution of the strategy doc. What beliefs changed, what signals moved them, what bets followed.
- Identifies the beliefs they're least confident in and sets up the discovery work to test them in the coming quarter.
- Surfaces blind spots: areas of the business where signal isn't flowing yet and needs to be built.
- Aligns on the framing for external comms (the public version of the strategy is necessarily slower-moving than the internal one).

A much more useful offsite. Also faster. You can do it in a day because you're not generating strategy from scratch. You're reviewing and refining what's already alive.

What changes for the rest of the org

When strategy is a living doc, the org's relationship to it changes.

Sales stops citing the deck and starts citing the doc. They have access. They see when beliefs change. They're no longer six months behind on what the company actually thinks.

Marketing stops launching campaigns based on stale strategy claims and starts shipping in cadence with the doc.

Engineering can see why priorities shift. The strategy doc explains the *why* behind the [bet portfolio](#), which makes priority changes feel coherent rather than capricious.

Customers (the public version) get a more honest view of where the company is going. The customer-facing version is necessarily lighter, but it's grounded in the same beliefs and updated on the same cadence.

The whole org gets smarter, faster, because strategy stops being a quarterly broadcast and becomes a continuously-readable artifact.

What you'll lose

You'll lose the satisfaction of "publishing the strategy." That moment of completion (deck done, email sent, offsite over) is real and feels good.

You'll lose the political clarity of having one immutable strategy nobody can question. Living docs are messier. They invite challenge. People will push back on belief-level changes more than they push back on annual deck rollouts because changes are happening at a higher cadence.

You'll gain accuracy. You'll gain speed. You'll gain the credibility that comes from being the team that updated their bet last Tuesday because the data shifted on Monday, instead of the team still executing against an annual plan everyone in the room knows is stale.

The trade is worth it. The trade is mandatory in a market that updates daily.

Pick one thing this week

Don't blow up your company's strategy planning. Run the doc in parallel for one quarter and let the comparison make the case for you. The [continuous discovery](#) system is what feeds the signals this doc watches.

1. Open a one-page doc. Title it "What we believe and what we're watching."
2. Write 5 to 7 beliefs you (and your team) currently hold about your market, customer, product. Tag confidence.

3. For each belief, write one signal that would change it.
4. Set a calendar reminder for every Monday at 9am for a 20-minute review.
5. Run it for a quarter. At the end, compare it to your company's official strategy. Notice which one is more accurate. Notice which one has actually been updated.

Within two quarters the doc starts replacing the deck in real conversations. By the third quarter, even your CEO is referencing it. A strategy that doesn't update with signal isn't strategy. It's a memorial to a meeting.

The Anti-Backlog

The backlog is a graveyard pretending to be an inventory. Burn it. Replace with a live queue, signal-fed, capped at two weeks.

Web version: <https://falkster.com/handbook/the-anti-backlog> · last updated 2026-06-12

The short version

The backlog is a graveyard pretending to be an inventory. A 400-item Jira backlog is not a plan: it is institutional amnesia in tabular format. The anti-backlog replaces it with three things: a live queue capped at two weeks of work, signal-fed from customer signal clusters, eval regressions, incident action items, and cost spikes only; a hypothesis library for half-formed ideas written as hypotheses not features; and a kill list for decisions you have already made not to do something. Migration from a large backlog takes four weeks. Week 1: archive everything older than six months. Week 2: sort the rest by signal evidence. Week 3: stop adding to the old backlog. Week 4: delete it. The discomfort is highest in week 4. It passes. What replaces it is a team that ships in response to signal.

A confession

I have walked into every PM job I've held with a backlog of 200 to 2,000 items waiting for me. I have looked at every one of those backlogs, felt overwhelmed, and tried to “groom” my way through them. I have failed every time.

Backlog grooming is a ritual we adopted because we couldn't think of anything better to do with the inheritance. The PM who left the team three years ago wrote 40 percent of the items. The customers who asked for them have churned, changed their minds, or had the underlying need addressed differently. The items remain. We pretend they're real input.

A 400-item Jira backlog is not a plan. It is a graveyard pretending to be an inventory. This chapter is the demolition.

What's broken

It's a memory nobody reads. Items written years ago by people no longer at the company, requested by customers no longer at their company, on top of products that no longer exist in their original form. We treat it as institutional memory. It's the opposite. Institutional amnesia in tabular format.

It's a queue processed by no one. A queue is something you process. The backlog is processed by nobody. Items just live there. Calling it a “queue” is wishful thinking. Calling it a graveyard is the

diagnosis.

It rewards capture over judgment. PMs get praised for “writing it down” rather than for deciding what matters. The act of capturing feels like progress. It’s deferral. The PM who refuses to add to the backlog and instead says “we won’t do this” is doing better work than the PM who captures everything.

It distorts conversations. When a stakeholder asks “why aren’t we doing X,” the PM points to the backlog. “It’s there, just not prioritized.” That implies “yes, eventually” when the honest answer is “no, never.” The backlog is the lie that lets you avoid the no.

And it loses information faster than it gains it. The customer who asked for X six months ago is not a fixed signal. Their context evolved. Alternatives in the market changed. The job-to-be-done may have shifted. The backlog item is a fossil. The signal in this week’s call recording is the only thing that’s actually current.

The live queue replacement

What I run instead has these properties.

It’s small. Capped at two weeks of work for the team. If you can’t see all of it on one screen without scrolling, it’s too big. The discipline of a small queue forces real prioritization.

It’s signal-fed. Items enter from one of four sources only: top customer signal cluster (from continuous listening), eval regression, incident post-mortem action item, cost spike investigation. *No item enters because someone had an idea.* Ideas go elsewhere.

It has explicit slots. Each engineer’s coming two weeks has a fixed number of slots. Items map to slots. When slots are full, no more items go in. This is the part that breaks PMs used to “we’ll figure it out” mode.

And it empties. Items get done within their two-week window or they get kicked out. They don’t migrate to “next sprint.” They go back to the signal source. If signal still says they matter, they re-enter. If signal has faded, they were never as important as the original capture suggested.

Where ideas go (since not the backlog)

Three places.

The hypothesis library. Half-formed ideas get written as hypotheses, not features. “I think users would benefit from X because Y” instead of “Add feature X.” The library is searchable, taggable,

and used as input to the bet portfolio. It's not a queue. Nothing in it is "scheduled." It's reference material for the team's thinking.

The signal stream. The continuous listening system catches what customers actually say. If five customers say the same thing, it cluster-bubbles. You don't need to remember the one customer who said it three months ago. The system will resurface the pattern when it becomes one.

The kill list. Decisions you've already made *not* to do something. "We considered X. Here's why we won't do it. Here's what would have to be true for us to reconsider." This is the most underrated artifact a product team can keep. It prevents the same idea from being relitigated every quarter.

Three docs replace the backlog. None grows unboundedly. None pretends to be a queue.

How to migrate

You have a backlog right now with 200 to 2,000 items. Don't try to triage it. You will fail.

Do this instead.

Week 1: Archive everything older than 6 months. Single bulk move. To an "archive" status nobody looks at. If something matters, it'll come back through signal within a quarter. Almost never does.

Week 2: Sort the rest by signal evidence. Items with active customer signal in the last 30 days survive. Items without it get archived. Don't agonize. Signal-supported items are the ones that matter.

Week 3: Stop adding to the old backlog entirely. New items enter the live queue (if signal-fed) or the hypothesis library (if speculative). Old backlog freezes.

Week 4: Delete the old backlog. Or archive it to read-only. Team will feel anxious. Hold the line. Within two months they'll wonder how they tolerated it.

The discomfort is highest in week 4. It passes. What replaces it is a team that ships in response to signal rather than performing prioritization rituals against a graveyard.

The cultural fight

A stakeholder, sometimes the CEO, will find the backlog comforting. Killing it will feel cavalier to them.

The argument that lands: a 400-item backlog is not rigor. It's the appearance of rigor without the substance. The work that ships is not coming from the backlog. It's coming from this quarter's bets, this week's incidents, and the signal you saw on Monday. The backlog is theater.

If they push back, offer a compromise: keep the backlog read-only as an archive, but the team operates from the live queue. Track for a quarter what gets shipped from the queue versus the backlog. The data closes the conversation. The backlog won't produce a single shipped item that wasn't already a signal-driven priority.

Pick one thing this week

You're not going to delete the backlog Monday. Do this instead.

1. Open your current backlog. Sort by date created.
2. Filter to anything older than 12 months. Count the items. (You'll be surprised.)
3. Archive them. Bulk move. One action.
4. Tell your team you archived them. Watch what happens. (Spoiler: nothing.)
5. For the next two weeks, don't add anything to the backlog. If it's a real bet, it goes in the live queue (small, signal-fed). If it's a hypothesis, it goes in the library.

After two weeks, look at what you would have added to the backlog and didn't. How much actually mattered? Usually about 10 percent. The other 90 percent didn't survive even the smallest filter.

The backlog isn't memory. It's amnesia in tabular form. Burn it, and trust the signal to bring back what matters.

The signal stream that feeds the live queue comes from [Continuous Discovery on Autopilot](#). The blog companion with the full signal synthesis agent blueprint is [Kill the Feature Request Queue](#). For the meeting equivalent of this move, see [Kill the Status Meeting](#).

The Builder PM 30/60/90

The 90-day plan for making the shift from traditional PM to product builder. Done in order. In 90 days, you have a different job.

Web version: <https://falkster.com/handbook/builder-pm-30-60-90> · last updated 2026-06-12

The short version

The Builder PM 30/60/90 is the structured shift from traditional PM to product builder inside the job you already have. Days 1 to 30 are invisible infrastructure: Claude Code, a prompt repo, one automated agent, a personal signal dashboard, and three fewer recurring meetings. Days 31 to 60 ship one visible artifact that replaces old practice (an eval-as-spec, a live product page, or a 60-minute prototype). Days 61 to 90 make the change structural by documenting it, converting a peer, shifting one metric of success, and publicly killing one old artifact. You will feel like a fraud at day 1, exposed at day 20, threatened at day 45, and vindicated at day 70. Each is on schedule.

The bridge nobody gave me

Every PM on earth has read some version of “AI is changing the role, you need to become a product builder.” They nod. They agree. Then they go back to their job Monday morning and nothing changes because there’s no bridge between “I believe this” and “here’s what I do differently this week.”

I made the shift the hard way, over about 18 months, with a lot of false starts. This chapter is the version of the playbook I wish someone had given me then.

It assumes you have a full-time job, a team, a boss with expectations, and roughly zero permission to throw everything out. That’s the real starting condition. Most career advice pretends otherwise. This chapter is about transforming the job you already have; if you’re starting a new job, that’s [The PM 30/60/90](#). For the operating changes that accompany this shift (killing the status meeting, replacing the backlog with a live signal queue), see [Kill the Status Meeting](#) and [The Anti-Backlog](#).

The thesis

You don’t convert to Builder PM in a quarter by announcing a rebrand. You convert by replacing one mechanical piece of your old role each week with a signal-driven, prototype-first, agent-assisted version of it. Done in order. Most of it is invisible to your team for the first 30 days, on purpose. At 60 days, you ship one visible artifact. At 90 days, you make the change structural so it outlasts you.

Days 1 to 30: install the infrastructure

The goal of the first 30 days isn't to change anyone's opinion of you. It's to install the tools and rituals that make the next 60 days possible.

Week 1: Set up your builder stack.

- Get Claude Code (or equivalent) running. Ship one working prototype in your first week. Small is fine. A script. A mini tool. Something not for work. Just prove to yourself you can.
- Start a prompt repo, even if it's a folder in your personal notes. Commit to editing prompts only there.
- Pick one existing task in your week that's entirely mechanical (weekly status report, QBR prep, competitor scan) and prototype an agent to do it.

Week 2: Instrument your own product.

- Map your top 3 surfaces. For each, write down: what eval set exists today, what adoption data exists today, what cost data exists today. Most boxes will be empty. That's the work.
- Build a personal dashboard (Notion, Looker, whatever) with the signals you wish you had. If you don't have the data, write down what you'd need to get it.

Week 3: Run continuous discovery on autopilot.

- Feed your last 20 customer calls into an agent. Generate a synthesis. Notice what you learn that you missed live.
- Subscribe to every customer signal source you can access (support tickets, sales call transcripts, churn surveys). Route them to a daily digest. Read it every morning for a week.

Week 4: Cancel three meetings.

- Find three recurring meetings on your calendar that are pure status. Decline them. If asked, say "I'll read the notes and flag anything that needs discussion." This will feel bad. Do it anyway.
- Repurpose the time for prototyping, discovery, or eval work.

End of day 30: you have a working prototype muscle, a prompt repo, a personal dashboard, a customer signal digest every morning, and three fewer recurring meetings. Nobody has noticed you're transforming. That's on purpose.

Days 31 to 60: ship one builder artifact

The goal of the next 30 days is to ship one visible artifact that replaces old practice with new. The whole team sees it. Some people hate it. A few love it. Both reactions are data.

Pick one of these:

The eval-as-spec swap. Pick your next feature. Skip the PRD. Write the eval set instead. Share it with engineering as the spec. Run the feature against the eval through development. Ship with evals as the acceptance criteria. The doc is 30 to 150 labeled examples, not a 10-page spec.

The live product page. Build the single URL that replaces your weekly status meeting. Put eval scores, adoption, customer signal clusters, cost, incidents on it. Share it in the general channel on a Friday. Email the CEO. See what happens.

The 60-minute prototype. Take one feature debate that's been going for weeks. Spend an afternoon prototyping two versions of it in Claude Code. Show them in the next meeting. Watch the debate end.

Pick the one that fits your organization's current weather. Eval-as-spec works best in engineering-heavy cultures. The live product page works best in data-driven cultures. The 60-minute prototype works best in stuck, opinion-driven cultures.

Ship the artifact. Collect reactions. Note who liked it and who pushed back. That's your future org map.

Days 61 to 90: make it irreversible

At day 60 you have infrastructure in place and one visible change shipped. The risk is regression. The team drifts back to old patterns, the artifact becomes an oddity rather than the new default, and in six months you're back where you started.

The last 30 days are about making the change structural, not personal.

Week 10: Document the new practice.

- Write a short internal doc (2 to 3 pages) on how you now work. What you ship. What you stopped shipping. What the new artifacts are. Share with your manager first, then your team.
- Offer to present it at the next PM all-hands or team forum. Most orgs will welcome this. The ones that don't are telling you something about their appetite for change.

Week 11: Convert one peer.

- Find the PM most likely to adopt one piece of your new practice. Walk them through it. Help them install it. This doubles your political cover. It also starts the shift from "that weird thing Falk does" to "the new way we do things here."

Week 12: Shift one metric of success.

- In your next perf conversation, propose changing one OKR or goal to something signal-driven. Replace “ship X feature by Y date” with “reach eval score Z on surface W by Y date.” This is the move that makes the new practice survive your next promotion cycle. If goals you’re measured against reward the old work, you drift back to it no matter how committed you are.

Week 13: Kill one thing.

- The last move. Pick one artifact of the old role (monthly status deck, backlog review meeting, detailed PRD template) and publicly kill it. Replace it with the new artifact you shipped in days 31 to 60. Write a short note explaining why.

This is the move that closes the loop. You’ve installed new infrastructure. You’ve shipped a visible artifact. You’ve made it a team practice. Now you’ve explicitly ended one piece of the old world. There’s no going back.

What you’ll feel at each stage

Day 1: like a fraud, because you’re reading this as a promise about who you’ll become, and you’re not that person yet.

Day 20: exposed, because you canceled meetings and you’re noticing who noticed.

Day 45: threatened, because someone senior is going to say some version of “this isn’t how we do things here.”

Day 70: vindicated, because the live product page will get quoted in a board meeting by someone who wasn’t in the room when you built it.

Each of these is on schedule. Keep going.

The one thing not to do

Don’t announce the shift. Don’t send the all-hands email titled “I’m becoming a Builder PM.” Don’t post about it on LinkedIn mid-transition. The move is operational, not rhetorical. The artifacts do the announcing. The work is the evidence.

People who announce change before they’ve shipped it do not change. They collect the reputational pre-pay for work they’ll never do. Ship first. Write about it after day 90 if you want. Most builder PMs don’t, because by then they’re too busy shipping the next thing.

Pick one thing this week

You can’t do all 13 weeks at once. Pick Week 1.

1. Install [Claude Code](#) or [Cursor](#). Ship one working thing that is not a deck. Could be a script that summarizes your inbox. Could be a small internal tool. Could be a clickable prototype of something in your product.
2. Put it somewhere you can point to (“I built this this week”).
3. Notice that you felt something when you did it. That feeling is the muscle activating.

That’s it. Week 1. You don’t need permission. You don’t need a meeting. You don’t need a new title. You just need to ship one thing.

You can’t read your way into being a Builder PM. You ship your way into it, one replaced artifact at a time, starting Monday.

Hiring the Builder PM

Most PM hiring loops test skills the role no longer needs. Hire by what they can ship, not by how they talk about shipping.

Web version: <https://falkster.com/handbook/hiring-the-builder-pm> · last updated 2026-06-12

The short version

Hiring the Builder PM means testing the actual skill: can this person ship a working prototype in four hours? The old PM hiring loop tests presentation, frameworks, and behavioral polish, all skills that coaching and LLMs have made easy to fake. The new loop has four rounds: a builder task (real customer transcript turned into a working prototype with an eval set and cost estimate), a review session (walk through what you built and defend the trade-offs), a pairing session (iterate on a production prompt against an eval set live), and one culture question about a decision you got wrong. Fewer candidates pass. The ones who do are visibly better.

The loop that stopped working

Most PM hiring loops in 2026 are optimized for a role that no longer exists.

The loop was designed around 2018 PM skills: a case study, a prioritization exercise, a strategy framing, a behavioral round, a culture fit. Every candidate you interview has been through this loop at least ten times before and has been coached (by an LLM, by a bootcamp, by a mentor) on exactly how to pass it. Signal-to-noise is so degraded you can't tell the difference between a candidate who *can* do the work and a candidate who has memorized the shape of the answer.

The fix is to stop testing the wrong thing.

What I test for now

A two-hour take-home where the candidate turns a real customer transcript into a working prototype, with an eval set, a cost estimate, and a rollback plan. The eval set concept comes from [The Eval Is The Spec](#), which is also what you use to score their submission. Skip the case study. Skip the strategy framing. Skip the hypothetical prioritization. Those tests are optimized for PMs I'm no longer hiring.

If a candidate can ship in two hours, they can do the job. If they can't, no amount of case-study polish will save the role.

Why each old round fails now

The case study tests packaging, not thinking. You give 72 hours and a dataset, ask for a deck. You learn: can they make a pretty deck. You don't learn: can they build a product. The deck is a vestige of consulting culture that crept into PM hiring and never left.

The prioritization exercise tests taste in frameworks. Every candidate knows RICE, ICE, MoSCoW. They perform the ritual. You nod. You hire based on whether their framework matches yours. You're hiring cultural alignment, not capability.

The strategy round tests confidence. A 45-minute conversation on "how would you think about entering enterprise" rewards the candidate most fluent at reasoning aloud about strategy. Real skill. Not the skill you need when the job is shipping working product.

The behavioral round is fully exploited. Every candidate has an LLM-rehearsed answer to "tell me about a time you dealt with conflict." You can only distinguish candidates by whether their answers feel more or less rehearsed, which is a terrible signal.

All four rounds test: can this person behave like a PM. That was the right question when PMs behaved like PMs. Now you need: can this person build.

The new loop

Round 1: The Builder task (take-home, 2 to 4 hours).

Send a real customer transcript or support-ticket cluster from your product. Ask them to:

1. Identify the opportunity.
2. Build a working prototype of a solution, using AI tools of their choice.
3. Write an eval set of 20 input/output pairs that define "good" for this solution.
4. Estimate cost per action.
5. Write a one-page "how we'd ship this" plan with an explicit rollback condition.

Time-box: 4 hours, honor system. You'll learn more from what they can build in 4 hours than from what they can write about in 40.

What I'm looking for: - Did the prototype actually run? - Is the eval set thoughtful (includes failure modes, not just happy path)? - Does the cost estimate have real numbers (not "we'd optimize later")? - Does the rollback condition show operational maturity, or is it hand-waved? - Is the code readable? Can an engineer build from what they submitted?

What I'm ignoring: - Polish. Ugly prototype that works is a yes. - Framework name-dropping. Citing Torres, Teresa, Cagan is table stakes now, not a differentiator.

Round 2: The review session (60 minutes, live).

Candidate walks me through what they shipped. I ask:

- Why this solution versus three alternatives?
- What did the eval set miss?
- How would this break in production?
- If cost per action doubled overnight, what would you do?
- What part of this are you least sure about?

The last question is the most revealing. Candidates who can identify their own weakest assumption with specificity are the ones I want. Candidates who say “I’m confident in all of it” are telling me they haven’t yet learned what operational work feels like.

Round 3: The pairing session (90 minutes, live).

Take a prompt from your production system. With the candidate, try to break it. Then improve it. Then run the updated prompt against your eval set and watch the score change.

This round can’t be faked. Either the candidate knows how to iterate on a prompt with signal feedback, or they don’t. Their prior practice shows in the first ten minutes.

Round 4: The culture conversation (45 minutes).

The only round I keep from the old loop, stripped down. One question: *tell me about a product decision you got wrong, what the signal was that you got it wrong, and what you did about it.*

If the candidate can’t recall one, they haven’t shipped enough to know. If they describe the situation but not the signal, they don’t operate the way I need. If they describe both, I have my answer.

Anti-signals I used to miss

Things I used to read as positive that I now read as yellow flags:

- **The polished case study.** In 2026, polished means they spent their time on presentation instead of thinking. The PMs actually building are producing messier, more alive artifacts.
- **Confidence in frameworks.** “I use RICE” was a green flag in 2018. Now it’s “I still think in frameworks, not signals.”
- **“I launched a feature that reached 1M users.”** Great, what was the eval score? What’s the cost per action? If they can’t tell me, the launch’s success was accidental or cosmetic.
- **MBAs with zero shipped prototypes.** Not a dealbreaker. But the burden of proof is on them. Ask to see something they built. If they’ve never built anything, they’re not yet a Builder PM. They might become one. They aren’t one today.

Positive signals I weigh heavily

- **They’ve shipped something in the last 30 days, even small.** A script. A weekend prototype. An internal tool. The muscle is active.
- **They use Claude Code or equivalent as a daily tool.** Not as a buzzword. They can tell me what they shipped with it last week.
- **They talk about customers in the present tense.** “My customers are telling me X right now” versus “historically my customers wanted Y.” Living signal versus fossil signal.
- **They admit uncertainty with specificity.** “I don’t know if this will work because X” is the PM who’ll catch the regression. “I’m confident this will work” is the PM who won’t.

The loudest pushback

When you propose this loop internally: “we can’t hire this way because no one can pass.” That’s exactly the point.

The current loop is easy to pass because the skills it tests are widely practiced. The builder loop is hard to pass because the skills it tests are the ones you actually need and that are scarce right now. Yes, fewer candidates will pass. The ones who do will be dramatically better.

The loop will feel harsh in the first six months while your hiring manager muscle adjusts. It will feel normal in year two. You’ll be hiring into a team that ships at a completely different velocity than the team using the old loop.

Pick one thing this week

If you have an open PM req, swap one round of your current loop for a builder round. For the full picture of what you’re hiring into, see [Old PM vs Product Builder](#) and [the Builder PM job ladder](#). Don’t do the whole redesign. Just swap one.

1. Pick one candidate you already have scheduled. Replace the case study round with the 4-hour builder task.
2. When you review their submission, score them on: did it run, eval set quality, cost realism, rollback maturity.
3. Compare what you learned from this round to what you would have learned from the case study. Notice the difference.
4. If you hire the candidate, run the same swap on the next loop.

One loop at a time. Within two quarters your whole hiring process has shifted and your hires are visibly different. Hire by what they can ship, not by how they talk about shipping.

PM-as-Editor: Managing a Fleet of Agents

The 39-agent chapter told you what to deploy. This is the skill you need once they're running: great editing at volume.

Web version: <https://falkster.com/handbook/pm-as-editor> · last updated 2026-06-12

The short version

PM-as-Editor is the skill you need once your agent fleet is running. Most PMs either trust agent output blindly (broken telephone) or rewrite everything from scratch (worse than doing the work themselves). The skill that scales is editing: reading agent output the way a senior PM reads a team member's PRD, cutting what doesn't serve the purpose, shipping the 80% version instead of perfecting toward 100%. The operational framework is a four-tier trust ladder: Tier 1 ships without review, Tier 2 after a one-minute sanity check, Tier 3 after a real edit, Tier 4 the agent assists but you author. Every edit you make is training data for the prompt. Spend 60 seconds after each edit writing what you changed and why, then update the prompt. Do this weekly and the fleet gets sharper faster than any competitor can replicate.

The skill you need once the fleet is running

The [39-agent chapter](#) told you what agents to deploy. This chapter is about the skill you actually need once they're running: being a great editor.

Most PMs, faced with a pile of agent outputs, do one of two things. They trust the output blindly and ship it, which turns their inbox into a broken-telephone game. Or they rewrite every output from scratch, which is worse than doing the work themselves because now they're writing *and* paying the LLM bill. Neither scales.

The skill that scales is editing.

This is a different muscle from writing. It's the muscle senior PMs have always used on their own team's output (reviewing a PRD, redlining a memo, shaping a deck), now applied to non-human output at 10x the volume. The [PM as a Team of AI Agents](#) chapter explains how the fleet itself is structured. If you can't develop this muscle, the agent fleet won't make you more effective. It will just make you tired in a new way.

The delegation-versus-verification ladder

Not every agent output deserves the same scrutiny. A mature fleet has a tiered trust model.

Tier 1: Ships without review. You trust the agent completely. It runs on a schedule, produces an artifact, delivers it. Examples: weekly customer signal digest, daily red-flag scan, release notes from a commit log. Trust tier. No human in the loop.

Tier 2: Ships after a one-minute sanity check. Agent produces, you skim, you accept or reject. Not editing the content. Verifying the output looks reasonable before it leaves your hands. Examples: stakeholder update, win/loss summary, roadmap one-pager.

Tier 3: Ships after real edit. You take the agent's draft and edit it meaningfully. Shaping framing, sharpening language, cutting, restructuring. Examples: board memo, crisis comms, anything read by a person whose opinion of you matters specifically.

Tier 4: Agent assists, you author. Agent produces a starter, output is so high-stakes or nuanced that you write the final version yourself. Examples: performance conversation, strategy shift announcement, customer apology for a real screwup.

Every agent in your fleet has a tier. Tiering is a product decision. Get it wrong in one direction (too much trust) and you ship garbage. Get it wrong the other way (too much review) and the fleet stops saving time.

How to move an agent up the ladder

New agents start at Tier 3: meaningful human edit every time. Over a few weeks I track one thing. When I edit this agent's output, what am I consistently changing?

It usually turns out to be one of two things. Sometimes it's the prompt. I'm always changing tone, format, length, level of detail. So I fix the prompt, re-run, re-evaluate, and the agent moves up a tier. Sometimes it's a capability gap. The agent doesn't know about recent customer conversations, or internal metrics, or whatever context it needed. Give it access to the source, re-evaluate, and it moves up.

Target: every agent from Tier 3 to Tier 2 within a month. Tier 2 to Tier 1 within a quarter. If an agent has been at the same tier for six months, I've stopped iterating. I'm using it as a crutch, not developing it as a system.

The metric I actually watch

The best operational metric for an agent fleet is: how much time am I spending editing agent outputs, total, per week?

Climbing: fleet is bloated and I'm reviewing outputs I shouldn't be. Audit tiers. Move things up or kill them.

Shrinking: I'm trusting agents more. Check eval scores. Trust built on vibes is a risk. Trust built on eval data is progress.

Stable for weeks: I've stopped improving the fleet. Schedule an agent review with myself. Find the three agents whose outputs need the most editing. Fix them.

The editing skill itself

Great editors do a few things well. None are mystical.

They know what “good” looks like for this artifact. Before they start, they have a clear picture of the ideal output. Without it you can't edit, you can only shove the draft around. For PMs that means a shelf of exemplars: memos you admire, comms you'd copy, decks that worked. When you edit an agent's draft you're measuring distance from one of those, not reacting to the draft in a vacuum.

They cut more than they add. Non-editors pile on context, caveats, detail. Editors cut. Agent outputs are usually over-explained, redundant, over-hedged, one transition too many. Cut until the structure shows. Then cut again.

They keep asking what the artifact is for. Every artifact has a job: a decision to unblock, a customer to inform, a stakeholder to align. Is this line doing the job? If not, cut. If yes, sharpen.

And they ship imperfect. They know when the output is good enough and they send it. Perfectionism is how agent fleets get re-absorbed into the PM's workload. If the agent's draft is 80 percent as good as what I'd write myself and it took three minutes instead of three hours, I ship. The customer never feels the 20 percent gap. My calendar does.

What not to edit

There's a failure mode where PMs edit everything, including things that don't need editing, because editing feels like “doing the work.” That's a performance, not a practice. Questions that kill this pattern:

- Am I changing meaning, or rearranging words?
- Would a reader notice the difference between my edit and the original?
- Am I editing because the draft needs it, or because I need to feel useful?

If the answer to the third question is yes more than occasionally, I'm not editing. I'm resisting the loss of the old role. I notice it. I let it pass. I ship the draft.

The feedback loop that matters

Every edit I make on an agent's output is training data for the prompt.

After editing, I spend 60 seconds writing down: what did I change, why, and what prompt change would prevent me from having to make this edit next time?

After a week of this, I have a list of prompt improvements. I make them in a single session. Run the new prompt against the eval. Ship. Edit-time for that agent drops. Compound this weekly and the fleet gets sharper faster than competitors' teams can replicate.

The team version

When the whole team uses agent fleets, edit-time becomes organizational signal. The PM with the best-tuned fleet has lowest edit-time and highest output. That's the person who gets promoted, not the PM performing "thoroughness" by editing everything.

Make team edit-time visible. Not as surveillance. As a shared learning pool. When one PM figures out how to improve an agent, the whole team inherits the improvement.

Pick one thing this week

Pick the one agent output you edit most often. Do the 10-minute version of this practice.

1. Open the last three outputs from that agent. Copy them into a doc.
2. Redline them the way you normally would. Note every edit.
3. For each edit, write one sentence: "if the prompt said X, I wouldn't have needed to make this edit."
4. Rewrite the prompt with those changes.
5. Re-run the agent. Edit the new output. Measure whether your edit-time dropped.

Ten minutes of this replaces about an hour of editing per week. Do it once a month for every agent in your fleet and you're compounding. For the cadence that ties agent outputs to actual product outcomes, see [The Impact Loop](#). The PM job in 2026 is not to write more. It's to approve faster, with better calibration. If you're still writing everything yourself, you haven't promoted yourself into the new role.

The Cannibalization Decision Framework

Four diagnostic questions, three operating modes (sunset, refresh, split), the political map, and the seven-decision sequence for the AI inflection.

Web version: <https://falkster.com/handbook/cannibalization-decision-framework> · last updated 2026-06-12

The short version

The hardest CPO call during the AI inflection isn't whether to ship an agent product. It's whether to keep selling the SaaS product that pays for it. Four diagnostic questions route the decision. Can the legacy architecture support the successor's quality bar? Is the legacy customer base the right ICP for the successor? Can the company afford the gross margin trough (it compresses from 78-82% to 58-65% during the ramp)? Is the buyer the same person? Those answers route to one of three operating modes: sunset (replace the legacy on a published date), refresh (rebuild with AI capabilities under the same pricing), or split (run both product lines under one company). The political coalition takes 90 days to assemble across CEO, CFO, CRO, CCO, and yourself. The seven-decision sequence must run in order: sunset date, reorg, comp rewrite, board pre-sell, customer migration in cohort waves, one tracked metric, and post-sunset reorg planned six months out. Out-of-order kills the transition.

The hardest call a CPO makes during the AI inflection is whether to keep selling the SaaS product that ships the agent product's payroll. Shipping the agent product itself is the easy part.

This chapter is the decision framework I use. Four diagnostic questions, three operating modes, and the political map that turns a strategic call into an executable plan.

The companion essay [The Cannibalization Playbook](#) carries the rhetoric. This chapter carries the operating practice.

The four diagnostic questions

Run these in order. Each one routes the decision.

Question 1: Can the legacy architecture support the successor's quality bar?

Look at the agent-native product you would ship in twelve months if your current architecture were the constraint. Can you build it on top of the legacy? Can the legacy data model, latency profile, and security posture support the successor's quality bar?

If yes, refresh becomes possible. If no, the architecture itself is forcing a sunset, regardless of the customer math.

The mistake here is wishful thinking. Engineers will tell you the architecture can be extended. They're sometimes right and often telling you what they hope. Test the answer concretely: have a senior engineer write a one-page sketch of how the successor would ship on the legacy architecture. If the sketch requires three foundational rewrites, the architecture cannot support it. Sunset.

Question 2: Is the legacy customer base the right ICP for the successor?

The successor product has a target buyer. Maybe it's the same buyer as the legacy. Maybe it's a different role, a different department, a different size of company.

If the legacy customer base is the right ICP for the successor, sunset is operationally clean (you migrate them). If the legacy customer base is the wrong ICP, sunset means losing those customers as you sunset. Then the question becomes whether you can replace them faster than you lose them.

Most companies discover the buyer for the agent product is different from the buyer for the SaaS product. The SaaS product sold to a function. The agent product sells to operations or to a different function entirely. This routes toward split rather than sunset.

Question 3: Can the company afford the gross margin trough?

Gross margin will compress from 78-82% (mature SaaS) to 58-65% (early Service-as-Software during ramp), bottoming around month 12 and recovering toward 70-75% by month 24-30.

Compute: how much cash will the company burn at the trough's bottom, given current revenue trajectory? Is that sustainable? Does the board have the patience? Does fundraising need to happen before the trough?

If the answer is no, the call cannot be sunset; you don't have the runway. The call is split (legacy revenue defends the cash position) or refresh (AI is additive). Choose carefully and pre-sell to the board.

Question 4: Is the buyer the same person?

Same buyer means migration is feasible. Same buyer means a single relationship handles both products. Same buyer means CS and sales motion don't have to be cloned.

Different buyer means split. The two products will need different sales motions, different CS playbooks, different positioning. Trying to run both under one product team breaks one or both.

Routing to the three operating modes

Take the four answers and route.

Sunset when: architecture cannot extend (Q1 = no), customer base is right ICP (Q2 = yes), trough is affordable (Q3 = yes). The successor replaces the legacy on a published date.

Refresh when: architecture can extend (Q1 = yes), customer base is right ICP (Q2 = yes), buyer is the same (Q4 = yes), and AI improves the feature set without replacing the business model. The legacy product evolves; positioning and pricing remain stable.

Split when: customer base is wrong ICP for successor (Q2 = no), or buyer is different (Q4 = no), or trough not affordable without the legacy revenue (Q3 = no). Two product lines under one company.

Most CPOs I work with want the answer to be refresh because it's the least disruptive. In 2026 the architecture and pricing realities of agent products are pushing more decisions toward sunset than most CPOs are comfortable with. The honest test is the four questions. Comfort level doesn't get a vote.

The political map

Once the operating mode is set, the work is moving the coalition. Five seats.

CEO

Will support if the option-value math is clear and if the board is pre-sold. Will fight if the transition feels like CPO empire-building.

The argument that wins: walk in with the seven-decision sequence already structured. Make the option-value argument (the legacy has a known cash flow and a declining option value; the successor has higher uncertainty and much higher option value). Ask "do you want me to lead this or escalate to the board?" CEOs who hesitate are not going to back you in the trough.

CFO

Will support if the trough is bounded and modeled. Will fight if revenue becomes variable and the forecast goes from "predictable" to "depends on customer volume."

The argument that wins: model both scenarios in unit economics terms. Propose hybrid pricing for the successor (committed minimum + per-outcome overage) so there's still a forecast floor. Co-author the board narrative. The CFO has to be your closest partner.

CRO

Will fight hardest. Comp plans and renewal economics are the levers.

The argument that wins: outcome pricing is typically 1.5x-3x the SaaS price for the same customer. The CRO's quota math actually gets better. Rewrite comp in their favor for successor deals (legacy at 50% historical comp, successor at 150%). Make them the hero of the new motion. Hold the line on the 50% legacy comp; this is the lever.

CCO

Will fight if CS metrics are tied to seat-based usage. Will support if the CCO can see CS redesigned around outcome quality.

The argument that wins: CS becomes the team that owns outcome SLAs, dispute resolution, and customer migration. CS gets strategic again. Senior CS leaders get a real budget and a real mandate.

CPO (you)

Will fight yourself, because cannibalizing the product line you spent years building feels like an admission of failure.

The argument that wins yourself: the legacy was the right answer for its time. The successor is the right answer for the next time. Both can be true. Loyalty to a product is not a virtue when the world has changed.

The coalition takes 90 days to assemble. Most of the work is one-on-one conversations, not meetings. By the time the seven decisions hit a leadership meeting, the calls have been made.

The seven-decision sequence

Once the coalition is formed, the decisions come in this order. Out-of-order is the most common reason transitions fail.

1. **Sunset date.** Set 18-24 months out. Internal first, external later. The forcing function for everything downstream.
2. **Reorg.** Maintenance team owns the legacy with a clear runway. Successor team owns the agent product. Bridge engineers (5-10% of engineering) own integration points: data, auth, identity, infrastructure. Hard fences, no rotation.
3. **Comp rewrite.** Legacy at 50% historical comp. Successor at 150%. Renewal accelerators on legacy disappear. Migration accelerators on successor introduced.
4. **Board pre-sell.** 30-minute slot in the next quarterly review. Pre-read includes trough math. Establish the leading indicators that will be reported every quarter.

5. **Customer migration in cohort waves.** Wave 1 (months 6-9): top 20 strategic accounts, individual CPO+CRO calls. Wave 2 (months 9-12): mid-market, structured email + webinar + self-service. Wave 3 (months 12-18): long tail, public announcement, deadline-driven.
6. **One number tracked.** Percent of legacy revenue migrated. CEO sees it weekly. Board sees it quarterly. Every leadership conversation references it.
7. **Post-sunset reorg planned.** Six months before sunset, communicate transparently to the maintenance team: here are the roles that exist after sunset, here's how to interview, here's the timeline, here's the severance for those who don't transition.

What I have learned by getting this wrong

Three failures from earlier transitions, kept here so you can avoid them.

One: I waited too long to publish the sunset date internally. I thought I was protecting the legacy team by keeping the sunset informal. I was actually preserving their ambiguity, which kept them holding on, which kept the transition slow. Publishing the date is the kindness. The ambiguity was the cruelty.

Two: I let the comp plan get watered down. The first version was strong. The CRO pushed back. We compromised. The compromise meant salespeople kept selling the legacy because it still paid better in some scenarios. Three quarters of slow successor growth could be traced to that one comp decision.

Three: I tried to make the post-sunset reorg humane by being vague about it until late. Vagueness is not kindness. People want to know what their lives look like after the transition. Plan the reorg early, communicate it transparently, give people time.

What to do this week

If you suspect your product line needs the cannibalization decision, run the four diagnostic questions on a single page before Friday. Show the page to your CFO. Just the CFO. Ask them what would have to be true for the gross margin trough to be acceptable to the board.

Their answer is your next month of work.

The companies that win the AI inflection are the ones running the seven-decision sequence. The companies that lose are the ones still running the soft pivot.

The companion blog essay is [The Cannibalization Playbook](#). Download the [Cannibalization Decision Tree](#) worksheet and the [CPO Coalition Map](#) template.

Dual Transformation: Running Two Clocks

Two cadences, three talent categories, six CEO scoreboard numbers, three rituals. The operating model that runs legacy and successor cleanly.

Web version: <https://falkster.com/handbook/dual-transformation> · last updated 2026-06-12

The short version

Dual transformation is two cadences, three talent categories, six CEO scoreboard numbers, and three rituals inside one product organization. The legacy SaaS runs on a six-week cycle with quarterly OKRs and seat-based forecasting. The agent-native successor runs on a one-week cycle with daily eval reviews, weekly outcome cohorts, and monthly pricing experiments. The talent rule is hard fences: a maintenance team (5 to 15 percent of engineering), a successor team (60 to 75 percent), and bridge engineers (5 to 10 percent) who own data, auth, identity, and infrastructure. The six CEO numbers are legacy revenue, NRR, and gross margin alongside successor outcome volume, gross margin per outcome, and percent of legacy revenue migrated. The three rituals are the Friday Read-Out (15 minutes, both teams, surface mid-week conflicts), the Sunset Drumbeat (monthly memo, migration progress), and the Quarterly Reality Check (honest review of cadences, talent allocation, and sunset-date adherence). Trying to run both products on one clock kills the new product first.

The cannibalization decision is the strategic call. Dual transformation is what you actually run after the call.

This chapter is the operating model. Two cadences, three talent categories, six CEO scoreboard numbers, three rituals. The companion essay [The Dual Transformation Operating Model](#) carries the rhetoric. This chapter is the operating practice you can put on the wall.

The two clocks

The legacy clock (six-week cycle)

This is what the legacy product was probably already doing. Don't change it.

Week	Activity
1	Planning, release prep
2	Build
3	Build

Week	Activity
4	Build
5	Stabilize, customer pilot
6	Release, retro, customer health review

Quarterly: OKR planning, NRR reviews with sales, pricing review. Annual: strategy review, customer advisory board, comp plan.

The legacy product is paying payroll. Predictability is its job. Resist the temptation to “modernize” the legacy cadence. It works. Leave it.

The successor clock (one-week cycle)

This is dramatically tighter than the legacy cadence. It needs to be. Agent products are non-deterministic, inference economics move, and the buyer is still teaching you what they want.

Day	Activity
1 (morning)	Eval review (overnight runs, drift, regressions)
1-3	Build, with continuous deploy to staging
3	Outcome cohort review (last 7 days, dispute reviews)
4-5	Ship to production behind quality gates
5	Weekly outcome retro

Monthly: pricing experiments, agent department reviews, NPS on the successor. Quarterly: strategic review, gross margin review per outcome, hiring plan.

Cross-team rituals on alternating weeks

The danger of two clocks is that the teams stop talking. Fix: one architecture review, one security review, one infrastructure review, one customer escalation review per fortnight, alternating between the teams’ weeks.

Even weeks	Odd weeks
Legacy customer escalation	Agent customer escalation
Agent infrastructure review	Legacy architecture review

Both teams are in both rituals (with rotating leads), so context flows. Neither team has to be in two same-day standups.

The three talent categories

Hard fences. No rotation. This is the highest-leverage decision in dual transformation.

Maintenance team (5-15% of engineering)

Dedicated to the legacy product. Keep the lights on, fix critical bugs, support escalations, maintain infrastructure. Do not ship features. Mandate is stability and migration tooling, not feature velocity. Bonus rewards uptime, dispute-rate reduction, and migration tool adoption.

Successor team (60-75%)

Dedicated to the agent-native product. Build, evaluate, ship. Includes agent specialists, ML engineers, prompt engineers, design engineers, FDEs working with early customers. Bonus rewards customer outcomes, eval quality, shipping velocity.

Bridge engineers (5-10%)

Senior ICs who own integration points: shared data, auth, identity, infrastructure, observability. Report to CTO or VP Eng directly. Sit in both teams' rituals on alternating weeks. The only people writing code that affects both products. Bonus tied to migration percentage (so they care about successor success) and to legacy uptime (so they don't break payroll).

The rotation mistake

Quarterly rotation of engineers between legacy and successor sounds fair. It destroys both teams' momentum. Engineers in motion lose context on what they were building. Engineers staying lose teammates and slow down. Six months in, both products are slow and the team is exhausted.

Hard fences. People know which team they're on for the next 12-24 months. They can opt to move, but the move requires a clean cutover, not a partial allocation. Bridge engineers exist precisely so that rotation isn't necessary.

The CEO scoreboard

Six numbers, every week, side by side.

Legacy: - Revenue - NRR - Gross margin

Successor: - Outcome volume - Gross margin per outcome - Percent of legacy revenue migrated

The blended margin is calculated and shown but it's not the headline. The headline is the migration percentage.

Why this set? Each number is owned by a different person, and the contradictions between them surface the operational tensions that are otherwise invisible. NRR on legacy and migration

percentage will move in opposite directions if the transition is working. Gross margin per outcome will be low and gradually rising. Revenue on legacy will gently decline; outcome volume on successor will exponentially climb. The blended margin will dip into the trough and recover.

Show all six together every week. Don't average them.

The three rituals

Beyond cadences and talent rules, three rituals do most of the work in keeping the clocks from colliding.

Ritual 1: The Friday Read-Out

15 minutes, every Friday. Each team's lead presents three things: what shipped this week, what's blocked, what changed in customer signal. CEO and CPO are in the room. Everyone else is optional.

The point is not status reporting. The point is to surface mid-week conflicts before they harden into Monday all-hands disasters.

Ritual 2: The Sunset Drumbeat

Once a month, the CPO writes a short memo to the entire company titled "Sunset Drumbeat." Three sections: where we are on migration percentage, what's working, what's hard. Named call-outs (engineer who shipped a great migration tool, salesperson who closed a successor deal). Short, public, signed.

The point is keeping the sunset visible. Without this drumbeat, the legacy team and the successor team start to feel like separate companies.

Ritual 3: The Quarterly Reality Check

Once a quarter, the CPO does an honest review of the two clocks with the leadership team. Three questions:

- Are the cadences still right?
- Is the talent allocation still right?
- Are we still on the published sunset date?

The reality check is hard because it forces the team to admit when something isn't working. The cost of avoiding it is higher.

What I have learned by getting this wrong

Three failures, kept here so you don't repeat them.

One: I tried dual transformation without naming it as dual transformation. Internally we called it “evolving the product.” The team didn’t know they were on two clocks. Naming the model is a leadership move, not a labeling exercise. Tell the team. Show them the cadences. The clarity buys you four months of execution speed.

Two: I let bridge engineering be a side activity for senior people instead of their main job. It got staffed with rotating volunteers. Predictably, it became nobody’s job. Integration broke. Auth broke. Observability gaps showed up in customer escalations. Bridge engineering needs to be an explicit role with explicit owners and explicit calendar time.

Three: I underestimated the political weight of the legacy product team. They had been the company’s identity for years. Treating them as “maintenance” felt insulting, even when the work was honest. Fix that worked: I asked the legacy team’s senior engineers to own the migration tooling itself. They became the people who built the bridge customers crossed to get to the successor. That gave them a heroic story to tell about their last 18 months on the legacy product.

What to do this week

Take any one weekly ritual on your legacy product (sprint review, weekly customer health review). Write down its current cadence in plain English. Then imagine running the same ritual for an agent-native product where eval drift can happen daily. What would change in frequency, content, decision rights?

The gap between those two answers is your dual transformation problem in miniature. If you can’t reconcile the gap with one ritual, you cannot reconcile it for the whole product line on one clock.

Have the conversation with your CTO or VP Eng about hard fences vs. rotation. If you’ve been rotating engineers between products quarterly, the next 90 days are a window to stop. Set fences. Define bridge engineering. Publish the sunset date if it isn’t already published.

The two-clock model is not glamorous. It is the difference between a transition that works and one that quietly fails for six quarters before being declared a strategic pivot.

The companion blog essay is [The Dual Transformation Operating Model](#). Download the [Dual Transformation Operating Cadence](#) template. For the strategy that precedes this operating model, see [The Cannibalization Decision Framework](#).

Pricing Migration: The 18-Month Quarterly Playbook

Six quarters, four mandatory CFO/CRO/Board/Lead-customer conversations, the gross-margin recovery curve, and the contract terms that defend outcome billing.

Web version: <https://falkster.com/handbook/pricing-migration-18-month-playbook> · last updated 2026-06-12

The short version

This is the six-quarter operating playbook for migrating from per-seat to outcome-based pricing. Quarter 1: internal alignment, four mandatory conversations (CFO on the gross margin trough, CRO on comp plan, board on the trough narrative, lead customer on the pilot), and the contract template drafted. Quarter 2: hybrid pricing live for new customers, comp plan rewritten with legacy at 50% historical and successor at 150% ACV, lead customer becomes a public reference. Quarters 3 through 5: three-wave customer migration ending with the long-tail announcement and self-service tooling. Quarter 6: final migration push and post-sunset reorg. The gross margin trough bottoms at 58 to 65% in months 10 to 12 and recovers to 70 to 75% by month 24. Pre-sell the curve to the board in quarter zero. The five non-negotiable contract terms are unit definition, dispute window, arbitration mechanism, committed minimum, and price ceiling. The comp asymmetry is the lever: hold the 50% number even when the CRO pushes back.

The cannibalization decision frames the strategic call. Dual transformation runs the operating rhythm. This chapter runs the actual pricing migration: six quarters, quarter-by-quarter, with the conversations and contract terms that decide whether the transition works.

The companion essay [The Pricing Migration Sequence](#) carries the full rhetoric. This chapter carries the operating practice.

The eighteen-month sequence at a glance

Quarter	Months	Focus	Key Output
Q1	1-3	Internal alignment + lead pilot	Unit economics model, contract template, lead customer signed
Q2	4-6	Hybrid live for new customers	Comp plan rewritten, lead reference, board pre-sold

Quarter	Months	Focus	Key Output
Q3	7-9	Strategic account migration	Wave 1 done (top 20 accounts)
Q4	10-12	Mid-market migration, trough bottom	Wave 2 done, sunset announced
Q5	13-15	Long-tail migration, recovery starts	Wave 3 underway, outcome quality investments
Q6	16-18	Sunset prep + new operating normal	Maintenance team graduating, post-sunset reorg planned

The failure modes and key outputs for each quarter are below.

Quarter 1: Internal alignment

Invisible from outside the company. Four things must happen.

1. **Build the unit economics model** for each major SKU: per-seat baseline, per-outcome scenarios (conservative, expected, aggressive), gross margin per outcome including inference and escalation cost.
2. **Run the four conversations to first commitment.** CFO (trough math), CRO (comp plan in draft), Board (30-minute slot in next quarterly review), Lead Customer (strategic account willing to pilot).
3. **Pick the lead pilot SKU.** One workflow. Bounded, frequent, measurable.
4. **Draft the contract template** with general counsel. The five non-negotiable terms (below). Budget six weeks for three drafts.

Failure modes: CFO unconvinced and you proceed anyway. Lead pilot too narrow. Unit definition fuzzy.

Quarter 2: Hybrid live for new customers

The pilot is signed. The board has committed to the trough. The unit economics held up.

1. **Hybrid pricing live for new customers.** Platform fee plus per-outcome overage. Committed minimums protect the forecast.

2. **Comp plan rewrite published.** Legacy at 50% historical, successor at 150%, renewal accelerators on legacy gone, migration accelerators on successor introduced. The CRO will push back on 50%. Hold the line.
3. **Lead customer becomes a public reference.** Logo, quote, outcome numbers. This single reference will close 30% of enterprise deals in the next four quarters.
4. **CS retraining begins.** Dispute mechanism, new SLAs, outcome-quality dashboard.
5. **First public board acknowledgment of the trough.** Three slides: the change, the curve, the leading indicators. Establish the format that runs every quarter for 24 months.

Failure modes: comp plan watered down. Lead reference delayed by procurement. CS not retrained.

Quarter 3: Strategic account migration

This is where the transition starts to feel real. Most CPOs lose nerve here.

1. **Wave 1 migration: top 20 strategic accounts.** CPO, CRO, account team meet with each customer. 90 minutes plus follow-up. Sixty-day project.
2. **Legacy roadmap shrinks.** Engineering at 70/20/10 (successor / migration tooling / legacy maintenance). Legacy stops getting feature work.
3. **Migration tooling ships.** Self-service dashboard projecting outcome-pricing bills. One-click migration with committed minimum based on historical usage.
4. **Pricing experiments begin.** Inference costs are dropping faster than expected. Run two cohorts at different price points. The Jevons cliff is real.
5. **Trough visible on financials.** Blended gross margin around 65-68%. Down from 80% but not catastrophic. Stay calm publicly.

Failure modes: strategic account migration delegated to account managers. Legacy roadmap shrinks too fast. Sloppy pricing experiments.

Quarter 4: Mid-market and trough bottom

The hardest quarter. Trough at its deepest. Team tired. Board twitchy.

1. **Wave 2 migration: mid-market.** Structured email, webinar series, self-service tooling. 200-500 customers in a quarter.
2. **Legacy tier sunset date announced.** Internal first, then external. Sunset is 12 months out.

3. **Q4 board update.** Trough at bottom. Blended gross margin 58-62%. Show the recovery curve. Walk through what month 18 looks like. Maintain the narrative.
4. **CFO's seasonal forecast.** Outcome revenue is variable. CFO needs to model seasonality, cohort behavior, inference cost curves. FP&A tooling upgrades.
5. **Internal morale management.** Show leading indicators. Celebrate strategic account references. Run an internal Q&A.

Failure modes: the board panics. CFO models can't keep up. Internal morale cracks. CRO hints at slowing down.

Quarter 5: Long tail and recovery

Recovery starts. Outcome volume scales. Inference amortizes.

1. **Wave 3 migration: long tail.** Public announcement, deadline. Self-service only. Some churn is expected and acceptable.
2. **Outcome quality investments.** Better evals, better routing, better escalation handling. Each quality improvement compounds margin.
3. **Pricing experiments mature.** Decide: hold prices and capture margin, or pass savings and capture share. Document the decision.
4. **CS rebuilt around outcome quality.** CS owns outcome SLAs, dispute resolution, customer education.
5. **Q5 board update.** Trough past. Margin recovering toward 65%. Outcome revenue at 60%+ of total. Narrative shifts from "managing the trough" to "scaling the new model."

Failure modes: sloppy long-tail migration creates negative public reviews. Quality investments deprioritized for new feature work.

Quarter 6: Sunset and the new normal

Legacy has a published sunset six months out. Migration mostly complete. Successor is the company's primary revenue driver.

1. **Final migration push.** Personal outreach to remaining customers. Some migrate, some churn, some negotiate one-year extensions on legacy at premium pricing (accept these; they're cash on a known end date).

2. **Sunset prep.** Engineering builds sunset tooling: data export, account closure, final billing.
3. **Post-sunset reorg planning.** Maintenance team's roles end at sunset. Plan now. Communicate transparently in month 17 with sunset in month 21.
4. **Q6 board update.** Trough past. Margin at 70%+. Outcome revenue at 75%+ of total. NRR on agent product 130%+.
5. **The new operating normal.** Dual transformation collapses to a single operating model focused on the agent product.

The five non-negotiable contract terms

Without these, you have a billing dispute pipeline waiting to happen.

Unit definition

What counts as a “resolved ticket” or “qualified meeting” or “accepted code review.” Specific. Measurable. Defensible. The hardest work of the contract template.

Dispute window

How long the customer has to flag a disputed outcome. Common: 14-30 days from the billable event. Shorter than 14 days is unfair to the customer. Longer than 30 days creates billing chaos.

Arbitration mechanism

Who decides when there's disagreement. For most B2B products, internal review by a senior CS lead, then escalation to a customer success VP, then escalation to a third-party arbitrator if needed. Define the path before the first dispute.

Committed minimum

The floor under your forecast. Most enterprise customers will commit to 60-80% of their projected outcome volume as a minimum. This converts your variable revenue into something forecastable.

Price ceiling

So the customer isn't exposed to runaway volume costs. Common: 130-150% of the forecasted volume capped at the price tier. The ceiling protects the customer's CFO and makes procurement easier.

The gross margin recovery curve

Months	Gross margin range	Phase
1-3	78-82%	Baseline
4-6	73-77%	Trough begins
7-9	65-70%	Trough deepens
10-12	58-65%	Bottom
13-15	62-68%	Recovery starts
16-18	67-72%	Trough past
19-24	70-75%	New normal

Pre-sell this curve to the board in quarter zero. Reinforce every quarter.

The seven leading indicators

Track these alongside the curve. They predict outcomes 4-8 weeks ahead.

1. Outcome volume per customer (climbing).
2. Gross margin per outcome (climbing).
3. Percent of legacy revenue migrated (climbing toward 100%).
4. NPS on the successor (steady or climbing).
5. Lead customer reference expansion revenue (climbing).
6. Dispute rate (falling).
7. Customer migration to legacy churn ratio (climbing).

If three or more are flat or falling at the same time, the transition is in trouble. Run the quarterly reality check from the dual transformation chapter.

What to do this week

Pick the major SKU with the clearest outcome unit. Write down the unit in one sentence. Estimate per-customer outcome volume. Multiply by 5-15% of the customer's alternative cost. Compare to current per-seat spend.

The ratio is your starting point. Now ask: has the CFO seen this math? Has the CRO seen the comp implications? Has the board been pre-sold on the trough? Is there a lead customer for the pilot?

The answers tell you exactly what your next quarter looks like.

The companion blog essay is [The Pricing Migration Sequence](#). Download the [Pricing Migration Quarterly Tracker](#) and the [Margin Recovery Curve Model](#). For the pricing framework that underlies this migration, see [Pricing for AI Products](#) and [Per-Outcome Pricing](#).

The Investor and Board Narrative for AI-Era SaaS

Gross margin compression, NRR redefinition, the new rule of 40, the right comp set, and the four CFO sub-agreements that make the trough boring.

Web version: <https://falkster.com/handbook/investor-and-board-narrative> · last updated 2026-06-12

This chapter is the investor and board narrative for the AI-era SaaS transition. It builds on the cannibalization, dual transformation, and pricing migration chapters. The companion essay [The CFO Conversation](#) walks through the actual conversation. This chapter is the standing narrative.

The short version

The AI-era SaaS investor narrative has four components: gross margin compression (accepting 55-70% GM during ramp versus the 75-85% of mature SaaS in exchange for higher absolute revenue), NRR redefinition (classical seat-based NRR breaks for outcome pricing, so you report two numbers: committed minimum NRR and outcome volume growth), the new rule of 40 (rule of 35 during the trough, recovering to rule of 40 as unit economics mature), and the right comp set (transition-peer companies like Sierra, Intercom Fin, and HubSpot AI, not pure-SaaS comps that make the trough look catastrophic). The enabling move is co-authoring the narrative with the CFO across three working sessions before any board meeting, so both of you are presenting the same numbers with the same framing. The board needs to be retrained on the new metrics over two quarters, not surprised by them.

The four sub-agreements with the CFO

The CPO and CFO co-author the narrative. Four explicit agreements:

1. The trough math. The gross margin curve over 24 months, with assumptions documented.
2. The comp set. Transition-peer companies, not pure-SaaS comps. Updated as new comps go public.
3. The leading indicators. Seven metrics that go in every board deck for 24 months.
4. The narrative tone. Calm. Specific. Owns the trough. Leads with the leading indicators.

Detailed walk-through of each sub-agreement with the data and templates that make it concrete.

The gross margin story

The full investor narrative on margin:

- Mature SaaS: 75-85% GM. Anchored to “software license” pricing.
- Service-as-Software: 55-70% GM during ramp. Anchored to “work delivered” pricing.
- Why the trade-off is favorable: lower percentage applied to higher absolute revenue (because outcome pricing captures value the SaaS price was leaving on the table).
- The recovery curve: 70-75% GM at maturity as inference falls and volume scales.

The investor story: “We are accepting structurally lower gross margin percentages in exchange for materially higher absolute revenue and stronger customer alignment. The trade is right at our company’s stage.”

The NRR redefinition

Why classical NRR breaks for outcome-priced products:

- Classical NRR = (starting ARR + expansion - churn - downgrades) / starting ARR.
- With outcome pricing, “expansion” depends on customer volume, which is variable.
“Downgrades” can happen if the customer’s volume drops. NRR becomes noisier, not because the customer is unhealthy but because the metric is mismatched.

The replacement: outcome NRR (committed minimum NRR + outcome volume growth). Two numbers, both reported. Boards retrained over two quarters.

The new rule of 40

Why the rule of 40 needs adjustment:

- Classical: revenue growth + profit margin ≥ 40 .
- AI-era practical: rule of 35 during the trough (year 1-2), recovering to rule of 40+ at maturity (year 3+).

The investor story: “The rule of 40 still applies. The gross margin compression during the trough means the rule of 40 math is temporarily suppressed. It recovers as the unit economics mature.”

The standing board update structure

Four-section quarterly format:

1. Trough position (1 slide)
2. Leading indicators (1-2 slides)
3. What changed this quarter (1-2 slides)
4. What’s next (1 slide)

Plus the comp set slide. Plus the standing questions and answer shapes.

Detailed in [Board Narrative Slide Outline](#). The downloadable slide-by-slide template that implements this chapter, PowerPoint included, is [the board deck product section kit](#).

What to do this week

Schedule the first of three CFO working sessions if you haven't already. Bring the trough math.

The Pricing-Tier Sunset Playbook

How to end-of-life a revenue line: announcement timing, three-wave customer migration, comp plan rewrite, dispute mechanism at scale, post-sunset reorg.

Web version: <https://falkster.com/handbook/pricing-tier-sunset-playbook> · last updated 2026-06-12

The short version

A pricing-tier sunset is not feature deprecation. It is the end of a revenue line, and it carries different stakes, different politics, and different communication. The four-stage structure spans 18 to 24 months: internal preparation (months 0 to 6), strategic account migration via Wave 1 individual calls (months 6 to 9), mid-market and long-tail migration (months 9 to 18), and sunset day plus post-sunset reorg (months 18 to 24). The comp asymmetry is the lever: legacy at 50% historical comp, successor at 150% equivalent ACV. Five patterns kill most sunsets before they close: Wave 1 calls delegated to account managers, the comp plan watered down by the CRO, customer cohorts treated uniformly when edge cases need separate tracks, the dispute mechanism untested at scale, and sales reps quietly negotiating extensions. Audit these five first.

The cannibalization decision sets the strategic direction. The dual transformation operating model runs the rhythm. The pricing migration sequence walks the 18 months. This chapter is about the sunset itself, the day the legacy tier ends.

The four-stage sunset structure

Stage	Months from announcement	Activity
1. Internal preparation	0-6	Coalition, comp rewrite, internal alignment
2. Strategic account migration	6-9	Wave 1 cohort: top 20 accounts
3. Mid-market and long-tail migration	9-18	Waves 2 and 3
4. Sunset day + grace period + post-sunset	18-24	Final migration, sunset day, reorg

One section per stage with detailed week-by-week activity, owner, and decision rights.

The communication waves

The three waves with specific channels, timing, and escalation paths. Cross-references [Sunset Communication Template](#).

The comp plan rewrite

The four changes:

1. Legacy comp at 50% of historical.
2. Successor comp at 150% of equivalent ACV.
3. Legacy renewal accelerators removed.
4. Migration accelerators on successor introduced.

The CRO will fight the 50%. The CEO has to hold the line. This is the lever.

The post-sunset reorg

Plan six months before sunset. Communicate four months before. Three categories of maintenance team members:

1. Transition to bridge or successor roles (most).
2. Take severance with full transition support.
3. Stay on a small post-sunset support team for legacy customer data export and final billing.

The fairness story: every maintenance team member knew the runway from day one. The transition is graceful for those who want to move and respectful for those who don't.

The dispute mechanism at scale

The dispute mechanism that worked at 30 cases needs to work at 300. Three preparations:

1. Hire dispute analysts in advance of need.
2. Test the unit definition at 10x projected dispute volume.
3. Publish the dispute resolution timeline so customers know what to expect.

The companion field report </blog/field-report-killing-per-seat-tier> walks the failure modes.

What to do this week

If you're 12-18 months from sunset, audit the five common failure patterns. Have you committed to the comp asymmetry? Are Wave 1 calls scheduled with the CPO present? Is the dispute mechanism stress-tested? Is the post-sunset reorg planned?

For the quarter-by-quarter operating practice that precedes this chapter, see [Pricing Migration: The 18-Month Quarterly Playbook](#). For the strategic framing that makes this decision, see [the cannibalization decision framework](#).

The CPO 30/60/90

The operational week-by-week plan for a new CPO: four audits, a trust ledger, a coalition map, and a day-90 readout the board will quote. With templates.

Web version: <https://falkster.com/handbook/cpo-30-60-90> · last updated 2026-06-04

The 90 days are an audit, not a tour

Most first-90-days plans for a new CPO are politeness schedules. Meet everyone. Listen a lot. Find a quick win. Don't break anything. I have run that plan. It produces a leader who is well liked at day 90 and operating on borrowed conviction at day 180, steering with a model of the company assembled entirely from what people chose to tell them.

I have stepped into the product leadership seat several times now, at Commure, Crisis Text Line, SOCi, and Smartcat. The plan below is what survived. It treats the first 90 days as an audit with a deliverable, not a tour with a vibe. The deliverable is a day-90 readout your board will quote back to you for a year. Everything before it exists to make that document true.

This chapter is the operational version. For the argument about why the classic plan expired, see [The First 90 Days as an AI-Native CPO](#). For the scoreboard you inherit the moment you take the job, see [the CPO Mandate](#).

The short version

The CPO 30/60/90 is built around four audits and three artifacts. Days 1 to 30: run the reality audit, the money audit, the quality audit, and the decision audit, while logging every commitment you make in a trust ledger. Days 31 to 60: instrument what the audits exposed, build your coalition map, and ship one visible decision that signals the new bar. Days 61 to 90: write the kill list, place your first two or three bets, and deliver a day-90 readout structured as an SCQA memo to the exec team and board. Do not reorg, do not rewrite strategy in week 2, and do not spend the 90 days collecting opinions you cannot falsify. The templates for the ledger, the interview guide, and the readout are in [the CPO First-90 Kit](#).

Day zero: before you walk in

The highest-leverage hours of your first 90 days happen before day 1. Three moves.

Become a customer. Sign up for the product with a personal email. Hit the paywall. File a support ticket. Try to cancel. You will never again experience the product without the org's framing wrapped

around it. This is your only unanchored read. Take notes you can quote later.

Read the unguarded documents. The last four board decks, the last two postmortems, the public changelog, the pricing page history on the Wayback Machine. Decks tell you the story the company tells itself. Postmortems tell you the truth it admits under pressure. The gap between them is your first map of the org.

Write your priors down as falsifiable statements. Not “I think the team is strong” but “I believe the activation drop is a pricing problem, not an onboarding problem, and I expect the data to show X.” Ten of these. Date the file. The first 30 days are designed to break these statements, and you cannot notice your model updating if you never recorded the model. I learned this one late. It changed how I onboard more than anything else on this page.

Days 1 to 30: the four audits

The first month is four audits run in parallel, plus one discipline that protects your credibility while you run them.

The reality audit. Use the product on real tasks weekly, not as a demo. Sit in at least eight raw customer calls, unedited, before you read anyone’s synthesis. The synthesis is downstream of someone’s judgment about what mattered, and you do not yet know whose judgment to trust. Build your own read first. My [interview guide](#) works as well for this as it does for discovery.

The money audit. Get cost per outcome visible by workflow. Pull compute and agent spend out of the aggregate cloud bill and attribute it to the workflows that incur it. In every AI-era product org I have looked at, at least one workflow was quietly underwater and nobody owned the number. Finding it pays for your first quarter. The full method is in [Gross Margin Is Your Job Now](#).

The quality audit. List every production AI feature. For each, ask one question: what is the eval score, and which way is it trending? The answers sort into three piles. There is a number and a trend (rare, treasure these teams). There is a number, somewhere, probably (the actual state of most orgs). Silence. The size of the third pile is your real quality roadmap. See [The Eval Is the Spec](#) for what good looks like.

The decision audit. This is the one nobody runs. Pick the last ten significant product decisions: a killed feature, a pricing change, a big bet, a deprecation. For each, reconstruct how it actually got made. Who raised it, where it was debated, who decided, how long it took, what evidence was in the room. You are mapping the real operating model, which is never the one on the wiki. Most of what you will want to change at day 60 lives in this map, not in the roadmap.

The discipline: keep a trust ledger. From hour one, log every commitment you make. “I’ll look into that.” “Let’s revisit in two weeks.” “I’ll get you an answer.” What you said, to whom, by when, status. New executives bleed credibility through small dropped promises they never noticed making, and the people watching you in month one are counting. Review the ledger every Friday. Close everything or renegotiate it explicitly. Trust at day 90 is compound interest on this file. The template is in [the CPO First-90 Kit](#).

One thing you are not doing in month one: sharing conclusions. You are allowed observations and questions. The moment you broadcast a thesis, every subsequent conversation becomes a referendum on it, and your information supply gets curated to match. Hold the thesis. Collect the evidence. The deep dive on this month is [the CPO listening tour, rebuilt as a ledger](#).

Days 31 to 60: instrument and align

Month two converts the audits into instruments and the conversations into a map.

Stand up the instruments. Whatever the money and quality audits exposed, make it permanently visible. Cost per outcome by workflow on a page. Eval scores with trend lines on every production AI feature. A direction metric that says whether the product is getting better, not just bigger. This is the difference between knowing the number once and leading with it. [Strategy From Signals](#) covers what to instrument; [Direction Metrics](#) covers how to choose the headline number.

Build the coalition map. By day 45 you know who runs on evidence, who runs on narrative, who is exhausted, who is dangerous, and who has been waiting years for someone to fix the thing they care about. Write it down as an actual map: allies, persuadables, blockers, and what each one needs to move. Your first bets will live or die on this map, not on their merits. I wrote the long version in [the CPO coalition map essay](#).

Ship one visible decision. Not a quick win. Quick wins are usually small enough to be ignored and cosmetic enough to be resented. Ship a decision that signals the new bar. Kill one zombie initiative everyone privately agrees is dead. Publish the eval scores nobody wanted on a wall. Replace the status meeting with a [decision-forcing review](#). The point is not the object, it is the precedent: decisions here now get made on evidence, in the open, and they stick.

Renegotiate the scoreboard. Somewhere in month two, your CEO’s patience for “still learning” runs out, usually unannounced. Get ahead of it. Bring a draft of what you want to be measured on for the next year, anchored in the audit findings. Margin-aware, eval-backed, outcome-shaped. This conversation is much easier to have at day 50 with evidence than at day 100 with excuses. The mechanics are in [days 31 to 60, instrumented](#).

Days 61 to 90: the kill list, the bets, the readout

Month three is where you spend the credibility you banked.

Write the kill list. Every product org carries initiatives, ceremonies, and features that survive on inertia. The audits told you which ones. List them, with what each costs and what killing it frees up. You will not kill them all by day 90. You will kill two, publicly, with a written explanation. The rest go in the readout as commitments. The craft of killing well is in [The Deprecation Playbook](#) and [the killed-feature playbook](#).

Place two or three bets, and name the anti-bets. Your first bets should be few, evidence-backed, and reversible-aware: know which ones you can walk back and which you cannot, and weight the irreversible ones accordingly. Just as important, name what you are explicitly not doing. An anti-bet list is the cheapest alignment tool that exists, because it converts a hundred future hallway debates into one paragraph. [The Anti-Backlog](#) is the standing version of this practice.

Deliver the day-90 readout. One memo, SCQA-shaped, five sections: where the product actually is, the two or three constraints that matter, the bets and anti-bets, the kill list, and the scoreboard you want to be judged on. Written to be read, not presented. Walk the exec team through it, then the board. This document is the actual deliverable of your first 90 days, and it is the artifact people will quote back to you for a year, so write it like you mean every sentence. Structure and template are in [the strategy memo template](#) and the board-facing version in [Investor and Board Narrative](#). The full month is broken down in [days 61 to 90, first bets](#).

What not to do

Do not reorg before day 90. You would be restructuring a system you do not understand using a model assembled from interviews with people who each had an agenda. Two exceptions: an integrity or safety problem, and the leader everyone is waiting for you to deal with, where every week of delay reads as endorsement. Otherwise decide the operating model first and let structure follow. [The Product Operating Model](#) is the decision that comes before any org chart.

Do not rewrite the strategy in week 2. Whatever document you produce that early is your priors wearing the company's letterhead.

Do not run the listening tour as theater. Forty meetings that produce a word cloud is a month of evidence spoiled. Run it as a ledger with structured questions and falsifiable claims, or do not bother.

Do not skip the trust ledger because it feels bureaucratic. It takes four minutes a day. It is the cheapest insurance you will ever buy on your own reputation.

The throughline

Everything above serves one shift. The classic CPO arrived to optimize an execution machine, because execution was scarce. Execution is cheap now. What is scarce is an accurate model of reality, instruments that keep it accurate, and the judgment to bet on it. The first 90 days are where you either build that or substitute charm for it. Charm decays. Instruments compound.

Pick one thing this week

If you are about to start the role: write your ten priors as falsifiable statements and date the file. It costs an evening and it will reshape your entire first quarter.

If you are already in the seat, whatever the day count: start the trust ledger today, backfilled with every open commitment you can remember. Then ask the eval question of your most important AI feature and watch what the org does with it.

The templates for everything in this chapter, the interview guide, the trust ledger, the coalition map worksheet, and the day-90 readout skeleton, are in [the CPO First-90 Kit](#).

Sources: [Michael Watkins, The First 90 Days](#) for the classic transition frame this chapter argues with, [Barbara Minto, The Pyramid Principle](#) for the SCQA structure of the readout, [Marty Cagan / SVPG](#) on product leadership transitions.

The Skill Stack: What PMs and CPOs Must Learn Now

The 12 skills that decide PM and CPO careers now, organized in five layers from judgment to storytelling, with a self-test and learning path for each.

Web version: <https://falkster.com/handbook/skill-stack> · last updated 2026-06-04

The stack moved

For twenty years the PM skill stack was stable: write specs, run rituals, coordinate execution, communicate up. Coordination was the scarce thing, so people who coordinated well got promoted. That stack quietly stopped paying. Agents absorbed the mechanical layer, [the translator function died with it](#), and the value moved in two directions at once: up into judgment, and out into expression.

This chapter is the map I wish I could hand every PM and CPO I have worked with: twelve skills in five layers, each with what it is, why it matters now, a falsifiable self-test, and the fastest acquisition path I know, on this site and off it. It is a long chapter. It is meant to be returned to, not consumed.

The short version

The skill stack has five layers. Judgment: problem selection, decision quality, taste. Expression: storytelling, writing for machines, prototyping. Systems: evals, agent orchestration, signal architecture. Economics: unit economics, pricing. Leadership: coalition building and killing things well. IC PMs live in expression and systems while building judgment; CPOs are paid for judgment, economics, and leadership. The two highest-ROI skills to start with are evals (rarest, most employable) and storytelling (multiplies everything else). Every skill below has a self-test; run them all, and spend the next two quarters on your two worst scores, not your two favorites.

Layer 1: Judgment

The layer agents cannot reach, and the one everything else exists to serve.

1. Problem selection

Choosing what deserves to be built at all, when building is no longer the expensive part. The cost of being wrong did not fall with the cost of execution; it rose, because you can now be wrong faster and at greater scale. I wrote about this in [the cost of being wrong](#).

Self-test: list the last five things your team shipped. For each, can you state the evidence that made it the best available use of the team at the time? Not the justification, the evidence.

Acquire it: run a real opportunity solution tree ([Your First OST](#)), then read Teresa Torres, [Continuous Discovery Habits](#), and practice the weekly cadence in [the weekly discovery playbook](#).

2. Decision quality

Treating decisions as bets with odds, keeping score, and separating decision quality from outcome quality. Most product orgs have zero memory of their own decisions, which means zero learning loop. The deep dive is [judgment reps](#), and the instrument is [the decision log](#).

Self-test: can you produce your last ten significant decisions, with the evidence present at the time and what happened after? If the answer lives in nobody's file, your org learns by anecdote.

Acquire it: keep a decision log for one quarter. Read Annie Duke, [Thinking in Bets](#). Run one [premortem](#) (Gary Klein's prospective hindsight method) on your next big bet.

3. Taste

The trained eye for what is good before the metrics confirm it. When everyone can generate ten plausible versions of anything in an hour, choosing well among abundance becomes the differentiator. I argued in [taste is the last moat](#) that this is now the scarcest input in product work.

Self-test: put three AI-generated versions of your next feature copy or flow side by side and rank them with written reasons. Then check whether a designer or your best customer ranks them the same way.

Acquire it: volume plus feedback. Critique sessions with designers, building your own [60-minute prototypes](#) until the difference between fine and good is felt, and studying products you admire one screen at a time.

Layer 2: Expression

Judgment that cannot be transmitted does not exist, organizationally speaking.

4. Storytelling

Not presentations. The structuring of evidence into an argument that survives retelling when you are not in the room. SCQA, the Minto pyramid, working backwards from the press release. This is the single most undertrained skill in product management and the full deep dive is [storytelling is a PM core skill](#).

Self-test: hand your last strategy memo to someone outside your team. An hour later, ask them to repeat the argument. If what comes back is the topic but not the argument, you wrote a report, not a story.

Acquire it: Barbara Minto, [The Pyramid Principle](#) (the book; the consulting world runs on it). Wes Kao's [Executive Communication course on Maven](#) and her [frameworks on Lenny's podcast](#). Bryar and Carr, [Working Backwards](#), for the PR/FAQ discipline, then write one with [the template](#). On this site: [show, don't tell](#) and [the board narrative chapter](#).

5. Writing for machines

Half your readers are now agents: the agents that draft your team's code, the AI overviews that decide whether customers find you, the fleet that runs your own workflows. Writing specs, evals, and prompts that machines execute faithfully is a distinct craft from writing for humans, with its own failure modes. Deep dive in [writing for machines](#).

Self-test: take your last PRD or prompt and run it through a model cold, no verbal context. Does what comes back match your intent? The gap is your score.

Acquire it: [Prompt Ops](#) on this site, [Anthropic's prompt engineering documentation](#), and the discipline of [the eval as spec](#), which forces precision no prose spec ever forced.

6. Prototyping

The argument you can touch. A working prototype ends debates that decks prolong, and it is now a two-hour skill, not an engineering request. [Prototype Before You Spec](#) is the foundational chapter.

Self-test: could you put a working prototype of your current top idea in front of a customer this week, yourself, without filing a ticket?

Acquire it: install Claude Code or equivalent and ship one small real thing this week, per [the Builder PM Week 1](#). Then use [the mob prototyping pattern](#) to spread it to your team.

Layer 3: Systems

The layer where PM work became infrastructure.

7. Evals

Defining quality as labeled examples and measurable criteria, then holding the product to them continuously. For anything AI-touched, the eval is the spec, the acceptance test, and the early-warning system in one artifact. This is currently the rarest skill on the stack and the fastest route to being indispensable.

Self-test: what is the eval score on your most important AI feature, and which way is it trending? If you cannot answer today, neither can your org.

Acquire it: [The Eval Is the Spec](#), then build [the five-row eval](#) this week. Outside: Hamel Husain and Shreya Shankar's [AI Evals course](#) and [Anthropic's evaluation guide](#).

8. Agent orchestration

Designing, briefing, and quality-controlling a fleet of agents that do the mechanical layer of your job. The skill is not prompting, it is delegation architecture: what to hand off, what to keep, and how to verify. The editor's stance is covered in [PM as Editor](#) and the org-level view in [PM as a Team of AI Agents](#).

Self-test: how many hours of your last week were mechanical work an agent could have done? If you do not know the number, the answer is too many.

Acquire it: stand up three agents from [the fleet](#), run them for a month, and tune them with [the agent tuning playbook](#). Outside: [Anthropic's agent-building guides](#) and Simon Willison's [weblog](#) for the practitioner's running commentary.

9. Signal architecture

Designing where truth enters your building: which customer signals get captured, synthesized, and routed to whom, continuously and mostly automatically. Discovery stops being a ritual and becomes plumbing. The foundation is [Continuous Discovery on Autopilot](#) and [Continuous Listening](#).

Self-test: when a customer told your company something important last Tuesday, what happened to it? Trace one real signal end to end. The number of manual hops is your architecture's debt.

Acquire it: build [the call triage pipeline](#) first, then [the discovery agent stack](#).

Layer 4: Economics

The layer that got promoted from finance's problem to yours.

10. Unit economics

Knowing what your product costs to run per outcome, and what moves that number. AI products carry real marginal cost, which means PMs now own a P&L line whether anyone told them or not. [Gross Margin Is Your Job Now](#) is the chapter; [financial fluency for PMs](#) is the skill-building deep dive.

Self-test: what is the gross margin on your product or feature area, and what are its two biggest drivers? If your answer is "that's finance's number," that is the gap.

Acquire it: the deep dive above, then Berman and Knight, [Financial Intelligence](#) (the standard text for non-finance managers), then one working session with your finance partner where you rebuild your product's cost per outcome together. Practice the conversation in [the CFO margin scenario](#).

11. Pricing

The highest-leverage decision most PMs never touch. AI broke per-seat logic, and pricing migrations are now product work, not pricing-committee work. Start with [Pricing for AI Products](#) and the [18-month migration playbook](#).

Self-test: can you articulate why your product is priced the way it is, what value metric it should be priced on, and what evidence supports the gap between the two?

Acquire it: the chapters above, plus April Dunford's [Obviously Awesome](#) for the positioning layer underneath pricing, and Madhavan Ramanujam's [Monetizing Innovation](#) for the willingness-to-pay discipline.

Layer 5: Leadership

The layer that decides whether any of the above scales past you.

12. Coalition building and killing things well

Two skills that are one skill: moving an org. Evidence does not implement itself, and nothing tests a leader like retiring something people are attached to. The coalition mechanics are in [the coalition map essay](#); the retirement craft is in [The Deprecation Playbook](#) and [The Anti-Backlog](#).

Self-test: name the last thing you killed publicly, with a written explanation, that someone senior wanted kept. If nothing comes to mind, you have been adding, not leading.

Acquire it: there is no course. Run the [CPO 30/60/90](#) coalition map at whatever scale you operate, kill one zombie initiative this quarter using the playbook, and study the org-shape essays: [the new org chart](#) and [hiring the Builder PM](#).

How to use this chapter

Run all twelve self-tests in one sitting. Score yourself honestly: have it, building it, don't have it. Then spend the next two quarters on your two lowest scores, not your two favorites. The favorites are favorites because you already have them.

For IC PMs: evals and storytelling first, almost always. For new CPOs: the [CPO 30/60/90](#) will surface your gaps faster than introspection will; most arrive missing economics or the killing skill.

One warning. The stack is not a curriculum to complete, it is a portfolio to rebalance forever. The 2021 stack looked permanent too.

Pick one thing this week

Run the twelve self-tests tonight. Write the three-column score in your notes. Pick the worst one and book two hours this week on its acquisition path. That is the whole move. The compounding does the rest.

Sources: [Barbara Minto](#), [Wes Kao on Maven](#), [Teresa Torres](#), [Product Talk](#), [Bryar & Carr](#), [Working Backwards](#), [Annie Duke](#), [Thinking in Bets](#), [Gary Klein on premortems](#), [HBR](#), [Hamel Husain & Shreya Shankar](#), [AI Evals](#), [Anthropic documentation](#), [Simon Willison](#), [April Dunford](#).

Part 6: Forward Deployed

The engineer who owns the outcome.

The Forward Deployed Engineer Is a Product Builder in Disguise

Palantir invented the FDE because software shipped without its engineer does not get adopted. It is the Product Builder, from the services side.

Web version: <https://falkster.com/handbook/fde-is-the-product-builder-in-disguise> · last updated 2026-09-17

Every few years the industry rediscovers the same fact under a new name: software that ships without the person who built it does not get adopted. Palantir learned it around 2008, selling data integration into agencies that could not operationalize what they bought. Their answer was to stop shipping software and start shipping engineers. They called the embedded ones Deltas, and for most of the company's first decade it employed more of them than it did product engineers.

In 2026 the role is everywhere. OpenAI raised more than \$4 billion for a company whose product is deployment. Anthropic launched a \$1.5 billion joint venture for the same job. Google Cloud, Databricks, Salesforce, and nearly every startup selling agents has a job listing for it. It is, by most accounts, the hottest engineering role in the industry.

I think it is the Product Builder, arriving from the other side of the building.

The short version

A forward deployed engineer is an engineer employed by a vendor, embedded inside a customer, and accountable for the outcome the software produces there. Palantir popularized the role around 2008 and ran four vintages of it, each adding responsibility without shedding the last: platform stability, then data integration, then custom solutions, then enablement. The thread that survived every vintage is customer accountability. The role exploded in 2026 because code got cheap and outcome pricing arrived, and both make the person who guarantees the result the scarcest seat in the company. That seat requires the same six skills this handbook attributes to the Product Builder. The FDE got there from services. The Product Builder got there from product. They are the same job with different badges.

Where the role came from

Nabeel Qureshi, who spent years as a Palantir FDE and wrote the most cited account of the experience, describes the split plainly. FDEs embedded with the customer three or four days a week and wrote code that got the job done, technical debt and all. Product development engineers back at

headquarters took what the FDEs built and generalized it. His line for the division: your job was to solve the problem and not worry about overfitting; PD's job was to take whatever you built and generalize it.

That loop produced the product. Palantir's data ingestion tooling, its visualization layer, and its app builder all started as things FDEs hacked together on site for one customer and PD later turned into Foundry components. The customer paid for the first version. Every later customer got it cheaper.

Natalie Meurer, five years a Palantir FDE and now running agent engineering at Sierra, breaks the role's history into four vintages. 2008 to 2012 was platform stability, getting the software to run on the customer's boxes. 2012 to 2016 was data integration and ontology modeling. 2016 to 2020 was building custom solutions on top. 2020 onward added customer enablement. Her observation is that these were not eras that replaced each other. Each one stacked on the last, so the total job kept growing.

What did not change across any of it, in her words: every single forward deployed engineer is accountable to the customer.

Why it is exploding now

Two forces arrived at once, and they point at the same seat.

The first is that code got cheap. Meurer's framing is that when a prompt to a coding agent produces a working solution, the marginal cost of software collapses and what remains valuable is translating an ambiguous customer problem into the right build, deploying it reliably, and guaranteeing it works. That was always the FDE's comparative advantage. It used to be buried under plumbing. The plumbing is now done by an agent, so the advantage is the whole job.

The second is outcome pricing. Sierra bills per resolved case. Salesforce put a work-completion metric on its earnings slide. I have written about [why Sierra is growing so fast](#) and the answer is an operating model, not a model: price the outcome, own the last mile, productize what the last mile teaches you. Meurer's version is sharper. Outcome pricing requires the seller to guarantee the outcome, and the question of how you guarantee it has one answer. That is forward deployed engineering.

So the vendors are not adopting the FDE because Palantir made it fashionable. They are adopting it because their pricing model has a guarantor-shaped hole in it, and the FDE is the shape.

The same six skills

Now hold the role up against the [Product Builder OS](#), which defines the AI-era product role as six skills: rapid prototyping, customer proximity, AI fluency, outcomes thinking, storytelling and

distribution, and end-to-end ownership.

An FDE prototypes on site, in the customer's environment, from the first week. Customer proximity is not a practice they schedule, it is where they sit. AI fluency is the job description at every lab hiring for the role. Outcomes thinking is the pricing model. Storytelling is how a Delta gets a skeptical operations director to change a workflow that has run the same way for twelve years, and Palantir paired every Delta with an Echo, a deployment strategist, because the politics of adoption are half the work. End-to-end ownership is the one thing Meurer says survived every vintage.

Six for six.

The difference is where each role came from. The Product Builder is what happens when a PM starts shipping because the cost of building fell. The FDE is what happens when a services engineer starts owning outcomes because the cost of building fell. Same collapse, two directions, one seat. I made the product-side version of this argument in [Old PM vs Product Builder](#). This wave is the services-side version, and I think reading both is the fastest way to see that the role is not a job title. It is what one person can own once building is cheap.

What this wave covers

The chapters that follow take the role apart. [Where the FDE starts and the sales engineer stops](#), because the confusion between them is expensive. How to make the transition from PM, engineer, or SE. What has to be true in the product before an FDE can do anything but bill hours. The loop that turns deployment into roadmap. The economics, including where the model stops paying. And how to build and run the team.

The argument underneath all of it is the one I keep making on [AI Product Management](#): the job did not get harder. The parts we trained people for got cheap, and the parts nobody trained for are what is left. The FDE is the clearest evidence I have found that this is true on the services side too.

Start this week

If your company sells AI into enterprises, go find the person who is accountable when a deployment does not produce the outcome the customer bought. Not the person who is blamed. The person whose job it is.

If you cannot name them, you have found the hole your pricing model is about to fall into, and you have found what this wave is for.

Sources: [Nabeel Qureshi, Reflections on Palantir](#) · [Natalie Meurer, The Dirty Secret of Forward Deployed Engineering, AI Engineer 2026](#) · [Plank, What Is a Forward-Deployed Engineer](#) · [MarkTechPost on the OpenAI Deployment Company and Anthropic joint venture \(May 2026\)](#)

FDE vs Sales Engineer: The Job Starts Where the Contract Is Signed

A sales engineer's job ends when the customer says yes. An FDE's job begins there. Confusing the two is how companies get expensive demos and no adoption.

Web version: <https://falkster.com/handbook/fde-vs-sales-engineer> · last updated 2026-09-17

The most expensive confusion in enterprise AI right now is between two people who look identical in an org chart. Both are technical. Both spend their week with customers. Both can build a demo that makes a CIO lean forward. One of them is done when the CIO signs. The other one has just started.

If you hire the first when you needed the second, you get a company with a beautiful pipeline and a graveyard of pilots. If you hire the second and manage them like the first, they quit. This chapter is the line between them, and it is a single line.

The short version

A sales engineer's accountability ends at the signature. A forward deployed engineer's accountability begins there. Every other difference, on-site time, integration depth, ownership of code, follows from that one. Solutions architects design the fit and hand off the build. Consultants deliver recommendations and leave. Professional services implements a fixed spec for a fee. Developer relations scales across thousands of users through docs. The FDE goes deep into one customer's production system and is measured on whether the outcome the customer bought actually shows up. The test for any of these titles is not what they are called. It is who is on the hook after go-live.

The one question

Ask this about any customer-facing technical role: when the customer's outcome does not show up, whose number goes red?

A sales engineer's number is bookings. Once the deal closes, the outcome is somebody else's problem, and the incentive structure is honest about it. That is fine. Pre-sales is real work and the best SEs are extraordinary at it. It is just a different job.

A forward deployed engineer's number is the outcome. At Sierra that is literally the invoice: the customer pays per resolved case, so an engineer who wires the integration badly does not get a bad review, they get a smaller bill. At Palantir it was renewal and expansion, which in a company that for

years ran almost no traditional sales force meant the FDE was the sales motion. Either way, when the outcome does not show up, the FDE is the one holding it.

That difference explains everything else people list. FDEs are on site more because you cannot own an outcome from a demo environment. They write production code because the outcome lives in production. They stay after go-live because the outcome is measured after go-live. None of those are the definition. They are consequences.

Five neighbors, one test

Plank's guide to the role has a useful map of the adjacent titles, and it is worth walking through with the test in hand.

The solutions engineer, or sales engineer, supports the sale. Demos, proofs of concept, security questionnaires, the architecture slide. Their job is to get to yes. At yes, they hand off. Test result: accountable up to the signature.

The solutions architect designs how the product fits the customer's stack. Good ones prevent a year of pain. They typically do not build what they design, and they are rarely measured on whether the design produced the business result. Test result: accountable for the design, not the outcome.

The management consultant delivers a deck. Sometimes a very good deck. The recommendation is the deliverable, and the client's execution of it is the client's problem. Test result: accountable for the analysis.

Professional services implements. The scope was fixed at contract, the hours are billed against it, and a change to the scope is a change order. Test result: accountable for delivering the spec, which is not the same as the outcome, and everyone who has run an implementation knows how far apart those two can be.

Developer relations scales one-to-many. Docs, samples, talks, community. Excellent at reaching a thousand developers. Structurally unable to go deep into one customer's messy production system. Test result: accountable for reach.

The forward deployed engineer sits after the signature, builds in production, and is measured on the result. Test result: accountable for the outcome. Everything else on the list is a real job. Only this one passes.

Why the confusion is so expensive

Two failure modes, both common.

The first is hiring SEs and calling them FDEs. It happens because the two look alike on a resume and because SEs are easier to find. The company ends up with excellent pilots and no production. Every pilot ends with a signed contract and a customer who cannot get the thing to work on their own data, and nobody whose job it is to fix that. The MIT NANDA figure that 95 percent of enterprise generative AI pilots show no measurable business impact is, in my read, mostly this failure mode at industrial scale. The models work. Nobody owned the outcome.

The second is hiring FDEs and managing them as pre-sales. Their calendar fills with demos. Their compensation tilts toward bookings. They are pulled off a deployment at the hard part to close the next one. The customer whose outcome they own gets an engineer for two weeks and a ticket queue after that. The best FDEs leave, because the thing they signed up for is the part after the signature, and the company keeps taking it away from them.

Both failures come from not asking the one question. Meurer's satirical FDE job posting, eight years as a staff engineer plus six years of direct sales plus four years of solutions architecture, is funny because it is what happens when a company wants the accountability of an FDE and the pipeline of an SE in one salary.

Titles are drifting, the test is not

Titles are already unreliable. OpenAI has used customer engineer and forward deployed engineer. AWS calls a pre-sales role solutions architect. Sierra calls its version agent engineer, and Meurer, who coined that, has since said the whole cluster of titles is converging on one function: engineering that is accountable to the customer. Decagon's framing is that FDE simply is product engineering with the same bar and reporting line.

That convergence is the point of this wave. Titles will keep moving. Who is on the hook after go-live is what you are actually hiring for, and it is what the customer is actually buying. I would put the question in the job description, in the comp plan, and in the first interview.

The next chapter covers [making the transition](#) from any of the five neighbors into the role, because the technical distance is short and the accountability distance is not. And if you are wondering what the product has to look like before an FDE can own anything, that is [what has to change in the product](#) and it comes before hiring, not after.

This wave sits in the running argument on [AI Product Management](#), which is that the role is defined by what one person can own once building is cheap.

Start this week

Pull the last three deployments that stalled. For each one, write down the name of the person who was accountable for the outcome after the signature. Not the account owner. Not the implementation lead. The person whose scorecard moved when the outcome did not.

If the same name appears three times, you have an FDE and should call them one. If no name appears, you have found why the deployments stalled.

Sources: [Plank, What Is a Forward-Deployed Engineer](#) · [Natalie Meurer, The Dirty Secret of Forward Deployed Engineering](#), [AI Engineer 2026](#) · [MIT NANDA, State of AI in Business 2025](#), via [MarkTechPost](#)

Becoming an FDE: The Transition From PM, Engineer, or SE

Three doors lead into the FDE role. Each one arrives with two of the three things the job needs and is missing the third. A 90-day plan for each.

Web version: <https://falkster.com/handbook/fde-transition> · last updated 2026-09-17

Three doors lead into forward deployed engineering. A product manager walks in with the customer and the outcome but not the production build. An engineer walks in with the build and the discipline but not the customer. A sales engineer walks in with the customer and the technical range but has never been on the hook after the signature.

Each door arrives with two of the three things the job needs. The transition is the third thing, and it is different for each path. What is the same for all three is that the third thing cannot be learned in a classroom, because it is a change in what you are accountable for, not a change in what you know.

The short version

The FDE role needs three things: production engineering, customer proximity, and accountability for the outcome. PMs have the second and third and need to close the engineering gap, which a coding agent has made a quarter's work rather than a career's. Engineers have the first and often the discipline behind the third, and need to move their desk to the customer's building. Sales engineers have the first two and need to stay past the signature, which sounds trivial and is the hardest of the three because everything in their compensation and calendar pulls the other way. In every case the 90-day plan is one real deployment, owned end to end, measured on the customer's result, with someone senior shadowing the first one. The role is senior by nature. Only 3 percent of open FDE roles in mid-2026 were early-career.

Door one: from product management

If you have been reading this handbook, you have already done most of the work. The [Product Builder](#) ships prototypes, sits with customers, and writes the eval before the code. What a PM usually lacks for forward deployment is the last mile of production engineering: authentication against the customer's identity provider, the batch job that runs at 2 a.m., the retry logic when the customer's API times out under real load.

Two years ago that gap was the whole distance. It is not anymore. A coding agent writes the retry logic. What it cannot do is know that the customer's API times out because their nightly ETL locks

the table, and that the fix is a schedule change nobody will approve without a conversation with the data team's lead. That knowledge is customer proximity, and you have it.

The 90 days: take one deployment that is already sold. Weeks one to four, pair with an engineer and build the integration yourself, with them reviewing and you typing. Weeks five to eight, own a second integration alone, with the engineer available but not driving. Weeks nine to twelve, run the go-live and the first month after, and be measured on the customer's outcome, not on the integration shipping. If at the end of that you can defend the production system to the customer's engineers without a translator, you are through the door.

The trap: treating the deployment as discovery. PMs are trained to extract requirements and hand them off. An FDE extracts requirements and then builds the thing, and the handoff you are used to does not exist. If you catch yourself writing a document for someone else to implement, you are still a PM.

Door two: from software engineering

Engineers have the build. Most good ones also have the production discipline, the instinct for observability and rollback that this handbook's [ship with observability](#) chapter tries to teach PMs. What they usually lack is the customer, in the specific sense Nabeel Qureshi describes from Palantir: sitting in the customer's building three or four days a week until you have the tacit knowledge of how they work, not the flattened list of requirements.

The 90 days: pick one customer and move in. Not a kickoff and a weekly call. Three days a week on site, or the remote equivalent of it, which is being in their Slack and their standup rather than in yours. Weeks one to four, learn the vocabulary and the org chart, including the informal one, and build nothing you were not asked for. Weeks five to eight, ship the first thing, small, in their production, and watch what happens when real users touch it. Weeks nine to twelve, own the outcome metric and report on it to the customer, not to your manager.

The trap: overbuilding. Qureshi's line about the FDE job was solve the problem and do not worry about overfitting. Engineers want to generalize. That is the product team's job, and the [deployment-to-product loop](#) covers how what you build gets generalized by someone else. Your job is the customer's outcome this quarter. If you find yourself building a framework, stop.

Door three: from sales engineering

The SE transition looks shortest and is hardest. SEs have the technical range and the customer fluency. Meurer's joke that FDE job postings ask for eight years as a staff engineer plus six years of direct sales is a description of a senior SE. What SEs have never done is stay.

Everything about the SE role ends at the signature. The comp plan pays on bookings. The calendar refills with the next prospect the day a deal closes. The demo environment is clean by design, because a clean environment is what sells. An FDE lives in the dirty one.

The 90 days: hand your pipeline to someone else, in writing, and take one signed deployment through go-live and 60 days past it. Weeks one to four, do the integration on the customer's real data and discover everything the demo hid. Weeks five to eight, ship to production and sit with the first users. Weeks nine to twelve, own the outcome number. Your compensation for the quarter should be tied to that number and to nothing else, or the transition will not hold, because the pull of the pipeline is stronger than intent.

The trap: the demo reflex. When something breaks in production, an SE's instinct is to show the customer what it looks like when it works. An FDE's instinct is to fix it. The first instinct built your career. It will end this one.

What all three have in common

The transition is complete when a customer calls you before they call your account manager. Not because you are friendlier. Because they have concluded you are the person accountable for their result, and that is the entire job.

Until then, whichever door you came through, you are a vendor's engineer who visits.

Senior people make this transition. Plank's July 2026 look at open FDE roles found 3 percent early-career, and that matches what the job demands: enough standing to tell a customer that their process is the problem, and enough range to fix it anyway. If you are early in your career and want this, the path is to become excellent at one of the three doors first.

The chapter on [building the FDE team](#) covers the hiring side of the same three doors, and the standing argument on [AI Product Management](#) is why all three are converging on one seat.

Start this week

Whichever door you are standing in, write down the third thing. For a PM: the last production system you shipped alone. For an engineer: the last week you spent in a customer's building. For an SE: the last outcome you were measured on after a signature.

If the honest answer is never, that is the gap, and the 90 days above is how to close it.

Sources: [Nabeel Qureshi, Reflections on Palantir](#) · [Natalie Meurer, The Dirty Secret of Forward Deployed Engineering, AI Engineer 2026](#) · [Plank, What Is a Forward-Deployed Engineer \(July 2026 role data\)](#)

What Has to Change in the Product Before FDEs Can Work

*Sending engineers into customers without changing the product turns them into a consulting firm.
Five things the product must expose before the first FDE lands.*

Web version: <https://falkster.com/handbook/product-changes-for-fde> · last updated 2026-09-17

There is a version of the forward deployed model that is just a consulting firm with a software company's cost structure. The company hires excellent engineers, sends them into customers, and each one builds something clever and bespoke. Two years later the company maintains forty custom systems, the product has learned nothing from any of them, and gross margin looks like Accenture's.

The difference between that and Palantir is not the quality of the engineers. It is that Palantir had a product underneath the engineers that was built to absorb what they made. Kevin Bai, who did the role at Palantir and now works on it at Anthropic, calls this the platform prerequisite. I would put it more bluntly. If you send FDEs into customers before changing the product, you have not adopted a deployment model. You have started a services business by accident.

The short version

Five things have to be true in the product before the first FDE lands. There must be a configuration boundary, a deliberate line between what an FDE can change per customer without code and what needs the product team, and most customer-specific work has to land on the configuration side. There must be a per-customer sandbox with guardrails, so the FDE builds outcomes instead of environments. There must be an eval harness, because the executable definition of the outcome is the contract the FDE works to and the thing outcome pricing bills against. There must be an observation record per customer, because an FDE who cannot explain why the number moved cannot own it. And there must be a feedback path from deployment to roadmap that has an owner, or the loop that made Palantir's model compound never closes. All five are product work. None of them are the FDE's job to build.

One: the configuration boundary

Every deployment has customer-specific logic. The approval that routes differently in one region. The field that means something else in this customer's CRM. The prompt that needs their vocabulary. The question is where that logic lives.

If it lives in code, the FDE forks the product for each customer, and the product team inherits forty forks. If it lives in configuration, the FDE changes a setting, the product stays one product, and the setting itself becomes data the product team can read across customers.

The boundary has to be drawn on purpose. It is the same argument this handbook makes in [substrate-first engineering](#): engineering invests in the thing that lets everyone else ship safely, not in features. For a forward deployed model, the substrate is the configuration surface. The test is simple. Take the last three things an FDE did for a customer. If any of them required a pull request to the core product, the boundary is in the wrong place, and the fix is a product change, not a better FDE.

Sierra's move of workflow building into a no-code studio is this boundary being pushed outward. Every workflow that used to need an agent engineer's code is now a configuration the customer or the FDE can change. That is the last mile being productized from the inside, and it is the whole answer to the objection that forward deployment does not scale.

Two: the sandbox

An FDE's first month at a customer is either spent on the outcome or spent building an environment. Which one depends entirely on what the product provides.

The sandbox needs to mirror the customer's data shapes and integrations closely enough that what works there works in production. It needs guardrails that make it impossible to write to the customer's live system by accident or to run up a model bill nobody approved. It needs isolated deploys, so a mistake affects one customer's sandbox and nothing else.

This is the same substrate a Product Builder needs to prototype safely, applied per customer. If your product team has already built it for internal builders, the FDE version is an extension. If not, the FDE will build it themselves, badly, for one customer, and the next FDE will build it again.

Three: the eval harness

The most important sentence in this chapter is that the eval is the contract.

Before an FDE builds anything, they and the customer agree on what the outcome looks like as an executable test on the customer's real data. Resolution rate above a threshold on a held-out sample of last month's tickets. Correct routing on a hundred real approvals. Whatever the outcome is, written as something that passes or fails. The deployment ships when it passes.

This does three things at once. It stops the definition of done from being negotiated by feel, which is where deployments stall. It gives outcome pricing something to bill against, because the metric on the invoice is the metric in the test. And it makes the FDE's work legible to the product team, because a test that passed on customer A's data is a test the product team can run on customer B's.

I wrote the general case in [the eval is the spec](#). For forward deployment, the eval is also the statement of work.

Four: the observation record

An FDE is accountable for an outcome. To be accountable for a number you have to be able to explain why it moved. That requires a record, per customer, of what the system did: which actions ran, which a human reversed and how fast, which got escalated, which rule fired, and how each outcome closed against the decision that produced it.

Without that record the FDE is guessing. The resolution rate dropped, and the answer to why is a week of log-diving in the customer's systems. With it, the answer is a query.

The record also matters for the product. Forty customers' observation records, read together, show which per-customer configurations are the same pattern with different names. That is what the product team generalizes. The [deployment-to-product loop](#) does not work without it, and I would argue it is the most underbuilt of the five, because it is invisible in a demo.

In Heidi every act carries a receipt and every outcome gets labeled against the decision that triggered it. I built that for the guard layer, so an agent's autonomy could be gated on evidence. It turns out to be exactly what a person accountable for a tenant's outcome needs as well, and I did not see that coming.

Five: the feedback path with an owner

Palantir's loop worked because product development engineers had an explicit job: take what the FDEs built and generalize it. That job had owners. It was not a suggestion box.

The equivalent in your company is a person on the product team whose scorecard includes how many per-customer configurations became product features this quarter, and a cadence where FDEs bring what they built. Without that owner, the observations pile up and nothing generalizes, and you are back to forty forks. With it, each deployment makes the next one cheaper, which is the only reason the economics of the model work.

The order matters

Do the configuration boundary first, because it determines whether anything else compounds. Then the sandbox and the eval harness, which are the same investment as making your own product team faster and can be built together. Then the observation record. Then name the owner of the feedback path.

Then hire the first FDE. Not before. An FDE who lands on a product without these five will produce a heroic deployment for one customer and a maintenance burden for everyone else, and it will look like the model failed when the product did.

This wave is part of the running argument on [AI Product Management](#): the role is defined by what one person can own once building is cheap, and what one person can own depends on what the product lets them.

Start this week

Take the last thing an FDE, an implementation engineer, or a solutions person built for a single customer. Ask one question about it: did it require a change to the core product's code?

If yes, that is the configuration boundary in the wrong place, and moving it is the highest-return product work available to you this quarter.

Sources: [Kevin Bai, FDE 101 and the platform prerequisite, AI Engineer 2026 \(via the Sierra talk series summary\)](#) · [Nabeel Qureshi, Reflections on Palantir](#) · [Falk Gottlob, The Sierra Playbook](#)

The Deployment-to-Product Loop: How FDE Work Becomes Roadmap

Palantir's ingestion, visualization, and app builder all started as one FDE's hack for one customer. The loop that generalized them is why the model made money.

Web version: <https://falkster.com/handbook/fde-to-product-loop> · last updated 2026-09-17

Palantir's data ingestion tool started as something a forward deployed engineer built on site because one customer's data would not load. Its visualization layer started the same way. Its app builder started the same way. Nabeel Qureshi lists them as examples of a single pattern: an FDE hacks together what one customer needs, a product development engineer takes it back and generalizes it, and it ships to everyone as a Foundry component.

That loop is the entire economic case for forward deployment. Without it, each engagement is a services contract with the margin of one. With it, the customer pays for the first version and every later customer gets it cheaper, which is the only thing that separates the model from consulting. Most companies adopting the FDE title in 2026 have the first half of the loop and not the second, and they are about to find out what that costs.

The short version

The deployment-to-product loop has two halves with two different owners. FDEs build the fast, overfit version for one customer and are measured on that customer's outcome. A product team with the explicit job of generalizing takes what was built, reads the observation records across customers, and turns the third occurrence of the same pattern into a feature or a configuration. The kill discipline is the same as for prototypes: one customer's workaround dies, three customers' identical workaround gets productized. The PM sits at the junction as editor, deciding what earns generalization and writing the eval the generalized version has to pass. When both halves have owners, each deployment makes the next one cheaper. When only the first half does, the company is a consulting firm with a software company's burn.

Half one: build fast, overfit on purpose

The FDE's half is the easy one to staff and the hard one to keep honest.

Qureshi's line for it was that your job was to solve the problem and not worry about overfitting. That is a real instruction, not a shrug. An FDE who tries to build the general solution on site produces

something that serves the customer in front of them badly and everyone else not at all. The overfit version ships this week and produces the outcome. Generalizing it is somebody else's job.

What keeps this half honest is the [configuration boundary](#). If the overfit version lives in configuration, it is data the product team can read. If it lives in a fork of the codebase, it is a maintenance obligation nobody will read. The FDE should overfit in configuration and prompts, and reach for code only when the boundary cannot be moved.

Half two: generalize, with an owner

The second half is where the model lives or dies, and it fails for a reason that has nothing to do with skill. Nobody owns it.

FDEs are measured on their customer. Generalizing what they built helps a customer they will never meet and costs time the customer in front of them needs. Product engineers are measured on the roadmap. A deployment learning arrives as an interruption to the thing they committed to this quarter. Both incentives are correct for the people who hold them, and between them the loop starves.

Palantir's answer was to make generalization a job. Product development engineers had it as their explicit charter. Sierra's answer, per Meurer, is to blur the line on purpose: one hiring bar, overlapping teams, shared accountability, so the same person who overfit on site is in the room when it gets generalized. Decagon's answer is that FDE simply is product engineering with the same reporting line.

Pick one. The wrong choice is any structure where generalizing is a nice-to-have.

The kill discipline

Not everything an FDE builds should become product. Most of it should not. One customer's workaround is a workaround. It reflects their legacy system, their politics, their one unreasonable stakeholder. Productizing it puts one customer's accident into everyone's product.

The rule I use is the same one this handbook applies to prototypes: one is an anecdote, two is a coincidence, three is a pattern. When the observation records from three customers show the same configuration solving the same problem with different names, that is the signal to generalize. Below three, the answer is no, and the product team has to be comfortable saying it to an FDE who is proud of what they built.

This is the [kill discipline](#) applied to a different input. The backlog of FDE learnings is as dangerous as any other backlog. It grows, nobody reads it, and the act of writing something down substitutes for deciding about it. Read the records, decide, kill or generalize, and do not keep a list.

The PM as editor

Somebody has to sit at the junction. In this handbook I call that person the [PM as editor](#), and the framing was built for reviewing agent output. It applies without modification to deployment output.

The editor does not build. The editor reads what the FDEs built and what the observation records say happened, decides which patterns have earned generalization, and writes the eval the generalized version has to pass. That last part matters. The FDE's eval was written for one customer's data. The product version has to pass on three customers' data, and the PM writes that test before the product engineer starts.

This is the highest-leverage product judgment work available in a forward deployed company, and it is almost never staffed. Most companies have a roadmap process that treats deployment learnings as feature requests, which puts them in a queue behind whatever the sales team promised. The editor role short-circuits the queue: it is a standing decision, made weekly, by someone whose scorecard is what got generalized.

What changes when the building is cheap

The loop was designed when the FDE's artifact was hand-written code. Generalizing meant reverse-engineering what one engineer did under pressure, which is why product development at Palantir was a large and separate team.

When the FDE built with a coding agent, the artifact is different. There is a spec, a prompt, a configuration, and a test that passed. Generalizing means comparing specs across customers, not reading code. Three specs that ask for the same thing in different vocabulary are a pattern you can see in an afternoon.

This is the part of the model that is new in 2026 and it is the reason I think the loop will close faster than it did at Palantir. The observation record and the eval make each deployment legible. Legible deployments are easy to compare. Easy comparison is what the editor needs. The whole loop gets shorter.

It is also, for what it is worth, the loop I am running inside Heidi between what the agents do in one tenant and what changes in the product for all of them, with the difference that the thing being generalized is an agent's behavior rather than a person's build. Same junction. Same editor.

The next chapter takes up [the economics](#), which is where the loop either pays for itself or does not. And the argument underneath, on [AI Product Management](#), is that once building is cheap the scarce work is deciding what gets built for everyone.

Start this week

List the last ten things your deployment or implementation people built for individual customers. For each one, write down how many other customers have the same problem.

Anything at three or more is a feature you have already paid to discover and have not shipped.

Anything at one is a workaround, and the honest move is to leave it there. If you cannot answer the question, you do not have an observation record, and that is the thing to build first.

Sources: [Nabeel Qureshi, Reflections on Palantir](#) · [Natalie Meurer, The Dirty Secret of Forward Deployed Engineering](#), [AI Engineer 2026](#) · [Falk Gottlob, The Sierra Playbook](#)

FDE Economics: Margins, Outcome Pricing, and When to Stop

Forward deployment costs like services and has to earn like software. The math works only if the loop closes and the pricing puts revenue behind the result.

Web version: <https://falkster.com/handbook/fde-economics> · last updated 2026-09-17

Forward deployment costs like services and has to earn like software. That sentence is the whole chapter. Everything else is the conditions under which it is possible and the signals that tell you it is not.

An engineer on site at one customer for a quarter is a payroll line against one contract. That is a services margin, and no amount of calling the person a forward deployed engineer changes the arithmetic. What changes it is whether the engagement produces something that every later customer gets for less, and whether the pricing captures the result the engineer guaranteed. If both happen, the services cost was an investment. If neither does, you have started a consulting firm with a venture-backed burn rate.

The short version

Forward deployment compresses gross margin in the short run, and the model only earns it back if the deployment-to-product loop closes and the pricing puts the vendor's revenue behind the outcome. Outcome pricing is the reason the model is spreading: billing per result requires someone to guarantee the result, and that is the FDE. The buyer's objection is lock-in and dependency, and it is legitimate, but the fix is contractual: the observation record and the configuration belong to the customer and port with them. The signal to stop sending engineers is when the product's configuration surface has absorbed enough that the customer, or their own AI engineer, can deploy from the product. That was the goal all along. A company that never reaches it has not built a product. It has built a staffing agency with an API.

The margin math

Start with what the objection gets right. Sacra named implementation intensity as Sierra's biggest scaling risk, and the concern is fair: high-touch deployment compresses margins and gates growth on engineer hiring. Every deployment needs a person. People do not scale like software.

Palantir ran this trade for a decade and the number worth knowing is that until 2016 it employed more FDEs than product engineers. That is the cost side of the loop. What made it work was the

other side: roughly \$1.5 million in revenue per employee, per Plank's figure, with almost no traditional sales force, because the FDEs were the sales motion and what they built became product. The services cost bought two things at once, the deployment and the roadmap.

If your loop does not close, you pay the cost and get one of the two. That is the entire difference between the companies for which this model works and the ones for which it is a slow way to discover they are a services business. I covered the loop in the [deployment-to-product chapter](#). The economics do not exist without it.

Why outcome pricing forces the question

Here is the part that makes the model unavoidable rather than optional.

Seat pricing never needed a guarantor. The customer bought seats, the vendor delivered software, and whether the software produced a result was the customer's problem. That arrangement is what [the AI business models argument](#) says is ending: software priced per seat is priced against a labor cost that AI removes, so the price has to move to the outcome.

The moment it moves, someone has to guarantee the outcome. Meurer's framing from inside Sierra is the cleanest I have seen. Agents push software pricing into the outcome quadrant. Outcome pricing requires the seller to guarantee the result. The question of how you guarantee it has one answer, and it is forward deployed engineering.

So a vendor pricing on outcomes is not choosing whether to have FDEs. It is choosing whether to name them. The cost is already in the pricing model. This handbook's [pricing for AI products](#) chapter covers the pricing side. The FDE is the delivery side of the same decision.

The buyer's side of the table

I want to give the objection its full weight, because it is the strongest argument against the model and it comes from people who are not wrong.

Andrew Ng has warned that letting a vendor's forward deployed engineers wire your processes to their stack significantly reduces optionality, and that a company's own AI engineer, on staff and building across models, is the structurally better bet. Plank's guide lists the three risks a buyer should hold: dependence deepens while the FDE is helping, capability does not transfer if the FDE ships without documentation or enablement, and Gartner expects enterprises to abandon FDE-heavy engagements on cost grounds. It also cites the figure that deployment is only about 20 percent of an AI system's lifetime cost, with the rest in operation and change, which means an FDE who leaves after go-live has helped with the small part.

All three risks are real. They are also all solvable in the contract, and a vendor who will not solve them there is telling you something.

The observation record belongs to the customer. What the agents did in your org, what got reversed, what got escalated, ports with you in a form you can load elsewhere. The configuration belongs to the customer. Every workflow, mapping, and threshold the FDE built is readable and exportable, not buried in the vendor's code. Enablement is a deliverable with a test: the customer's own engineer can change the configuration without the vendor by day 60, or the engagement is not done. If a vendor agrees to those three, Ng's optionality concern is answered. If they will not, the concern was correct.

I am on the vendor side of this table with Heidi, and I would rather write those terms myself than have a buyer extract them. A vendor whose moat is that the customer cannot leave does not have a moat. It has a hostage.

When to stop sending engineers

The model has an endpoint, and companies that forget this are the ones Gartner is describing.

Every deployment should move the [configuration boundary](#) outward. Every generalized pattern should mean the next customer needs less of an engineer. The endpoint is a product a customer can deploy themselves, or with their own AI engineer, from the configuration surface. Sierra's move of workflow building into a no-code studio is this. Airbus distributing the FDE function across its own customer engineers, per Meurer, is this from the buyer's side.

When you reach it, the FDE headcount per customer falls toward zero and the margin recovers to software. That was the plan. A company that reaches year four with the same FDE ratio it had in year one has not been running a forward deployed model. It has been running a services firm and calling it product.

The signal to watch is FDE hours per deployment, quarter over quarter, for the same class of customer. Falling means the loop is closing. Flat means it is not, and the fix is in the product, not in hiring more engineers.

Start this week

Pull the FDE hours, or implementation hours if that is what you call them, for your last four deployments of roughly the same size, in order. Draw the line.

If it slopes down, the loop is closing and the margin will follow. If it is flat, you have the cost structure of the model without the compounding, and the next chapter on [building the team](#) will not fix it. Only the product will.

Sources: Natalie Meurer, The Dirty Secret of Forward Deployed Engineering, AI Engineer 2026 · Plank, What Is a Forward-Deployed Engineer (revenue per employee, lifetime cost share, Gartner expectation) · Andrew Ng on forward deployed engineers and optionality · Falk Gottlob, The Sierra Playbook (Sacra on implementation intensity)

Building the FDE Team: Reporting Line, Bar, Pairing, and Comp

FDEs who report to sales become sales engineers within two quarters. Reporting line, hiring bar, pairing, and comp decide whether the team lasts a year.

Web version: <https://falkster.com/handbook/fde-org-design> · last updated 2026-09-17

Four decisions decide whether a forward deployed team exists in a year. Who they report to. What bar you hire against. Whether you pair the builder with someone who owns adoption. And what you pay on. Get the first one wrong and the other three do not matter, because within two quarters you will have a sales engineering team with a fashionable name.

The short version

FDEs report to product or engineering, never to sales, because the reporting line sets the scorecard and a bookings scorecard turns an FDE into an SE. The hiring bar is the product engineering bar plus the customer: production code, plus the ability to learn an industry's vocabulary in weeks and earn an operator's trust. It is a senior role; 3 percent of open FDE roles in mid-2026 were early-career, median comp was \$185,000, and frontier labs paid above \$250,000. Pair every builder with someone who owns adoption, Palantir's Delta and Echo, because the technical build and the organizational change are different skills. Pay variable comp on the customer's outcome and on the renewal it produces, never on new bookings. And size the team to shrink per deployment, because a permanent FDE ratio is a services business.

Decision one: the reporting line

Everything follows from this, and it is the decision most companies get wrong because sales asks first.

Sales wants FDEs because FDEs close deals. That is true. An engineer who can build the customer's workflow on site during the evaluation is the most persuasive demo in the building. So sales asks for the headcount, gets it, and the FDE reports to a sales leader whose number is bookings.

Within a quarter the FDE's calendar is pipeline. Within two, their variable comp is tied to it. The deployment they were accountable for gets handed to a ticket queue at the hard part because the next prospect needs a proof of concept. The customer whose outcome they owned has an engineer for three weeks. The FDE either becomes an SE or leaves, and either way the model is gone.

The alternative is the one Decagon states plainly: FDE is product engineering, same bar, same reporting line. Sierra gets to the same place by blurring the teams on purpose, one hiring bar and shared accountability, so the person who overfit on site is in the room when it gets generalized. Either structure keeps the FDE's scorecard where the [deployment-to-product loop](#) needs it: on the customer's outcome and on what got generalized.

If you take nothing else from this chapter: the FDE's manager should be the person whose job it is to close that loop, and that person does not carry a bookings number.

Decision two: the hiring bar

The bar is the product engineering bar, and then something on top.

Nabeel Qureshi's description of who succeeded as a Palantir FDE is worth reading as a job description. Rapid acquisition of an industry's vocabulary. Reading social dynamics and power structures. Earning a customer's trust by being present. Delivering visible value fast. He notes that the same instincts make good founders, which is why Palantir became a founder factory and why the role is hard to fill.

Then the engineering. Production code, in the customer's environment, with their identity provider and their nightly jobs and their one API that times out under load. A coding agent now does the plumbing, so the bar is not typing speed. It is knowing when the agent's output is wrong for this customer, which is a judgment only someone with real production scars can make.

Plank's July 2026 data puts the seniority in numbers: 3 percent of open FDE roles were early-career, median comp \$185,000, frontier labs above \$250,000. That is a senior hire. The failure mode is treating the FDE req as a way to add engineering capacity cheaply, filling it with someone who can build but cannot read a room, and discovering six months later that the deployments are technically complete and unadopted.

The three doors in the [transition chapter](#) are the three sourcing pools. PMs who already ship, engineers who want the customer, and SEs who want to stay. Interview for the missing third thing in each case. I covered the general version of this in [hiring the Builder PM](#), and the interview is the same shape: put them in front of a real customer problem with real data and watch what they build and what they ask.

Decision three: the pairing

Palantir did not send engineers alone. The Delta built. The Echo, a deployment strategist, handled adoption, executive alignment, and the politics of getting an operations director to change a workflow that had run the same way for twelve years.

Most companies copying the model in 2026 hire Deltas and skip Echoes. The result is a deployment that works and does not get used. The engineer built exactly what was asked, the eval passed, and the team on the floor kept using the spreadsheet because nobody did the organizational work. Then the outcome number stays flat and the FDE, who is accountable for it, is blamed for a problem that was never technical.

The Echo does not have to be a separate headcount at a small company. It can be the FDE's manager, or a customer success lead assigned to the deployment, or in the best case an FDE who has both skills. But the work has to be owned by name. Adoption is a deliverable, and it is not the same deliverable as the build.

Decision four: comp

Pay variable on the outcome. Not on bookings, not on shipped integrations, not on hours. On the customer's outcome metric and on the renewal or expansion that the outcome produces, because that is what the customer is paying for and what the company is billing for.

Base at the product engineering band or above. The role is harder than product engineering, not easier, and paying below the band selects for people who could not get the product job. If the company runs outcome pricing, the FDE's variable comp and the customer's invoice should move on the same number. That alignment is the whole point of the model and it should be visible on the pay stub.

Sizing: plan to shrink

The last decision is the one nobody wants to make in a planning meeting. Size the team to need fewer FDEs per deployment every quarter.

A permanent FDE-to-customer ratio is a services business. The model works when each deployment moves the [configuration boundary](#) outward and the next customer needs less of an engineer. The team may grow in absolute terms with the customer count. Per deployment, it has to fall, and the plan should say so.

Hire for the deployments in the next two quarters. Watch FDE hours per deployment for the same customer class. If the number is flat, the fix is in the product, and the [economics chapter](#) covers what that costs.

This wave sits in the running argument on [Product Leadership](#), which is that the org chart is a product decision, and the FDE team is the clearest case of it I know.

Start this week

Find where your FDEs, or the people doing that job under another title, sit on the org chart. Follow the line up until you reach someone with a number.

If the number is bookings, move the line before you hire another one. If the number is the customer's outcome, you are set up for the model to work, and the next thing to check is whether anyone is paired with them on adoption.

Sources: [Nabeel Qureshi, Reflections on Palantir](#) · [Plank, What Is a Forward-Deployed Engineer \(July 2026 role and comp data\)](#) · [Natalie Meurer and the AI Engineer 2026 FDE talk series, including Decagon's framing](#)

Keep going

- The living version of every chapter: <https://falkster.com/handbook>
- The Product Builder Standard, the six skills: <https://falkster.com/pm-standard>
- What's your Builder level? A twelve-question self-assessment: <https://falkster.com/builder-level>
- The free 7-day Product Builder course: <https://falkster.com/course>
- Every template in the book, free: <https://falkster.com/toolkit>
- Working with me: <https://falkster.com/work-with-me>