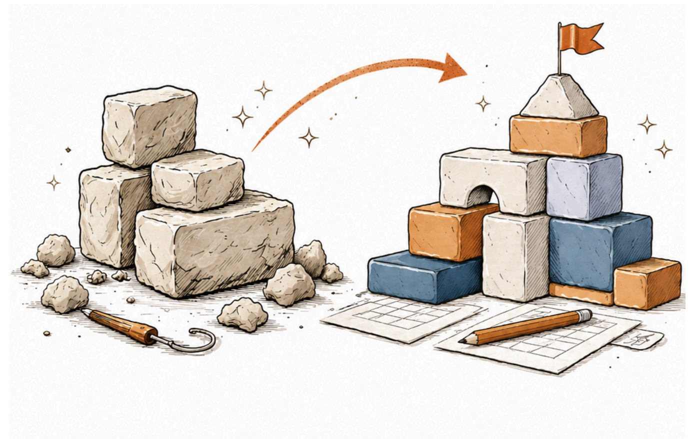
Shaping Enterprise



A Field Guide to Shape Up in B2B Product Organisations

Michael Backes

Copyright © 2026 Michael Backes

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

Published by Inflection Advisory

ISBN: 9798258924506

Independently published
First published 2026

Disclaimer

The information in this book reflects the author's personal experience and is intended for educational purposes. Results will vary depending on your organisation, team, and context. The author makes no guarantees of specific outcomes from applying the methods described.

Shape Up is a methodology developed by Basecamp and described in the book *Shape Up* by Ryan Singer, available free at basecamp.com/shapeup. This book builds on those principles and describes one practitioner's adaptation of them in an enterprise B2B context. It is an independent work and is not affiliated with or endorsed by Basecamp.

Acknowledgements

Books like this don't come from a single person sitting in a quiet room having brilliant thoughts. They come from years of messy, real-world experience — and the people generous enough to share it with you.

Ryan Singer deserves the first mention. Of the hundreds of books I've consumed over the years, few have had the end-to-end impact that *Shape Up* had on me. What made it different wasn't that it handed me a silver bullet — it didn't, and anyone looking for one will be disappointed. What it gave me was a reframing. It put risk at the centre, which is where it belongs for anyone operating at the executive level. It gave my team and me a language and a foundation we could make our own. We didn't follow it to the letter. We weren't supposed to. Ryan was onto something when he wrote the first book, and this one exists because of what he started.

Nicola Marshman — a steady presence through the rough days. Writing is mostly a solo exercise, which means it mostly happens in the dark, fuelled by bad humour and stubbornness. She tolerated both, and that made more of a difference than she probably knows.

Luis Rangel, Jose St. Bernard, and Denys Burdeniuk were there at ground zero. The first batch of engineers who were all-in. They didn't just follow the principles — they helped shape them. Engineers get thrown into stereotypical buckets far too often. These three had a larger footprint on walking the path than any label could capture.

Hakan Kaya and Pau Vicedo came into the picture later, and what they gave me was something harder to find than early enthusiasm: proof. Proof that this held up at scale, that it wasn't a happy accident of a small team, and that it worked even as my role changed around it. That kind of validation matters more than most people realise.

Ryan Scoville and Olaf Gunkel — for being there during the rough stretches, tolerating my humour, and the burritos. Some relationships are best measured in shared meals and bad jokes, and I'm grateful for both.

And finally, **my family**. They gave me something that no methodology, no framework, and no book can manufacture: the freedom to try things and not be afraid to fail. That's the real foundation. Everything else is built on top of it.

Table of Contents

- Introduction: The Methodology Shouldn't Get in the Way
- Chapter 1: The Roadmap Isn't Dead
- Chapter 2: Appetites Over Estimates
- Chapter 3: Shaping for Enterprise
- Chapter 4: The Two-Horizon Betting Table
- Chapter 5: Appetite-Weeks
- Chapter 6: Running the Cycle
- Chapter 7: Cool-Down: The Sacred Space Everyone Tries to Kill
- Chapter 8: Shape Up Isn't the Whole System
- Chapter 9: When Shape Up Breaks
- Chapter 10: The Organisational Change Nobody Warns You About
- Chapter 11: Getting Started Monday Morning
- Chapter 12: AI in the Process
- Acknowledgements
- Appendix: Templates & Artifacts

Introduction: The Methodology Shouldn't Get in the Way

You're going to go nuts.

If you're leading product at an enterprise B2B company, you already know this. You have large customers who represent real revenue pulling you in one direction. You have a sales team making commitments based on what they heard in discovery calls. You have a CS team fielding escalations and expansion opportunities. And somewhere in the middle of all this, you're trying to hold onto a product vision and roadmap that makes strategic sense.

It's pretty simple, really. In enterprise—whether you're building SaaS platforms or AI products—you have very specific types of stakeholders, long sales cycles, and high expectations. The tension is constant. You can try to force a single methodology onto everything and watch it bend until it breaks. Or you can build something more flexible.

Here's what I learned after 25 years building enterprise platforms: the methodology shouldn't get in the way of people having success.

That's the thesis of this book. Shape Up isn't a silver bullet. It's an engine. And engines only work when they're part of a system designed for the conditions you're actually operating in. What I'm going to show you isn't Basecamp's Shape Up copied into an enterprise context. It's an operating system I built around Shape Up over six years at a B2B fintech company, adapted from hard lessons learned across startups I founded, companies I worked in, and businesses I watched as a VC.

This is a practitioner's guide. If you're looking for theory, there are other books. If you want to know what actually worked—and what broke along the way—keep reading.

How I Got Here

I've been building enterprise platforms for over 25 years. I've been inside corporations with 100,000+ employees and founded several companies myself. I ran a startup factory inside a large organization, launched businesses from scratch, closed deals as the first salesperson, and scaled companies up as a venture builder at Liquid Labs, where we backed 16 companies with €87 million in capital and six exits.

All of that meant I was constantly exposed to different processes. Traditional project planning in the big companies. Various flavors of Agile in the startups. Zero process in the earliest stages. I've been on the front lines doing customer implementations while simultaneously managing the product roadmap. I've done the first several million in sales for my own platforms and had to figure out how to stop being dragged around by large customers while maintaining some control over where the product was going.

And I became very critical of how people were using Agile.

Here's the thing: Agile is a framework. But everyone was implementing it as a copy-paste template. We all want the silver bullet solution. It feels easier than building something based on context. The problem was that the rigid implementations I kept seeing couldn't adapt while still providing stability. Teams would fall behind. A single impediment would throw everything off.

It felt like watching Just-In-Time manufacturing applied to an assembly line where you're not entirely sure what parts are coming next. JIT works beautifully when you have predictable components arriving at predictable intervals. It falls apart when the parts show up out of order, or not at all, or turn out to be the wrong size.

That's what enterprise product development actually looks like most of the time. You don't have predictable parts.

The Decision

When I started my last company in 2019, I'd already experimented with Shape Up a bit. I decided to go all-in. Not copy-paste it from Basecamp's book, but use it as a framework—the way Agile was supposed to be used—and adapt it to my context.

That context was a B2B collections and recoveries platform. Enterprise sales cycles. Large financial institutions as customers. Regulatory requirements. The exact kind of environment where people assume you need heavyweight processes and Gantt charts and quarterly planning rituals.

I ran Shape Up in that environment for over six years. It worked. Not because Shape Up is magic, but because I built an operating system around it that addressed the realities of enterprise work.

That operating system included things Basecamp never talks about because they don't need them:

A thematic roadmap structure that gave stakeholders visibility without locking us into feature commitments.

A two-horizon betting table that let us triage immediate needs while making strategic bets on the future.

A capacity model based on appetite-weeks instead of estimates, which completely changed how we talked to sales and customer success teams.

A dual-track system where Shape Up governed product development, but a parallel Kanban flow handled the service-layer work that enterprise customers generate—integrations, onboarding, external-timeline dependencies.

And a lot of organizational change that nobody warns you about when you adopt a new methodology.

Why This Book Exists

This is the field guide I wish I'd had when I started. It's the operating system I built, piece by piece, over six years of running Shape Up in an enterprise B2B environment. Every chapter addresses something that Basecamp's book doesn't cover because they don't face the same constraints.

You'll learn how to maintain a roadmap that satisfies stakeholders without turning back into a feature factory. How to shift everyone from demanding estimates to thinking in appetites. How to train product people to shape with business framing and ROI awareness, not just solution sketches. How to run a betting table when you can't have the founders in the room making every call. How to handle the reality that some work just doesn't fit the six-week cycle model, and what to do about it.

And here's something I haven't seen addressed anywhere else: Shape Up and AI are surprisingly good companions.

I've used this operating system with AI startups, and I've started using AI in the process itself. The nature of AI product work—high uncertainty, emergent requirements, the need to experiment and learn quickly—maps remarkably well to Shape Up's appetite-based approach. When you're building features that depend on behavior that you can't fully predict upfront, estimation is theater. Appetite is honest.

I want to introduce the entire process without AI and then give you arguments and a few examples of how AI puts 100x acceleration on top. But the underlying principles don't change, and I find that they are important to cover instead of a list of trendy AI hacks.

Who This Book Is For and How It's Structured

This book is written for two audiences simultaneously: the executives deciding whether to adopt Shape Up—CPOs, VPs of Product, heads of engineering—and the practitioners who need to run it day-to-day. Every

chapter delivers both the strategic rationale and the steal-ready frameworks and templates to make it operational.

You don't need to have read Basecamp's *Shape Up* book to understand this one, but it helps. I'm not going to re-explain the basics of shaping, betting, or hill charts. If you need that foundation, go read Ryan's book first—it's free online. I'm assuming you understand the core concepts and you're here because you want to know how to make them work in an enterprise context.

The book moves from philosophy to operating model to implementation. Part One sets the context—why Shape Up works for enterprise and how to shift from estimation to appetite thinking. Part Two gives you the system, every piece of it in detail: shaping, betting, capacity, cycles, cool-down, and the dual-track model. Part Three covers the human side—what breaks, what cultural changes you need to make, and how to actually get started on Monday morning. Lastly, I give you how AI fits into this process, and why the pitch is even more important now. There's also an appendix with templates you can steal, including a couple that you can use with an AI model.

I'm going to be honest about what worked and what didn't. About the disruptions we allowed (twice in six years) and why we allowed them. About the organizational resistance we hit and how we worked through it. About the tradeoffs inherent in this approach.

Because here's the reality: Shape Up isn't right for every organization, and even when it's right, it's hard. But if you're in an enterprise B2B environment, tired of estimation theater and backlog grooming as a full-time job, and looking for an operating system that can handle competing stakeholder demands without losing strategic coherence—this might be what you need.

Let's get started.

Chapter 1: The Roadmap Isn't Dead

Maybe once software started becoming "agile," someone should have picked a different name for the roadmap. Something that didn't carry the legacy of a very stable, top-down execution plan. Because most of the time in enterprise software, you're navigating uncharted waters. You're making educated guesses about where you're going, adjusting as you learn, responding to market shifts and customer feedback.

Sailors used to call this "dead reckoning"—estimating your position based on your last known location, your speed, and your direction, while accounting for wind and currents. It's navigation through uncertainty.

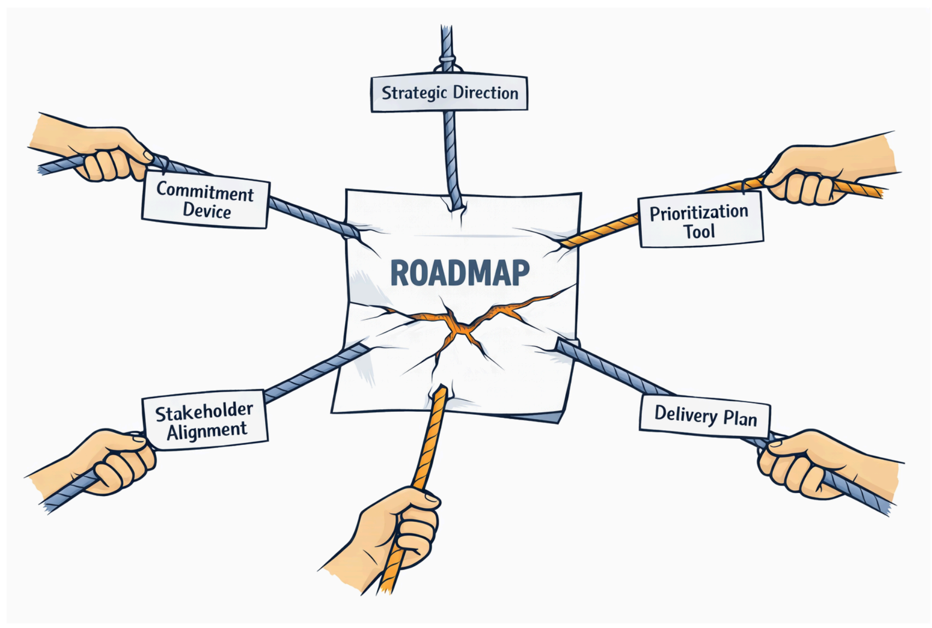
But if we called it that, people would think we're all pirates instead of product leaders making measured movements toward a destination.

Might be a little more fun, though.

Here's what I want to establish right up front: the roadmap isn't the problem. The problem is that enterprise organizations overload the hell out of it. They try to make the roadmap function as a commitment device, a prioritization tool, a stakeholder alignment artifact, and a detailed delivery plan all at the same time.

That's too much weight for one document to carry. And when it inevitably buckles under the pressure, everyone blames the roadmap itself instead of recognizing that we were asking it to do five different jobs.

Shape Up lets you separate those concerns. And when you do, something interesting happens: the roadmap becomes more valuable, not less.



Separation of Concerns

In the operating system I built, three different mechanisms handle what traditional roadmaps try to do all at once:

The thematic roadmap handles the what and why. It communicates strategic direction without false precision on timing. It's organized around problems you're solving and value you're creating, not features you're shipping.

The betting table handles the which now. It's where you decide what gets built in the next cycle and what doesn't. This is your prioritization and selection mechanism, and it operates on a six-week cadence.

The cycles handle the how. Six weeks of focused, uninterrupted work where teams execute on what was selected at the betting table.

This separation is more honest than traditional roadmaps that pretend to be precise about dates and deliverables six months out. Everyone knows those dates are fiction. Everyone knows the priorities will shift. But we

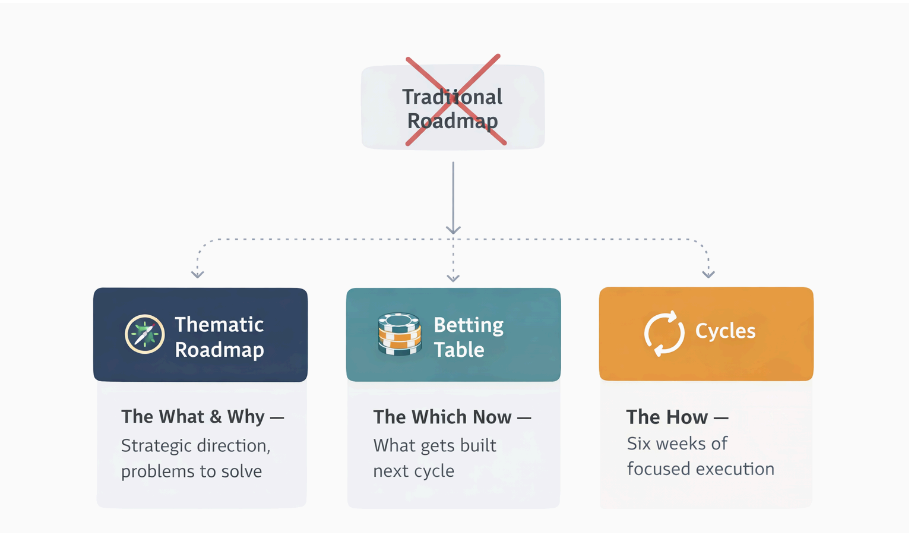
maintain the theater anyway because stakeholders feel better when they see a plan.

The thematic roadmap says: here's the strategic direction. Here are the problems we're committed to solving this quarter.

The betting table says: here's what we're building in the next six weeks, based on what we know right now.

The cycles say: here's how the work is actually progressing.

No fiction required.



The Quarterly Rhythm

Here's a practical advantage that matters in enterprise: two six-week cycles map perfectly onto a quarter.

Cycle one: four weeks. Cool-down: two weeks. Cycle two: four weeks. Cool-down: two weeks.

That's 12 weeks, which gives you about a quarter depending on how the calendar falls.

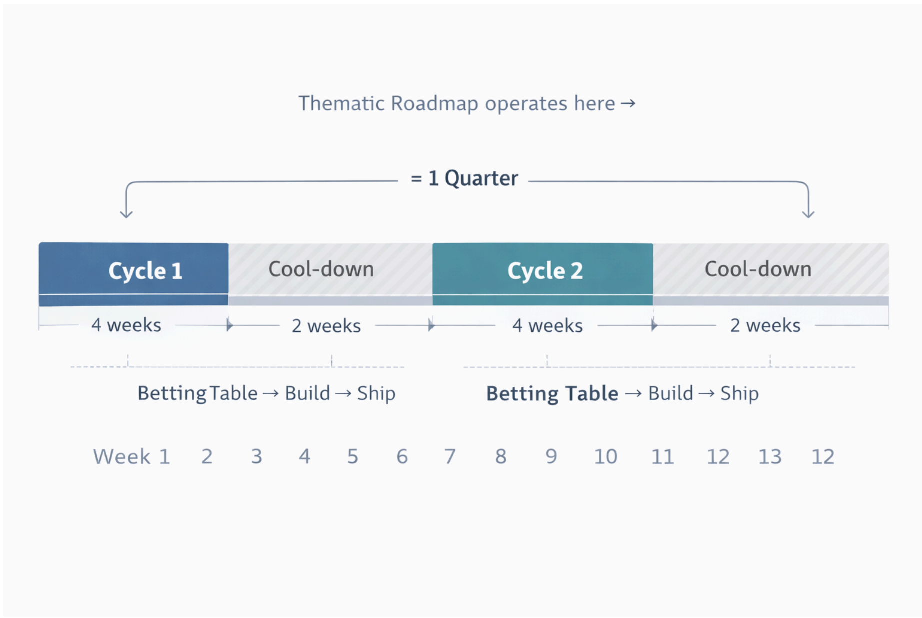
This isn't an accident. It's by design. Because B2B enterprise customers already think in quarters. They have quarterly business reviews. They touch budgets quarterly. Their executives report quarterly results.

When you align your product development rhythm to the cadence your customers are already operating on, communication gets easier. You're not constantly translating between your internal sprint schedule and their quarterly expectations.

The thematic roadmap operates at this quarterly level. Every quarter, you're communicating: here are the themes we're tackling, here's the value we're delivering, here's how it maps to your strategic priorities.

Then within that quarter, you run two cycles. Two betting tables. Two opportunities to adjust based on what you learned in the previous cycle.

It's strategic stability with tactical flexibility. Which is exactly what enterprise customers need, even if they don't always know how to ask for it.



The Before/After Story

Let me show you what this looked like in practice, because the transformation wasn't immediate and it wasn't clean.

Before we implemented the thematic roadmap, we didn't have a quarterly plan that anyone could actually work with. I don't mean we didn't have a document called a roadmap. We had plenty of those. I mean we didn't have a plan that reliably got everyone moving in the same direction.

There was constant conflict between sales, customer success, and product — with engineering caught in the middle.

Sales would come back from customer meetings with commitments they'd made—or at least heavily implied—about when certain features would be available. CS was fielding escalations and expansion opportunities, trying to get product to prioritize the work that would save accounts or grow them. Product was trying to maintain some coherent vision about where the platform needed to go strategically.

Everyone had visibility into their own part of the business and was trying to shove that part into what was essentially a build pipeline.

And look, when you're really small—early-stage product, direct sales, maybe doing customer implementations yourself—that fluid, Kanban-style motion is fine. You can be reactive. You should be reactive. You're learning.

But you hit a point in enterprise where that breaks down. People need better visibility into when certain core things are coming. Not exact dates necessarily, but a sense of sequence and priority that they can plan around.

We hit that point hard.

After we implemented the thematic roadmap, the nature of the conflict changed. Not overnight — this took a quarter or two to really settle in. But the arguments stopped being about features and started being about problems.

Here's the moment I knew it was working. Sales came back from a customer meeting — a large financial institution — and instead of saying "they need a reconciliation dashboard with these five specific features," they said "they have a reconciliation workflow pain point, and it's showing up differently than we expected because of how their upstream systems work."

That's a completely different input. That's something product can actually shape.

Under the old model, if we'd promised five specific features and then discovered during shaping that three of them didn't solve the core problem, we'd created an expectation we couldn't meet. But because the thematic roadmap put the focus on problems and outcomes, sales was going into meetings saying "we're tackling the reconciliation pain point — our goal is to reduce the time your ops team spends on manual reconciliation by 60%." That gave us room to find the right solution.

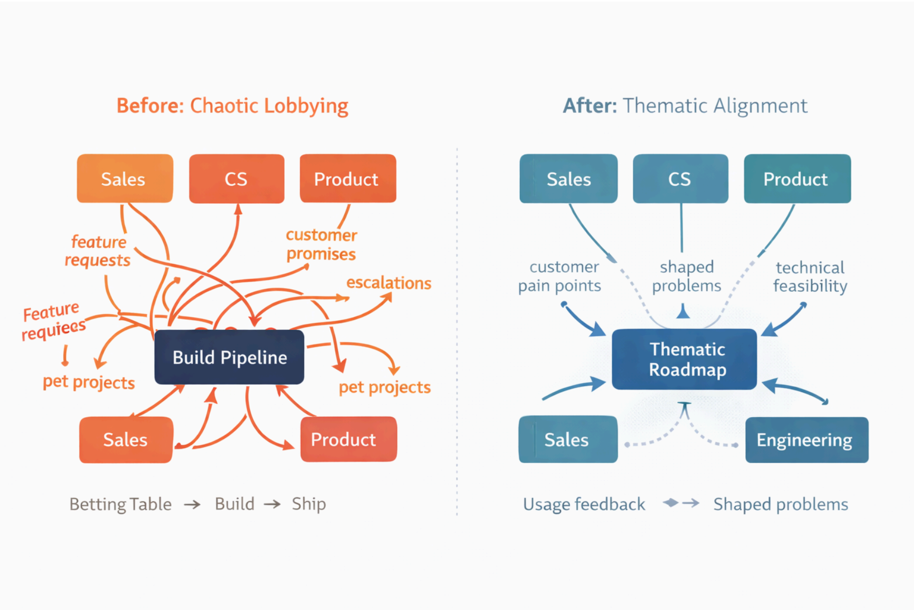
> ***"Sales didn't need to know what color the button was going to be. They needed to understand what problem we were solving."***

The shift cascaded. CS stopped lobbying for pet features and started collecting feedback that refined work already being shaped. They'd go into quarterly business reviews knowing the themes, so when a customer asked "what about that feature I requested?" CS could say: "Here's the theme we're working on that addresses that — let me show you what we're thinking and get your input." They became a feedback mechanism, not a lobbying group.

Product was back in control of what got built for the core platform — not in a dictatorial way, but informed by customer needs without being yanked around by whoever had the loudest voice or the biggest contract that week.

The chaos didn't disappear. But it became directed. Information was flowing better and everyone understood their role.

Making It Work



None of this happens automatically just because you publish a thematic roadmap.

The first time I presented the thematic roadmap to my VP of Sales, he looked at me like I'd taken something away from him. And in a sense, I had. I'd taken away the feature list he'd been using to make promises to customers. What I was giving him instead — themes and problems — felt less concrete. Less useful in a sales meeting. He wasn't wrong to be skeptical.

So I sat down with him for an hour. We walked through a real customer conversation he had coming up that week, and I showed him how to position the thematic roadmap: "We're solving the reconciliation problem" instead of "We're building a reconciliation dashboard." I showed him how framing it around the problem actually gave him more flexibility, not less, because he wasn't locked into features that might change during shaping.

By the end of the conversation, he got it. Not because I explained the theory, but because we rehearsed it against a real deal.

That's shuttle diplomacy — and it's the most important communication technique you'll use. Before you broadcast the roadmap to the wider organization, sit down one-on-one with every key stakeholder. Your VP of Sales, your head of CS, your engineering leads. Walk them through it. Listen to their concerns. Adjust where it makes sense. Explain the rationale where you're not going to adjust. Make sure they're not going to be surprised when the broader communication goes out.

This takes time. It's not efficient in the moment. But these people are your translators. They're going to explain this to their teams, and they need to understand it well enough to do that credibly.

After shuttle diplomacy, you broadcast — Slack, email, all-hands, whatever channels your organization uses. And depending on your org size, you follow up with grassroots efforts: town halls, office hours, Slack channels where the roadmap is discussed and questions are answered.

The goal with all of it is to reduce fear of uncertainty. Because that's what you're really dealing with.

Feature roadmaps with every dependency linked end-to-end aren't planning tools — they're security blankets. People are trying to build a guaranteed path from point A to point B because the alternative feels like chaos.

You're asking them to trust a model where the destination is clear but the route adjusts. That goes against human nature. It requires deliberate change management.

The Twist

Here's the thing nobody expects when they first hear about thematic roadmaps and Shape Up: you don't get rid of date-driven commitments.

You protect both your product vision and those commitments.

People assume that moving away from feature roadmaps means you can't make commitments anymore. That you're going to be too fluid, too agile, to ever promise a customer anything.

That's not what happens.

What happens is you separate the strategic roadmap (thematic, flexible about solutions) from the delivery schedule for customer commitments. You can have one roadmap and one delivery schedule. They inform each other, but they're not the same document.

When you need to make a commitment—a big customer with a contract contingent on a specific capability, a regulatory requirement with a hard deadline, a partnership that requires integration by a certain date—you can still do that.

But those commitments flow through a different part of the system. They get handled differently than your core product development work.

How does that work? We'll get to that in Chapter 8. For now, what matters is this:

> ***"The thematic roadmap isn't about avoiding commitments. It's about creating the space to make the right commitments, at the right time, for the right reasons."***

And when you're not making a commitment, you're free to chase the best solution to the problem instead of being locked into a feature specification someone wrote six months ago based on incomplete information.

Try This

If you're already operating with a thematic roadmap, great. Skip ahead.

If you're not, here's what to try this week:

Take your current roadmap and highlight everything that's a feature specification, a date commitment, or a "we're going to build X" statement. That's probably most of it.

Now ask: what problem does each of those things solve? What value does it create? Can you reframe the roadmap around those problems and value propositions instead of the features?

You don't need to publish this yet. Just try the reframing exercise and see what it reveals about how your roadmap is currently structured—and what's getting lost in translation when different stakeholders try to use it for their own purposes.

Chapter 2: Appetites Over Estimates

There's a question that haunts every enterprise product organization: "How long will this take?"

It's an impossible question. But we ask it anyway. We ask engineers to estimate work before they understand the problem. We ask product managers to predict timelines before the requirements are clear. We ask teams to commit to dates based on information we know will change.

Then we're upset when the estimates are wrong.

We run elaborate estimation ceremonies. Story points. Planning poker. T-shirt sizes. We slice epics into stories and stories into tasks. We calibrate velocity. We track burn-down charts. We adjust our models based on historical data.

And at the end of all that theater, we're still wrong about half the time.

Because estimation is fundamentally broken in product development.

Not because engineers are bad at estimating. Not because product managers are being unrealistic. But because we're asking the wrong question.

"How long will this take?" assumes there's a knowable answer if we just think hard enough about it. It assumes the requirements won't change. It assumes we fully understand the problem. It assumes no surprises, no dependencies, no discoveries that reshape what we're building.

Those assumptions are fantasy.

Shape Up asks a different question: "How much time is this worth?"

That's a strategic decision, not a prediction. And it changes everything.

Appetite: A Time Budget, Not a Prediction

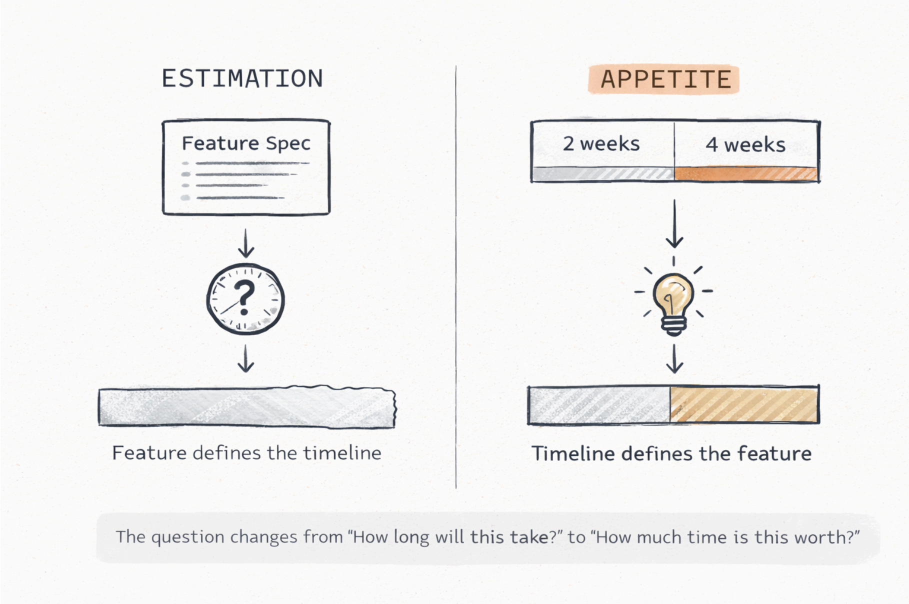
An appetite is the amount of time you're willing to invest in solving a problem. It's a budget. A constraint. A forcing function that shapes how you approach the work.

When you set a two-week appetite for a feature, you're not predicting it will take two weeks. You're saying: this problem is worth two weeks of our time and attention. If we can't solve it meaningfully in two weeks, we either need to reduce the scope, find a different solution, or accept that this problem isn't worth solving right now.

The framing is completely different.

With estimation, the feature defines the timeline. You ask: given everything this feature needs to do, how long will it take?

With appetite, the timeline defines the feature. You ask: given the time we're willing to spend, what's the best solution we can deliver?



This isn't just wordplay. It's a fundamental shift in how you think about building software.

When you frame work with appetites, you're forced to make strategic trade-offs up front. Is this problem worth two weeks? Four weeks? Six weeks? The answer depends on the value you expect to create, the risk you're managing, the opportunity you're pursuing.

And here's what happens when you start talking about appetite instead of estimates: the quality of your strategic conversations improves dramatically.

Instead of arguing about whether something will take three weeks or five weeks—a debate nobody can win—you're arguing about whether a problem is worth three weeks or five weeks of investment. That's a debate you can have productively, because it's about priorities, value, and trade-offs.

Leadership can engage in that conversation. Sales can engage in that conversation. Engineering can engage in that conversation. Everyone has relevant input.

But in enterprise, especially in B2B where you're managing large customers, complex sales cycles, and cross-functional stakeholders, getting people to shift from estimation to appetite is hard.

The Hardest Audience

The hardest group to convince of this shift wasn't engineering. It wasn't product. It wasn't even leadership.

It was sales and customer success.

The reason: they saw appetites as taking away their ability to control delivery timelines they were committing to with customers.

Think about it from their perspective. Sales is in the closing stages of a deal, and the prospect asks: "When can we have this feature?" They want to say: "It'll be live in Q4." That's how deals get closed.

Customer success is managing an expansion opportunity with an existing account. The customer is frustrated about a workflow gap, and CS wants

to say: "We'll have this fixed by February 19th." That's how you save the account or grow the relationship.

So when you show up and say "we're not doing estimates anymore, we're doing appetites," what they hear is: "We're taking away your ability to make commitments."

And look, they're not wrong to be concerned. This is a real tension. Enterprise B2B runs on commitments. Customers expect them. Sales needs them to close deals. CS needs them to manage relationships.

But here's what was actually happening under the old model—and everyone knew it, even if they didn't say it out loud.

Product would ask: "How long will this take?"

Engineering would say: "Given what we know right now and assuming nothing changes, probably six weeks."

Product would relay that to sales or CS with a bunch of caveats.

Sales would strip out the caveats and tell the customer: "We'll have it in six weeks."

Engineering would discover the requirements were incomplete or the problem was harder than expected or a dependency broke.

The timeline would slip.

Sales and CS would escalate: "You said six weeks."

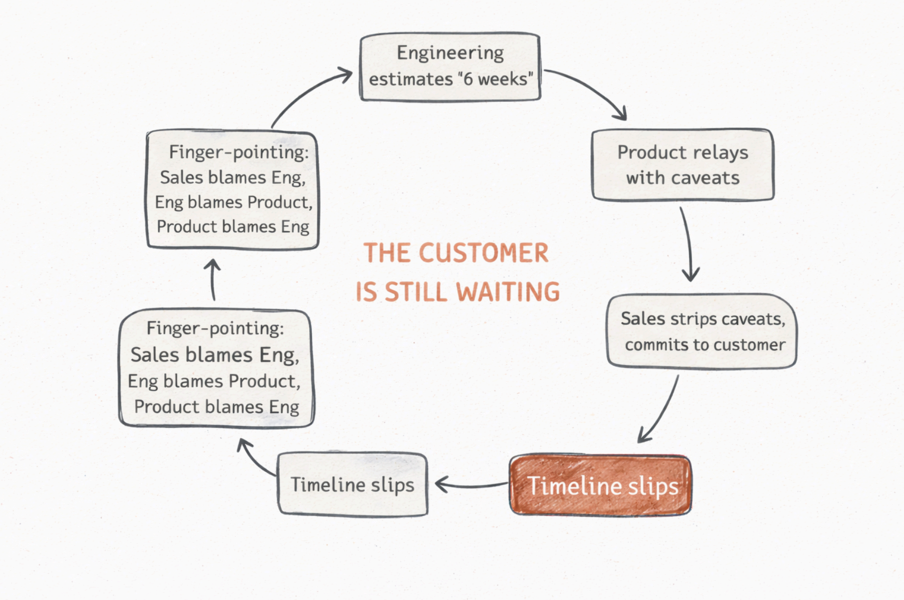
Engineering would say: "We said six weeks based on the information we had. The scope changed."

Product would say: "Why didn't engineering flag this earlier?"

Engineering would say: "Why did product give us incomplete requirements?"

Everyone would point fingers at everyone else, and the customer would still be waiting for their feature.

This is the classic conflict between these groups. And the root cause isn't that anyone is doing their job badly. It's that the system is broken.



Appetite doesn't eliminate this tension, but it reframes it honestly.

Those conversations could take on a different tone. Now we're having the conversation about the right thing: "We're going to go all-in for two weeks on that problem, and here is the scope and why we're confident that it gets you live." No longer is it the fantasy feature, but a real, calculated commitment.

A Better Outcome

Let me show you how this played out with a specific example, because it's easier to see how appetite works when you see it in action.

We had a prospect that was a mid-tier bank. There were some product gaps—features they needed that we didn't have yet. Everyone on my team was used to the old pattern: ask how long it'll take, estimate the work,

commit to a timeline, and then scramble when it takes longer than expected.

We applied appetites instead.

One of the gaps was a mass data import feature. The bank had historical data from their previous system, and they needed a way to import thousands of records in bulk when they went live.

Instead of asking "How long will it take to build everything the bank wants for data import?" we asked: "How much time is this worth?"

We set a two-week appetite.

Not because we thought it would take two weeks. But because that's what we were willing to invest in solving this problem for this customer at this stage of the product.

Then we handed that constraint to the engineer and said: "You have two weeks. What's the best solution you can deliver in that time?"

This is the key flip. The engineer wasn't guessing how long a predefined feature list would take. The engineer was designing a solution that could be delivered confidently within the time budget.

The engineer came back and said: "I'm not sure if this will work with that scope."

At this stage, the Product team and the Engineer went over each part of the scope and started seeing where it might be possible to get something in place that might handle the work. They came to the conclusion that if they could allow the process to run slowly, it would work, but it would probably take a couple of days for it to run. It would mean those accounts wouldn't all be live right away.

We evaluated that scope with the product team and the implementation team. Would it work? Would it be enough to get the bank live and operational?

Yes. It would.

So we shaped the pitch around that scope, bet on it at the betting table, and kicked off the work.

Two weeks later, the engineer delivered exactly what they'd promised.

The bank went live on time. No delays. No change requests extending the timeline. No finger-pointing about why it was taking longer than expected.

And here's the thing: if the bank later needed more sophisticated import features—batch processing, validation rules, custom field mapping—we could handle that. We could shape a follow-up pitch with a new appetite. But we didn't let perfect be the enemy of good enough to go live.

> ***"The two-week appetite forced us to solve the real problem, not build the fantasy version of the feature that looks good in a requirements document but never actually ships."***

As we ran more implementations with this approach, it became much easier to say: "These key product things can be delivered on time. It's just a matter of scope becoming the variable instead of time."

That shift—making scope flexible and time fixed—is uncomfortable at first. Especially for stakeholders who are used to walking into meetings with a feature checklist and asking when it'll all be done.

But it's honest. And in the long run, it builds more trust than estimation theater ever did.

Handling Reality

Now, here's the part nobody talks about in the Shape Up book, because Basecamp isn't running enterprise B2B deals with six-figure contracts.

What happens when a team uses up its appetite without finishing?

In the pure, dogmatic version of Shape Up, you throw away the work and move on. If it didn't fit in the appetite, the problem wasn't scoped correctly, and you start over.

That's not viable in enterprise.

If you've made a commitment to deliver a feature—especially if there's a customer contract tied to it—you can't just throw it out the window and say "we'll try again next cycle."

So you're going to be in a rough patch initially as teams get used to what's actually achievable within different appetite sizes. This is normal. It's part of the learning curve.

Here's how you handle it.

First, you use the cool-down period as a buffer. If you're running four weeks on, two weeks off, those two weeks can absorb a bit of overflow. Not every cycle. Not as a matter of course. But when a team is 80% done and needs a little more time to finish properly, the cool-down gives you that breathing room.

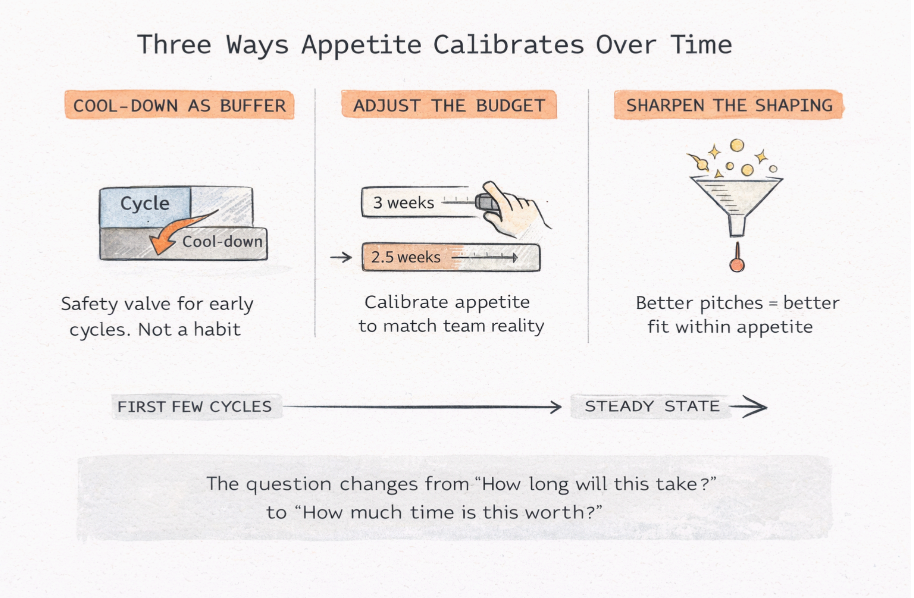
This isn't ideal. Cool-down has other purposes—technical health, exploration, recovery—and you don't want to consistently burn it on overflow work. But in the early cycles, when teams are still calibrating what fits in a two-week appetite versus a four-week appetite, cool-down acts as a safety valve.

Second, you adjust the total appetite budget over time. Maybe you initially thought a team had three weeks of capacity in a cycle, but you're consistently seeing work bleed into cool-down. So you shave the budget a bit. You call it 2.5 weeks of appetite to build in a buffer.

This isn't cheating. It's calibration. Different teams, different domains, different parts of the system have different levels of uncertainty. You're learning what your teams can reliably commit to, and you're adjusting the model to fit reality.

Third, you get better at shaping. As product people and engineers get more experience framing pitches with appetites, they get better at scoping work that actually fits. They learn what a two-week bet looks like versus a four-week bet. They get better at identifying the no-gos—the things that would balloon the scope and blow the appetite.

If you see patterns—the same team, the same types of work—that's a signal. Either the team needs more support learning to scope hammering



(cutting scope to fit time), or your appetite budgets for that team or that type of work need adjustment.

The goal isn't perfection. The goal is a system that learns and adapts. Where you're getting better at matching appetite to outcomes, and where teams trust that if they hit the unexpected, there's a way to handle it that doesn't blow up the entire roadmap.

The Strategic Shift

Here's what changes when you shift from estimates to appetites.

Your strategic conversations get better. Instead of debating whether something takes three weeks or five weeks, you're debating whether it's worth three weeks or five weeks. That's a debate about value and priorities, not a guessing game.

Your commitments get more honest. You stop promising dates based on guesses and start framing commitments around what you're willing to invest and what scope that investment will deliver.

Your teams get more autonomy. When you give a team an appetite and a problem, they have room to find the best solution within that constraint. They're not locked into a feature specification written before anyone understood the problem.

Your customers get better outcomes. Because you're solving the real problem, not building the feature list someone imagined before the work started.

But the shift is hard. It requires unlearning deeply ingrained habits. It requires sales and CS to communicate differently with customers. It requires leadership to accept that you can't predict the future, only make smart bets about where to invest time.

And it requires you to protect the integrity of appetites. If every time a team hits the appetite boundary, you just extend the timeline, you're back to estimation with a different name. The constraint has to be real for the system to work.

Try This

Pick one upcoming piece of work. Not the most critical bet-the-company project, but something meaningful. A feature, a technical improvement, a customer request.

Don't ask: "How long will this take?"

Ask: "How much time is this worth to us?"

Set an appetite. Two weeks, four weeks, whatever feels right based on the strategic value and the opportunity cost of not working on other things.

Then hand that appetite to the team and say: “You have X weeks. What’s the best solution you can deliver in that time?”

See what they come back with. See if the conversation changes. See if the outcome is different than it would have been with estimation.

One experiment won’t change your organization. But it’ll give you a data point. And sometimes that’s enough to start questioning whether the way you’ve always done it is actually the best way.

Chapter 3: Shaping for Enterprise

You've been in this meeting.

Engineering just delivered a feature. It took eight weeks instead of four. The scope ballooned. The customer who requested it is happy—but sales is asking why you built *that* instead of the three other features in the pipeline. Leadership wants to know what the ROI is. Product says they delivered what was in the spec. Engineering says the requirements kept changing.

Everyone's frustrated. And nobody's wrong.

This is what happens when you skip shaping.

Shaping is the most under-appreciated part of Shape Up. Most people who read the Basecamp book skim over the shaping chapters and jump straight to cycles, betting tables, and hill charts. Those are the visible parts—the parts you can demo in a meeting or point to on a dashboard.

Shaping is the invisible work that happens before anything gets built. And in enterprise B2B, it's the difference between a product organization that's strategic and one that's just reactive.

Here's why shaping matters more in enterprise than it does at a company like Basecamp.

At Basecamp, they're building for themselves and a broad market. They can make bets based on intuition, try things, learn, iterate. If a bet doesn't work out, it's not catastrophic.

In enterprise B2B, you're managing large customer contracts, complex sales cycles, and organizational stakeholders who need to understand why you're building what you're building. You can't just say "we're trying something" and expect everyone to be okay with it when there's a seven-figure deal on the line.

Shaping in enterprise isn't just about defining the problem and sketching a solution. It's about connecting the product work to business outcomes. It's about framing the work so that sales, customer success, leadership, and engineering can all see why this matters and what success looks like.

That requires a different level of rigor than what's in the Shape Up book. Not because the book is wrong—it's excellent. But because the context is different.

What Traditional Specs Miss

Traditional product specs and PRDs focus on what you're building. Requirements, user stories, acceptance criteria, wireframes, edge cases.

Shaping flips that. It starts with why you're building it and what you're willing to invest.

But in enterprise, you need more than a problem and a solution. You need business framing — the explicit connection between the product work and the economic outcomes. The revenue you're unlocking. The costs you're reducing. The strategic value you're creating. If you can't articulate it clearly, you probably shouldn't be building it.

An enterprise pitch has three layers that traditional specs miss: an at-a-glance summary that quantifies the bet, a product marketing blurb that tests whether you can explain it in plain language, and the business framing itself — the strategic argument for why this matters right now. Here's how I structured each one.

The At-a-Glance Table

At the top of every pitch, before you get into the problem or the solution, there's a table. One table. Fits on a single screen.

This table answers the question every stakeholder is thinking but might not ask out loud: "Why should we care about this?"

Here's what goes in it:

Revenue potential. What's the upside? Is this unlocking deals in the pipeline? Expanding existing accounts? Preventing churn? If it's new customer revenue, the data should come from the sales pipeline. If it's expansion or retention, it comes from customer success.

This isn't a fantasy number. It's grounded in actual pipeline data or customer conversations. And yes, sometimes the answer is "we don't know yet." That's fine. Say that. But if there's a known revenue impact, quantify it.

Cost savings. Is this reducing operational overhead? Cutting cloud costs? Eliminating manual work in billing or customer onboarding? If the revenue impact is low but this saves the business meaningful money, that's your ROI justification.

Again, quantify it. Even if it's approximate. "This should save us roughly 20 hours a week of manual work in CS" is better than "this will improve efficiency."

Effort level. This is your appetite. Two weeks. Four weeks. Six weeks. Whatever time budget you're assigning to this problem.

If it's something that requires multiple people—say, three engineers for four weeks—note that. The point is to make it clear what you're asking for in terms of capacity.

Strategic value. Is this a hygiene feature (competitors have it, we need it to stay competitive)? A differentiator (we're first to market with this, it positions us uniquely)? A platform capability (it unlocks a whole category of future work)?

This is qualitative, not quantitative. But it matters. Some work is worth doing even if the immediate revenue impact is unclear, because it's strategically important. Call that out.

Risk level. What could go wrong? Are there dependencies across teams? Is this based on early-stage pipeline data that might evaporate? Are you working with technology nobody on the team has used before?

Listing the risks forces you to think through them. And it sets expectations. If you bet on a high-risk pitch, everyone knows what they're signing up for.

Confidence. This is a rating on your estimates. How confident are you in the revenue potential? The effort level? The cost savings?

I added this field because otherwise, people would put down "$50 million in ARR" based on three deals in the pipeline that mentioned something vaguely related in an RFP. That's not a $50 million opportunity. That's a low-confidence guess.

When you force people to rate their confidence, they start being more honest about what they actually know versus what they're hoping.

Expected ROI. Based on the revenue potential (or cost savings) and the effort level, what's the payback? Do you need ten clients to sign up over twelve months? Do you need to win one seven-figure deal? How long until this pays for itself?

This is the math. And yes, at the beginning, it's more art than science. But you're building the muscle. The more you do this, the better your pipeline data gets, the more accurate your ROI estimates become.

This table is the executive summary of your pitch — the business case in one screen. It forces product people to think like business people, not just feature designers. But it's a summary, not the full argument. The deeper reasoning — why now, for whom, and how this fits in the market — comes in the business framing section that follows.

Here's what this looked like for the bank data import feature from Chapter 2:

Revenue potential: $450K (one mid-tier bank deal in final stages, data import identified as blocker in technical review)

Cost savings: Not relevant. This is revenue-driven

Effort level: 2 weeks, 1 engineer

Strategic value: Signing this bank means opening up a new segment for us

Risk level: Low (well-understood problem, no cross-team dependencies)

Confidence: High on effort, Medium on revenue (deal not yet signed)

Expected ROI: Break-even after first customer goes live; positive ROI within 90 days if deal closes

One table. One screen. Every stakeholder knows why this matters and what success looks like.

AT-A-GLANCE	
REVENUE POTENTIAL	Pipeline data, expansion, churn prevention
COST SAVINGS	Operational overhead, manual work eliminated
EFFORT LEVEL	Appetite in weeks × people
STRATEGIC VALUE	Hygiene / Differentiator / Platform
RISK LEVEL	Dependencies, unknowns, data confidence
CONFIDENCE	How sure are we? (High / Medium / Low)
EXPECTED ROI	Payback period, clients needed, deal size

The Product Marketing Blurb

Right after the at-a-glance table, include a short product marketing summary. Two to four sentences. Written as if a salesperson is explaining the feature to a prospect over coffee.

This isn't a tagline exercise. It's a forcing function.

If you can't describe the feature in plain language that a non-technical stakeholder would understand, the pitch isn't clear enough yet. If the

person shaping the work can't articulate what this means for the customer in a few sentences, the team building it will struggle too.

Here's what the product marketing blurb looked like for the bank data import feature:

> *"Our platform now supports direct bank data imports. If your operations team is spending hours manually reconciling transactions because our exports don't match your accounting system, that's over. Upload your bank's transaction format directly, and reconciliation happens automatically. No more spreadsheet gymnastics, no more end-of-month fire drills."*

That's it. No jargon. No technical architecture. Just: here's what it does, here's why you'd care.

This blurb serves double duty. First, it tests the shaping. If you can't write it, you don't understand the problem well enough. Second, it gives sales and customer success something they can actually use. When the feature ships, they don't have to translate a technical spec into customer language—it's already done. Marketing can pull from it for release notes. Sales can drop it into an email. Customer success can reference it in onboarding calls.

Write it during shaping, not after launch. By the time you're shaping, you know the problem, the customer pain, and the solution concept. That's exactly when this is easiest to write. If you wait until after the feature ships, you're reverse-engineering the story from the code, and it's never as crisp.

Business Framing

The at-a-glance table tells you *what* the numbers are. The product marketing blurb tells you *how* to talk about it. But neither answers the deeper question: why this, why now, and for whom?

That's what business framing does. It turns a feature request into a strategic argument. Without it, you end up with pitches that have solid numbers and a clear solution but no connective tissue explaining the business logic behind the bet.

Business framing has three parts: strategic context, job-to-be-done context, and market and competitive context. Not every pitch needs all three in equal depth — use judgment. But think through each one before deciding what to skip.

Strategic Context

This answers the question leadership always asks: why now?

Every organization has more ideas than capacity. The at-a-glance table shows the numbers, but strategic context explains the timing. What's converging that makes this urgent? What happens if you don't do this?

For the bank data import feature, the strategic context was straightforward: a mid-tier bank deal worth $450K was in final stages, their technical review flagged data import as a blocker, and no workaround existed. The "why now" wrote itself—lose the deal or build the feature.

Not every pitch has a ticking clock. But every pitch needs a clear answer to "what changed that makes this urgent now?" Be specific. "This has been on the backlog for a while" isn't strategic context. "Three enterprise prospects in the last quarter cited this as a blocker, our largest competitor now offers it as standard, and our CS team is spending twelve hours a week on manual workarounds" is.

Also include the strategic pillar this supports. If your organization has defined strategic themes—market expansion, customer retention, operational efficiency—link the pitch to one or two of them. This prevents the betting table from becoming a collection of disconnected bets.

Finally, define success metrics at this level. Not the detailed post-launch metrics (those come later in the pitch), but the headline numbers. For the bank data import: primary metric was closing the $450K deal. Secondary

metrics were time-to-onboard for bank clients and reduction in manual reconciliation hours. Targets were deal closed within 90 days, onboarding time cut by 60%, reconciliation hours reduced from five per week to under one.

These aren't commitments. They're hypotheses. You're saying: "If this pitch works the way we think it will, here's what we expect to see." That's what makes the accountability loop possible later.

Job-to-be-Done Context

Strategic context explains why the business should care. Job-to-be-done context explains why the customer should care.

This isn't about demographics or personas. It's about understanding the specific job the customer is trying to accomplish and why they can't do it well today.

Start with the customer job: what are they actually trying to do? Not "they want a bank data import feature." That's a solution, not a job. The job is: "reconcile transactions from their bank against platform data without manual effort, so they can close their books on time."

Then map the current workaround. How are they solving this today without your feature? This reveals the real pain. If they have no workaround, the pain is acute. If they have a clunky workaround, the pain is friction and wasted time. If they have a reasonable workaround, maybe this isn't as urgent as you think.

For the bank data import, the workaround was ugly: operations teams were exporting data from both systems into spreadsheets, manually matching transactions by timestamp and amount, and flagging discrepancies for investigation. Five hours a week, every week, with errors creeping in at month-end when volume spiked. The core frustration: they were forced to choose between accuracy and speed at month-end because the manual process couldn't keep up with transaction volume.

Finally—and this is the part most product people skip—map the forces at play. The jobs-to-be-done framework identifies four forces that determine whether customers will actually adopt what you build. Push is what drives them away from the current approach. Pull is what attracts them to something better. Anxiety is what makes them hesitate about switching. Habit is what keeps them doing things the old way even when it's painful.

Here's what the forces looked like for the bank data import:

Push: Manual reconciliation taking five hours weekly, errors increasing at month-end, operations team frustrated and asking for headcount.

Pull: Automated matching, one-click reconciliation, confidence in accuracy, operations team freed up for higher-value work.

Anxiety: Will the import handle our bank's specific format? What if transactions don't match? Do we need to change our accounting workflow?

Habit: Team has built spreadsheet templates over two years. They know the manual process cold. New hires are trained on it. "It works, it's just slow."

That force map tells you things the at-a-glance table never will. It tells you that you need to support the bank's specific format (not a generic one), that you need clear handling for non-matching transactions (or anxiety kills adoption), and that you need to make the new workflow simple enough that the team doesn't fall back to their spreadsheets. A few bullet points under each force is enough — but thinking through all four prevents you from building something that solves the pain perfectly but nobody adopts.

Market and Competitive Context

The third piece of business framing zooms out from the individual customer to the market. This matters most for pitches that are about competitive positioning, entering a new segment, or building a capability that multiple prospects are asking for.

Market opportunity. How many prospects or clients does this affect? Is this a niche need or a broad one? If your pipeline data shows that 75% of enterprise prospects need this capability, that's a very different pitch than one where a single client is asking for something custom.

Competitive landscape. What do competitors offer? Are you behind, at parity, or ahead? If a key competitor already has this capability and you're losing deals because of it, say that. If you'd be first to market with this approach, say that too. Ground it in specifics—name the competitors, describe what they offer, explain how prospects are comparing you.

Your unique angle. Even if competitors have something similar, what's your differentiation? This is where you articulate not just "we need this to compete" but "here's how our version will be better or different."

Positioning impact. How does building this change how the market sees you? Does it move you from "point solution" to "platform"? Does it open a new segment?

Not every pitch needs deep competitive analysis. For internal tooling, this section might be a single sentence: "Not applicable—internal capability." But for customer-facing features in a competitive market, this context is what separates a reactive feature request from a strategic bet.

For the bank data import, the market context was brief but important: two competitors offered something similar but required a rigid file format. Our approach of supporting the bank's native format was a differentiator. Three prospects in the pipeline had specifically asked about this during technical evaluations. The positioning impact was clear—it moved us from "fintech platform that requires banks to adapt" to "fintech platform that adapts to banks."

Putting It Together

Business framing doesn't need to be long. For a small pitch, it might be half a page. For a major capability bet, it might be two pages. The length should match the size of the bet and the complexity of the strategic argument.

What matters is that it exists. When the pitch goes to the betting table, the people deciding aren't just looking at a solution and an appetite. They're looking at why this matters to the business, why it matters to customers, and how it fits in the market. That's what makes the difference between betting on instinct and betting on insight.

The Pitch Structure

After the at-a-glance table, product marketing blurb, and business framing, the pitch follows a structure similar to what Shape Up describes, with a few enterprise-specific additions.

Problem statement. The business framing analyzed the customer's world — their job, their workaround, the forces at play. The problem statement narrows the focus: what specific trigger brought this to the shaping table, who exactly is affected, and how severe is the impact?

This should be concrete and scoped. Not "customers want better reporting," but "operations teams at mid-tier banks are spending 4-5 hours per week manually reconciling transactions because our current export doesn't include the fields they need for their accounting systems. This was flagged as a blocker in the technical review for the $450K mid-tier bank deal."

Appetite. How much time is this worth? Not how long it will take. How much time you're willing to invest.

Yes, this is in the table, but now include the rationale. Did you talk to an engineer? Did you slice this down from something bigger? Is this one cycle or two?

It's legitimate to have a pitch that spans two cycles. Some work is bigger. As long as the appetite doesn't stretch past two cycles, you're fine. If it's bigger than that, break it down or question whether it's actually one coherent bet or three separate bets bundled together.

Solution overview. The concept. The key elements. The fat marker sketch that shows the general shape of the solution without locking in the details.

This is where you stay at the right level of abstraction. You're not designing the UI pixel-by-pixel. You're not writing detailed technical specs. You're giving the team enough direction to understand what you're aiming for, but enough room to find the best way to get there.

Boundaries and constraints. This is critical.

First, rabbit holes. What are the things that would derail this? Perfect edge case handling. Premature optimization. Over-engineering for future scenarios. Redesigning tangential features that aren't core to the problem.

Call these out explicitly. Because if you don't, someone will go down one of these paths and blow the appetite.

Second, no-gos. What's explicitly out of scope? Hard boundaries. "We are not rebuilding the entire reporting engine. We are not adding real-time streaming. We are not integrating with every third-party accounting system in existence."

When you write down what you're not doing, it's easier to protect the appetite.

Third, must-haves versus nice-to-haves. Since time is fixed, scope is variable. What are the core elements that have to be there for this to work? What are the things that would be great to include if there's time, but we can ship without them?

This gives the team a clear prioritization framework. If they're two weeks in and realizing the appetite was a bit tight, they know what to cut.

Dependencies and risks. Are there technical dependencies on other teams? External factors that could change? If you can identify de-risking strategies up front, note those. But at minimum, make the risks visible.

Success criteria and validation. How do you know this is done? Not just "the code is merged," but "the pilot customer is using this in production and the reconciliation workflow is working."

Then, post-launch metrics. One to three months after launch, what are you checking? Did we sign the deal? Did we expand the account? Did churn improve?

And long-term success. Six months or a year out, you're going back to that expected ROI. Did it happen? If not, why not? Was the pipeline data wrong? Was the feature harder to sell than expected? Was the onboarding more complex than anticipated?

This is the accountability loop. You're not just shipping features and moving on. You're checking whether the work delivered the outcomes you expected. And if it didn't, you're learning from that.

Anatomy of an Enterprise Pitch

SHAPED PITCH

AT-A-GLANCE TABLE

Business Framing — Business Framing

BUSINESS FRAMING

Really why this is important.

PROBLEM STATEMENT — Customer pain, frequency, who's affected

APPETITE — Time budget + rationale (max 2 cycles)

SOLUTION OVERVIEW — Fat marker sketch, key elements, right level of abstraction

BOUNDARIES — Scope protection

- Rabbit Holes
- No-Gos
- Must-Haves vs Nice-to-Haves

Scope protection

DEPENDENCIES & RISKS — Cross-team, external, de-risking strategies.

SUCCESS CRITERIA

- Launch metrics
- 1-3 month check
- 6-12 month ROI review

We return to this connection in Chapter 12, where we explore how AI dramatically accelerates the shaping process. ## From Roadmap to Pitch

If pitches are shaped independently and the roadmap is organised around themes, how do the two connect?

This is worth answering explicitly, because without a clear link, you end up with pitches that are well-shaped but strategically disconnected — or roadmap themes that sound good but never translate into concrete work.

Here's how I do it.

The thematic roadmap has overarching themes — the problems you're solving at a strategic level. Something like "Making Payments 40% Faster." That theme might live on the roadmap for two or three quarters, depending on the scope.

Underneath that theme, you shape individual pitches that tackle specific problems contributing to the theme:

- Q1 pitch: Add another payment provider to create faster settlement options (appetite: 4 weeks)
- Q2 pitch: Give customers a single-click-to-buy option that eliminates checkout friction (appetite: 6 weeks)

Each pitch goes through the full shaping process — at-a-glance table, problem statement, boundaries, the works. But the theme on the roadmap is the connective tissue. It's what you show stakeholders when they ask "where are we going?" The pitches are what you bring to the betting table when you ask "what are we building next?"

In practice, I link pitches to roadmap themes in whatever project management tool the team uses. The mechanism doesn't matter much — it could be a tag, a parent epic, or a column on a board. What matters is that the link exists and is visible, so anyone can trace a pitch back to the strategic theme it serves.

The same principle applies to how engineering tracks their work. If they use epics and tasks in their tooling, link one or more epics to the pitch. I get my tracking at the roadmap and pitch level; they get flexibility in how they plan and execute. Nobody's fighting over tool choices or process — the connection point is the pitch, and everyone works outward from there.

The Accountability Shift

This is where things get uncomfortable for product teams.

When I introduced this pitch template—the at-a-glance table, the business framing, the ROI checks—the pushback wasn't about the format or the structure. It was about the accountability.

Product people are used to being asked what features to build. They're used to writing specs, running discovery, gathering requirements. They're used to debating whether the button should be blue or green.

They're not used to being asked: "What's the ROI on this? How do you know it's worth building? And can we check that six months from now?"

That's a different bar. And the initial resistance is real.

It's not because product people can't do this work. It's because they've never been asked to. Nobody ever checked whether the color of the button actually made economic sense.

When you start asking product people to think about revenue impact, cost savings, and ROI, you're asking them to operate at a different level. You're asking them to connect their work to the business in a way they probably haven't had to before.

Some people love this. It makes the work more meaningful. They're not chasing random feature requests anymore. They're building things that matter, and they can prove it. They start proactively going to sales calls instead of being dragged there.

Some people resist it. Because it's a change. Because it's uncomfortable. Because they're not sure they'll get it right.

If you're introducing this in your organization, expect both reactions. And give people time to adjust.

Run a few cycles where you work with them to fill out the at-a-glance table. Help them understand why the ROI matters. Show them that when they can articulate the business impact, their pitches get bet on more often. And when they go back and check the metrics, they get better at predicting what will work.

Over time, product people start to see the benefits. They're doing less random work. They're building things that actually move the needle. And when they go into meetings with sales, customer success, or leadership, they can speak the same language.

The product person can talk to sales about pipeline. They can talk to leadership about strategic value. They can talk to engineering about technical feasibility. They're piecing everything together instead of just being the middleman who writes tickets.

This eliminates the traditional fight where product wants to build something, sales wants to sell something different, and engineering is stuck in the middle wondering what they're supposed to prioritize.

> ***"When everyone's looking at the same pitch with the same business framing, those fights get a lot quieter."***

Who Does the Shaping?

The question I get asked most often: who's responsible for shaping? Is it product managers? Designers? Engineers? Leadership?

The answer: it depends on your organization. But here's what I've seen work.

In most enterprise B2B organizations, product people do the bulk of the shaping. They're sitting in sales calls. They're in customer success meetings. They're doing market research. They're talking to engineering about feasibility.

They're already gathering the information. They're just not necessarily formulating it into a pitch with all the business framing.

That said, the best pitches come from collaboration. A product person shapes the initial pitch, but they're talking to an engineer early to gut-check the appetite. They're pulling data from sales or CS to ground the revenue impact. They might work with a designer to sketch the solution at the right level of abstraction.

Shaping isn't a solo activity. But someone has to own it. Someone has to pull it together into a coherent pitch that can go to the betting table.

In some organizations, senior engineers shape technical infrastructure pitches. That's fine. As long as they're also doing the business framing—connecting the work to the outcomes—it works.

The skill set required isn't magical. It's not something you need a specific degree or certification for. It's the ability to think strategically about problems, communicate clearly, and connect product work to business outcomes.

If you can do that, you can learn to shape.

That said, when you first introduce structured shaping, you'll hit friction.

Product teams without business fluency. If your product people have never analyzed pipeline data or calculated ROI, they're going to struggle with the at-a-glance table. This is a skill gap, not a will gap. Pair them with someone from sales or finance for the first few pitches. Show them where to find pipeline data in your CRM. Teach them how to estimate cost savings. This is coaching, not criticism.

Engineers who don't want to shape. Some engineers love shaping—it gives them strategic context and agency. Others see it as "product's job" and resist getting involved early. If you're trying to get senior engineers to shape infrastructure pitches, you'll need to make the case: shaping isn't extra work, it's the work that prevents rework. It's thinking before building. Frame it as a leverage activity, not a distraction.

Sales skepticism about product-led framing. Sales teams may push back on product people representing pipeline data. "But it's not accurate enough. It's being misrepresented." Product drafts the pitch, sales reviews the revenue potential field and pressure-tests the assumptions. Sales doesn't own the pitch, but they validate the business framing. This builds trust over time.

These friction points are normal. Don't let them derail the process. Work through them with coaching, collaboration, and patience. Over a few cycles, the muscle builds.

When Shaping Isn't Enough

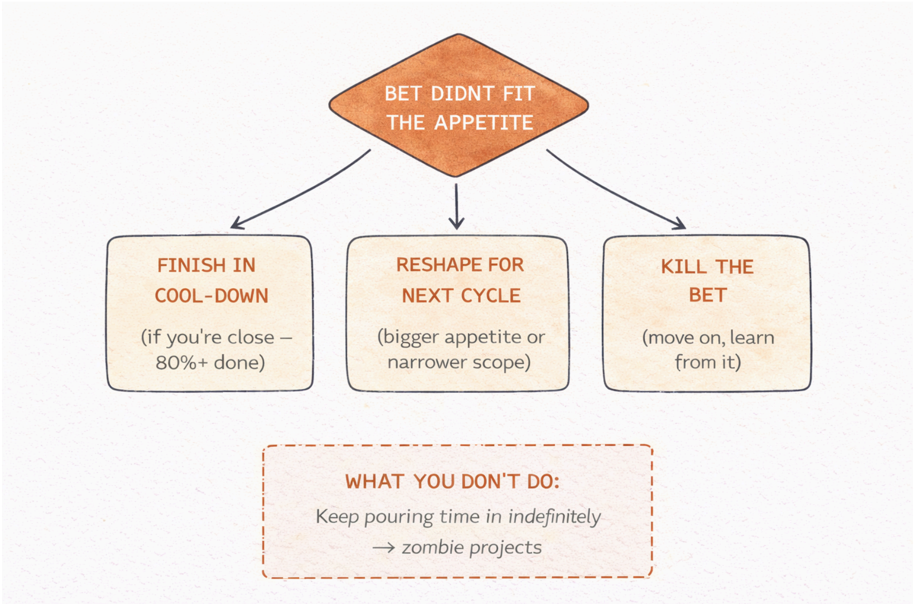
Here's the honest part: sometimes you bet on a poorly shaped pitch and it goes sideways.

Maybe the problem wasn't as well understood as you thought. Maybe the appetite was too small. Maybe the dependencies were more complex than they appeared.

When that happens, you have a few options.

You can use cool-down to finish the work (if you're close). This is my go-to option. You can reshape the pitch for the next cycle with a bigger appetite or a narrower scope. Or you can kill the bet and move on.

What you don't do is keep pouring time into a poorly shaped pitch indefinitely. That's how you end up with zombie projects that drain capacity for months without delivering value.



The betting table is where you decide what to work on. But shaping is what determines whether those bets are winnable. If you're consistently betting on pitches that blow their appetite or miss the mark on outcomes, the problem isn't the cycles. It's the shaping.

Get better at shaping, and everything downstream gets easier.

Try This

If you're not already using a structured pitch format, here's an experiment.

Pick an upcoming piece of work. Something meaningful but not mission-critical.

Before you write the spec or the user stories, fill out the at-a-glance table. Revenue potential. Cost savings. Effort level. Strategic value. Risk level. Confidence. Expected ROI.

See if you can articulate the business framing. Why this matters. What job the customer is trying to do. What success looks like.

Then shape the solution. Problem statement. Appetite. Solution overview. Boundaries and no-gos. Must-haves versus nice-to-haves.

Don't present it to the betting table yet. Just write it. See what it reveals about the work.

My guess: you'll discover gaps. Things you thought were clear that aren't. Questions you can't answer yet. Risks you hadn't thought through.

That's the point. Shaping forces you to think before you build. And in enterprise, where every bet carries significant opportunity cost, thinking first is worth the investment.

Chapter 4: The Two-Horizon Betting Table

The betting table is where strategy meets reality.

It's the moment when you stop talking about what could be built and start committing to what will be built. It's where the thematic roadmap, the shaped pitches, and the appetite budget come together into actual decisions.

And it's where most organizations struggle.

Not because betting is hard conceptually. The idea is simple: you look at the shaped work, you look at your capacity, you decide what to bet on for the next cycle.

The struggle comes from two sources.

First, enterprise organizations have more stakeholders with competing interests. Sales wants the features that close deals. Customer success wants the features that reduce churn. Product wants to build the platform capabilities that unlock future growth. Engineering wants to pay down technical debt before it becomes a crisis.

Traditional prioritization tries to make everyone happy and ends up making no one happy. You get a roadmap that's a mile wide and an inch deep. Everything is "in progress" but nothing ships.

Second, enterprise deals don't move on six-week cycles. A large customer opportunity might develop over three months. An RFP might take six weeks just to respond to. A regulatory requirement might have a deadline nine months out.

If your betting table only looks one cycle ahead, you're constantly reacting. You don't have time to shape the work properly. You don't have visibility for sales to communicate with customers. You can't coordinate dependencies across teams.

The standard Shape Up betting table—shaped work goes in, bets come out, one cycle at a time—works beautifully for Basecamp. It doesn't quite fit enterprise B2B.

So I adapted it. Not because Shape Up is wrong, but because the context demands something slightly different.

Two Horizons

The betting table I ran had two time horizons.

Horizon 1 is a triage check on the upcoming cycle. It's the first ten minutes of the meeting. The question is simple: is anything on fire that requires us to disrupt what's already in flight?

Most of the time, the answer is no. And you move on.

When the answer is yes, you're looking at a genuine emergency. A customer-critical bug that can't wait. A contract-dependent feature that needs to ship or the deal falls apart. A competitive threat that just emerged and requires an immediate response.

These disruptions should be rare. Very rare. If you're triggering Horizon 1 disruptions every other cycle, the problem isn't the betting table. The problem is your system for separating strategic work from reactive work. We'll get to that in Chapter 8.

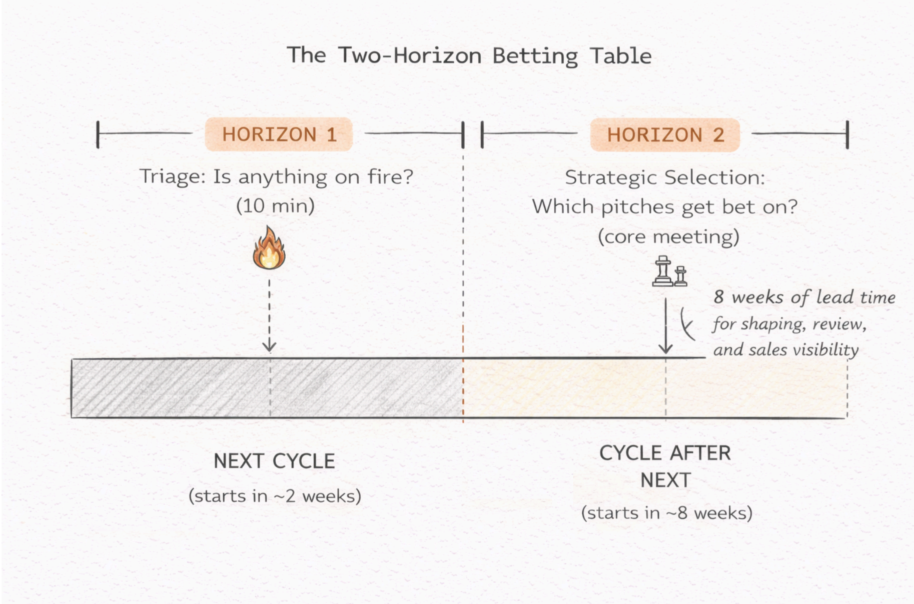
Horizon 2 is the strategic selection for the cycle after next.

Not the cycle that starts in two weeks. The one that starts eight weeks from now.

This is different from standard Shape Up, which bets one cycle ahead on a tighter turn. But in enterprise, you need that additional buffer. You need time for the pitches to be fully shaped. You need time for engineering to review the technical feasibility. You need visibility for sales to have conversations with customers about what's coming.

Horizon 2 is where you invest the time. It's where you look at the shaped pitches, the appetite budget, the strategic priorities, and make the selection.

This is the core of the betting table. And when it's set up properly, it's surprisingly lightweight.



The Mechanics

Let's walk through what a betting table session actually looks like.

You're in week two or three of the current cycle. The team running the current bets is heads-down executing. You're not bothering them.

The betting table convenes. This is a small group. In my organizations, it was typically the head of product, the head of engineering, and sometimes a tiebreaker (me, when I was founder/CTO, or someone else with strategic authority).

Customer success might be there. Sales might be there. It depends on the organization. But the core group should be tight. Three to five people max.

If you need ten people in the room to make a decision about what to build, your pitches aren't shaped well enough or your strategic priorities aren't clear.

Horizon 1: The Triage Check

First ten minutes. The question on the table: does anything need to change in the short-term?

If nothing's on fire, you're done. Move to Horizon 2.

If something is on fire, you discuss it. What's the disruption? What's the impact if we don't act? What gets displaced if we pull capacity away from the current bets?

You decide. Either the disruption is significant enough to justify the chaos, or it's not. If it's not, it waits. If it is, you make the call and communicate the change to the team immediately.

In six years of running this model, I triggered Horizon 1 disruptions twice.

Twice.

Both times were large-customer-driven. Both times were predictable—we saw them coming weeks in advance. Both times, the disruption was justified by the ARR impact.

I'll walk through one of those in a moment. But first, let's talk about Horizon 2, because this is where most of the work happens.

Horizon 2: Strategic Selection

Horizon 2 is about selecting the bets for the cycle that starts eight weeks from now.

By the time you're in this meeting, the work is already done.

The pitches have been shaped. They've gone through the template from Chapter 3: at-a-glance table with business framing, problem statement, appetite, solution overview, boundaries, success criteria. The whole thing.

Engineering has reviewed them. Sales has provided pipeline data. Customer success has weighed in on churn risks or expansion opportunities. Product has done the market research and competitive analysis.

All of that input is baked into the pitch. So when you sit down at the betting table, you're not starting from scratch. You're looking at fully formed pitches with clear business justification.

Here's how the selection works.

You start with the appetite budget. This is the total capacity available for the next cycle. If you have three engineers and each works four weeks (we'll get into the appetite-weeks calculation in Chapter 5), you have 12 weeks of appetite to allocate.

That's your starting point. That's your constraint.

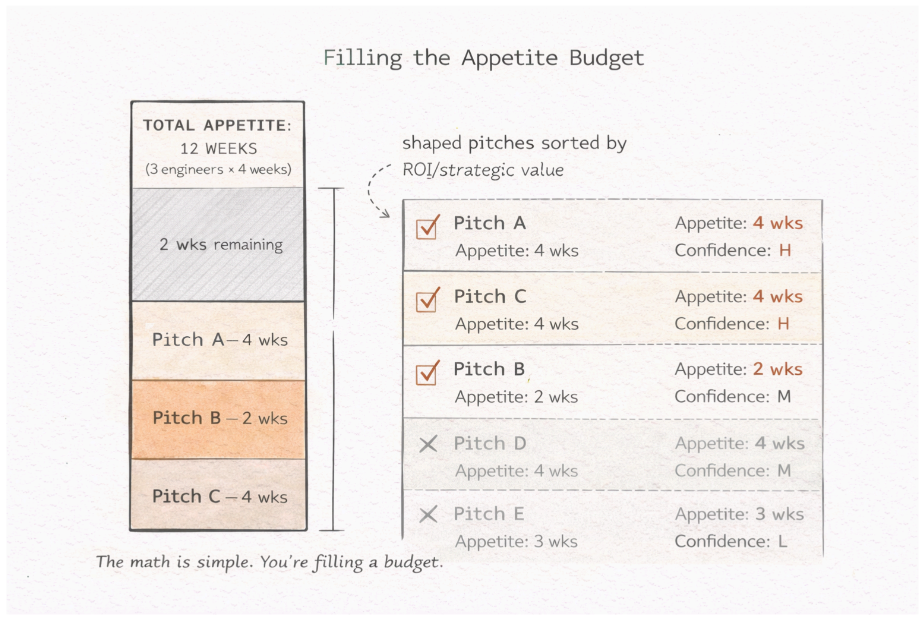
Now you look at the shaped pitches. Each pitch has an appetite cost. This one is two weeks. That one is four weeks. This other one is six weeks but requires two people, so it costs 12 weeks of appetite.

The math is simple. You're filling a budget.

If the at-a-glance tables are well-constructed—and they should be, because that's the whole point of Chapter 3—you can make the first cut without even reading the full pitch.

Which pitches have the highest confidence? The highest ROI? The strongest revenue potential or cost savings? The most strategic value?

Sort them. Start filling the budget. When you're out of capacity, you stop.



The math is simple. You're filling a budget.

You make the call.

Sometimes, you don't even need a meeting. The selection is obvious. The pitches with the strongest business cases, the clearest strategic fit, the highest ROI—they rise to the top naturally.

When I was founder/CTO, the betting table ran quite often without me. The team knew the strategic priorities. They knew how to read the at-a-glance table. They knew how to evaluate pitches.

I only showed up when there was a tie that needed breaking or when someone wanted a strategic check on a particularly large or risky bet.

If your betting table requires the founder or CEO to attend every session to make decisions, something's broken. Either the pitches aren't framed well enough, or the strategic priorities aren't clear enough, or the team doesn't have the authority to make the call.

Fix those things. Don't solve them by adding more people to the meeting.

When Your Pitch Doesn't Get Selected

Here's the thing nobody warns you about when you implement this system: at some point, someone's going to have a pitch they care deeply about, and it's not going to get bet on.

And they're going to be unhappy.

In traditional prioritization, that unhappiness turns into lobbying. They go to their VP. They escalate to the CEO. They try to get their feature added to the roadmap through sheer force of will.

That doesn't work here. Because the economic drivers are visible.

When a stakeholder's pitch doesn't get selected, you can show them why. "Here's the pitch we bet on instead. Here's the ROI comparison. Here's the strategic value. Here's the confidence level in the revenue impact."

They might not like it. But they can't argue that the decision was arbitrary or political.

And here's what happens over time: people get better at pitching.

If someone puts forward a pitch with inflated ROI numbers—padding the revenue potential to make their case look stronger—and it gets bet on, you go back and check the metrics later. Did the revenue materialize? Did the cost savings happen?

If the answer is no, you trace it back. You ask why the estimates were off. You hold people accountable for the framing they provided.

This doesn't need to be punitive. It's a learning loop. But it does need to happen, because otherwise, people will game the system.

When everyone knows the metrics will be checked, the quality of the pitches improves. People stop padding numbers. They start being more honest about confidence levels. They start doing the work to gather real pipeline data instead of guessing.

And when a pitch doesn't get selected, the feedback is clear: "Come back next cycle with better data, or a stronger business case, or a different framing that makes the strategic value clearer."

> ***"That's a conversation you can have productively. It's not personal. It's not political. It's economic."***

The Rules for Disruption

Here's what I learned about Horizon 1 disruptions: you need clear rules, or the exceptions will become the norm.

In enterprise, there's always a reason to disrupt. A customer with an urgent request. A deal that's almost closed but needs one more feature. A bug that's annoying but not critical. A competitor announcement that feels threatening.

If you allow disruptions every time someone has a compelling story, you're back to the chaos you were trying to escape. Teams don't know what they're building next. Sales can't communicate with customers. The roadmap becomes fiction.

So you need a threshold.

For me, the threshold was ARR-based. A disruption had to be tied to a significant revenue impact—either new ARR that would materially change the business, or churn risk that would materially harm it.

How significant? That depends on the size of your company. But the threshold should be high enough that it's genuinely rare.

If you're a $10 million ARR company, maybe the threshold is $500k. If you're a $100 million ARR company, maybe it's $5 million. The exact number matters less than the principle: disruptions are expensive, so the juice has to be worth the squeeze.

And here's the other rule: the disruption has to be predictable.

I know that sounds contradictory. How can an emergency be predictable?

Let me show you.

The Credit Card Issuer Story

About three years into running this model at one of my companies, we were going through an RFP process with a large credit card issuer. This was a seven-figure deal. Potentially eight figures over the life of the contract.

The RFP had specific requirements. Some of them were gaps in our product. We knew this going in.

As we progressed through the RFP, we started shaping the pitches for those gaps. Not because we'd won the deal yet—we hadn't. But because we knew the features they wanted, and we knew that if we won, we'd need to deliver on them.

So the pitches were already being developed. The at-a-glance tables were already being filled out. The business framing was already being done, all tied to this particular customer.

As we got closer to signing the contract, we had more visibility. We knew what the disruption would look like. We knew which current bets would need to be displaced. We knew the capacity trade-off.

By the time we signed the contract and said "okay, now we need to execute on this," we didn't scramble. We already had the pitches shaped. We already knew what would fit in which cycle. We already knew the timeline.

We triggered a Horizon 1 disruption. We adjusted the next cycle to include the highest-priority features for this customer. We communicated the change to the team, to sales, to the rest of the organization.

And because we'd been preparing for weeks, it wasn't chaos. It was a controlled adjustment.

That's what I mean by predictable disruption. The trigger was a large customer contract. But the work leading up to it—the shaping, the

framing, the planning—meant that when the disruption happened, it was manageable.

If this had come out of nowhere, it would have been a disaster.

But it didn't come out of nowhere. We had visibility. We had shaped pitches. We had clear rules about when disruption was acceptable.

In six years, this happened twice. Both times, large customer-driven. Both times, predictable.

> ***"The system worked because disruptions were the exception, not the rule."***

Now let me show you the second type of disruption—smaller in scale, but worth understanding because it demonstrates how the system handles dynamic opportunities.

The Revenue Expansion Swap

We had a cycle selected. Two small pitches, already shaped, ready to go. Total appetite cost: maybe six weeks across the team.

Four weeks before the cycle kicked off, customer success came to us with an opportunity. An existing customer—a mid-tier bank—wanted to launch our platform in a new market. But they needed one additional feature to meet a regulatory requirement in that market.

If we could deliver it in the next cycle, they'd expand the contract immediately. If we couldn't, they'd wait six months and maybe look at competitors.

The pitch was already shaped. It had been in the pipeline for a future cycle. We just hadn't prioritized it yet.

We looked at the at-a-glance table for this pitch. We looked at the two smaller pitches already selected. We compared the ROI.

The revenue expansion was higher. Not by a huge margin, but enough to matter.

We made the swap. Pulled the two smaller pitches. Bet on the revenue expansion feature instead.

Then we went back to the stakeholders who'd been expecting the smaller pitches and explained the trade-off. "Here's why we're making this swap. Here's the revenue impact. Here's when we'll revisit the pitches we're deferring."

Because the framing was clear—because the at-a-glance tables made the economic logic visible—the conversation wasn't contentious. It was straightforward.

Sales understood why. Product understood why. Engineering understood why. The stakeholders who'd been advocating for the smaller pitches weren't thrilled, but they got it. The revenue expansion was the right call.

This is what transparency does. It takes the politics out of prioritization.

Those two disruptions—the credit card issuer and the revenue expansion swap—show the system in practice. Now let's zoom out and look at what makes this model work consistently across cycles.

Making It Work

Here's what makes a betting table lightweight and effective.

First, do the shaping work up front. If pitches show up at the betting table half-baked, the meeting devolves into a working session where you're trying to shape and decide at the same time. That doesn't work.

Shape first. Then bet.

Second, establish clear strategic priorities. If every pitch looks equally important because you don't have a shared understanding of what the company is trying to accomplish this quarter, the betting table becomes a negotiation. That's exhausting.

When the strategic priorities are clear, most of the bets select themselves.

Third, keep the group small. Three to five people. If you need more, you're trying to solve organizational problems with a meeting. Don't.

Fourth, protect Horizon 1 from becoming a regular escape hatch. If you're disrupting every cycle, you don't have an emergency. You have a broken system. Fix the system, don't normalize the disruptions.

Fifth, use the two-horizon model to give sales and customer success visibility. They can't communicate with customers if they don't know what's coming. Horizon 2 gives them nine weeks of visibility. That's enough to have meaningful conversations without over-committing.

When all of this is in place, the betting table becomes the easiest meeting you run. You're not arguing about priorities. You're not negotiating with stakeholders. You're looking at shaped work, applying clear criteria, and making decisions that everyone can understand.

It shouldn't take more than an hour. Often, it takes less.

Try This

If you're not already using a betting table model, here's a low-stakes way to try it.

1. Pick an upcoming planning session. Don't change your whole process yet. Just try this experiment within it.
2. Before the meeting, ask the people proposing work to fill out the at-a-glance table from Chapter 3. Revenue potential. Cost savings. Effort level. Strategic value. Risk. Confidence. Expected ROI.
3. Then, in the meeting, start by comparing those tables. Don't jump into discussing the solutions. Just look at the framing.
4. Which pieces of work have the strongest business case? The clearest strategic fit? The highest confidence?

5. See if that changes the prioritization conversation. See if it makes the trade-offs more visible.

You don't have to restructure your entire planning process after one experiment. But my guess is you'll discover that the conversation gets better when the economic logic is transparent.

And once you see that, it's hard to go back to prioritization by loudest voice.

Chapter 5: Appetite-Weeks

The velocity game never ends well.

You've probably played it. The team delivers ten story points one sprint, twelve the next, eight the one after that. Someone suggests averaging the velocity. Someone else suggests weighting recent sprints more heavily. Someone points out that the team composition changed. Someone notes that last sprint included a holiday. The product manager wants to know when the feature will ship. The engineering manager can't give a straight answer because the math keeps moving.

Velocity-based planning feels scientific. It's not. It's estimation theater wearing a lab coat.

Here's what worked instead: appetite-weeks.

It's embarrassingly simple. You have people. You have weeks in a cycle. Multiply them together. That's your capacity budget for the cycle. Pitches cost appetite-weeks. Fill the cycle like you're filling a budget. When you're out of capacity, you're done betting.

No velocity tracking. No burn-down charts. No sprint retrospectives analyzing why the team "only" delivered eight points instead of ten. Just arithmetic.

The Formula

People × 4 weeks = total appetite-weeks.

That's it.

A cycle is six weeks total: four weeks of building, two weeks of cool-down. The cool-down doesn't count toward capacity — it's recovery time, exploration time, technical health time. The building weeks are what you budget with.

If you have four engineers available for the full four-week build period, you have sixteen appetite-weeks to allocate. If one of those engineers is on vacation for a week, you have fifteen. If another is covering firefighting duties, you might allocate just one appetite-week from them instead of four.

The math adjusts naturally for reality.

Vacation? Subtract the weeks. Someone leaving mid-cycle? Subtract their contribution. New hire ramping up? Add them at half capacity for their first cycle. Engineering managers splitting time between coding and management? Count them at two appetite-weeks instead of four.

You're not estimating productivity. You're counting available time and using appetite as the unit of measurement.

A Real Cycle's Allocation

Let me walk you through how this actually worked.

We had a rule in the engineering team: request vacation at least nine weeks in advance. That gave visibility into the next cycle or two. Usually people requested further ahead, which made planning even easier.

About a week before the betting table, the engineering team would tally up capacity. They'd look at who was available, who was taking time off, and which weeks people would be present.

Here's a simplified example from an actual cycle:

- Engineer A: Full four weeks available = 4 appetite-weeks
- Engineer B: One week vacation = 3 appetite-weeks
- Engineer C (Engineering Manager): Full time present, but splitting duties = 2 appetite-weeks
- Engineer D: Assigned to firefighting/support = 1 appetite-week
- Engineer E: Full four weeks available = 4 appetite-weeks

Total capacity: 14 appetite-weeks

That's the budget we brought to the betting table. We knew exactly what we had to work with before we walked into the room.

Now, you might look at that and think, "Can't you just pile fourteen weeks of work on any one person?" Technically, yes. Realistically, no. If a pitch required deep expertise in a specific area, you'd probably allocate most of it to the person with that expertise. But product team members could look at the capacity breakdown in advance and plan accordingly.

Sometimes a pitch would need two weeks from a specialist and two weeks from the general pool — whoever could grab it, grabbed it. The simplicity of the model made those conversations fast. No one was debating story points or velocity trends or whether a task was a five or an eight. It was just: "This costs four appetite-weeks. Do we have four appetite-weeks left? Yes or no."

Costing a Pitch

Pitches came to the betting table with an appetite already assigned: two weeks, four weeks, six weeks (though six-week pitches were rare and usually a sign that the shaping wasn't tight enough). We would also allow 1/2 week once we got better at all this. Or we allowed a general improvement pitch that included a batch of small-scoped items.

The appetite wasn't an estimate of how long the work would take. It was a strategic decision about how much time it was worth. That shift — from prediction to constraint — changed everything.

When you're filling the cycle, you're matching pitch appetites against available appetite-weeks. If you have fourteen appetite-weeks and three pitches costing four, six, and four appetite-weeks respectively, you're at exactly fourteen. Done. Cycle filled.

If you have fourteen appetite-weeks and four pitches costing four, four, four, and six, you have to choose. That's the point. The capacity constraint forces real prioritization.

The Moment of Truth

The betting table gets interesting when you're running out of room.

You've allocated eleven appetite-weeks. You have three left. The next pitch on the table costs four.

What do you do?

You either skip that pitch and look for something smaller, or you go back and re-examine what you've already bet on. Maybe one of the earlier bets isn't as critical as you thought. Maybe you can swap it out for the four-week pitch and find a two-week bet instead.

This is where the transparency of appetite-weeks pays off. There's no ambiguity. You're not debating whether the team "might" be able to squeeze in one more feature if they work weekends. You're looking at simple arithmetic and making trade-offs in daylight.

The at-a-glance table from Chapter 3 makes this even cleaner. You can see the revenue potential, the strategic value, the confidence level for every pitch. When you're deciding whether to swap a four-week bet for two smaller ones, you're not guessing. You're making an informed trade-off based on business framing.

That's why the betting table ran fast. No politics. No lobbying. Just data, appetite constraints, and decisions.

When a Pitch Costs More Than Expected

Here's the uncomfortable truth: sometimes you get it wrong.

You bet on a four-week pitch and three weeks in, the team realizes it's going to take six. What now?

If you've shaped the pitch properly — with must-haves and nice-to-haves clearly separated — you start cutting scope. The nice-to-haves get stripped out. They become a separate pitch in the next cycle. They were actually nice-to-have and you never build them at all.

That doesn't always work. Sometimes there's too much in the must-haves. Sometimes the problem was more complex than anyone realized during shaping.

In that case, you have a few options.

First, shuffle other work. If another pitch in the cycle is ahead of schedule or can be scoped down, you reallocate appetite-weeks. This is rare, but it happens.

Second, use the cool-down as a buffer. I know, I know — the cool-down isn't supposed to be for overflow. But early on, until you get the rhythm dialed in, it's your emergency reserve. If one big pitch explodes and bleeds into cool-down, that's not ideal, but it's not catastrophic either.

What you can't do is let this become the norm.

If you're bleeding into cool-down every cycle, something's broken. Either your shaping process isn't rigorous enough, or you're systematically under-scoping appetites, or there's a skills mismatch on the team. You need to diagnose and fix it, not just accept cool-down as overflow time.

We had cycles where we used some of the cool-down to finish a pitch. It happened. The key was that it was the exception, not the rule, and everyone understood why it happened.

One other thing: if a pitch is genuinely massive — something that would take twenty-four appetite-weeks across two cycles — you need to break it down. Shape Up is designed for work that fits into one or two cycles. If you're looking at something bigger, you're not shaping at the right level of abstraction.

You can have a theme in the roadmap that frames the larger strategic problem, and then individual pitches underneath that theme solve specific pieces of it. That way you're shipping value every cycle, not betting on a six-month monolith and hoping it works out.

Why Teams Don't Game It

You might be wondering: did anyone ever inflate their appetite estimates to give themselves breathing room?

No. And here's why.

Shape Up eliminates most of the tension that causes estimation games in the first place. In a traditional Agile setup, the product team pushes for faster delivery and the engineering team pushes back with higher estimates. It's adversarial. Engineering inflates estimates because they don't trust product to give them enough time. Product deflates estimates because they don't trust engineering to work efficiently.

Appetite flips that dynamic.

The product team isn't asking engineering to estimate. They're saying, "This is worth four weeks. Can you solve the problem in four weeks?" If the answer is no, the conversation isn't about negotiating timelines. It's about negotiating scope. What can we cut? What's the simpler version? What are we actually trying to solve here?

That builds trust.

The other reason teams didn't game the system: the cool-down. If they made a mistake — if they genuinely misjudged the complexity of a pitch — they could use the cool-down as a buffer. They weren't going to get penalized the way they would in a sprint-based system where missing a deadline feels like failure.

So instead of inflating appetites defensively, engineers would have honest conversations about scope. If a product person wanted to squeeze one more tiny feature into a four-week pitch, the engineer could say, "That thing is tiny, but it doesn't fit in the scope we shaped. Put it in as a nice-to-have, and if we have time, we'll get to it."

And product would do it. Because the system was transparent. Because the framing was clear. Because everyone trusted that the process wasn't designed to squeeze every last ounce of productivity out of the team.

We had more problems with people being upset when we did bleed into cool-down than we ever had with people gaming appetites.

> ***"The team looked forward to that cool-down time. They wanted to protect it. That's a much healthier dynamic than estimation theater."***

What This Means for You

Appetite-weeks won't fix a broken shaping process. They won't save you if your pitches are poorly scoped or your team doesn't trust each other. But if you're doing the work in Chapters 2, 3, and 4 — setting appetites strategically, shaping with business framing, running a disciplined betting table — then appetite-weeks turn capacity planning into simple arithmetic.

You stop debating velocity. You stop second-guessing estimates. You stop pretending you can predict the future.

You just count the weeks, fill the budget, and get to work.

Try This

Before your next cycle: Tally your team's available appetite-weeks using this formula:

1. List every person who'll contribute to the cycle
2. Count how many of the four build weeks they'll be available (subtract vacation, other commitments)
3. Adjust for role complexity (managers at partial capacity, firefighters at reduced capacity, etc.)
4. Add it all up

Bring that number to your next planning meeting. When someone asks, "Can we fit one more thing in?" you'll have a real answer.

Chapter 6: Running the Cycle

Your first cycle will be a mess.

This is not a warning. It's a fact. No matter how well you shape your pitches, how disciplined your betting table is, or how clearly you communicate the appetite-weeks model, the first cycle will feel like chaos. Teams trained on sprints will keep looking for rituals that don't exist. You'll scope things wrong. Something will bleed into cool-down. Someone will ask when the daily standup is.

That's fine. You're not copying a methodology. You're adapting to reality.

It took us about six months to get the rhythm right. Six months of experimenting with cycle lengths, figuring out what scope actually fit into an appetite, and unlearning habits that sprints had burned into muscle memory.

Here's what that looked like.

The First Six Months: Everything Went Wrong

When we started, we didn't just adopt Shape Up's six-week cycle format. We thought it was too slow for what we needed. Enterprise clients were demanding faster turnaround. The sales pipeline had its own rhythm. So we experimented.

First attempt: three weeks build, one week cool-down.

Too tight. Pitches bled into cool-down constantly. We weren't good at scoping appetites yet, so everything felt rushed. We also weren't great at holding boundaries with large enterprise clients. A big prospect would come in mid-cycle, and suddenly we'd have to swap out planned work for

something that could win the contract. Three weeks didn't give us enough buffer to absorb that kind of disruption.

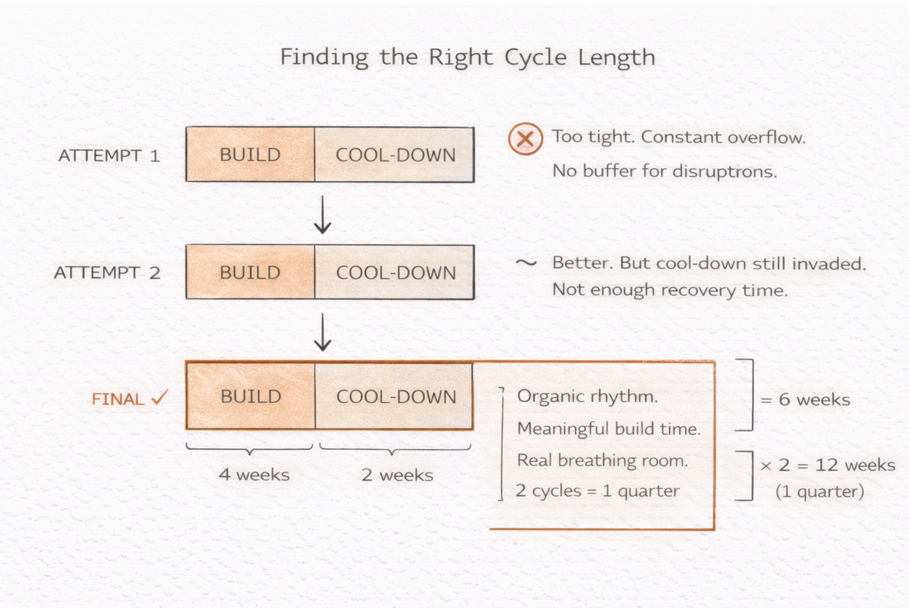
Second attempt: four weeks build, one week cool-down.

Better, but still not quite right. We had more breathing room, but the cool-down was still getting invaded regularly. One week wasn't enough time for technical health work, skill development, and recovery.

Final landing: four weeks build, two weeks cool-down. Two cycles per quarter.

This worked. Not immediately — we still made mistakes for another few months — but the rhythm started to feel organic. Teams had enough time to build something meaningful. The cool-down gave them actual breathing room. The quarterly cadence aligned with how we communicated roadmaps to customers and stakeholders.

We'd come full circle back to what Shape Up recommended. Not because we copied it, but because we tested the alternatives and learned why the six-week structure made sense.



The lesson: you're allowed to experiment. Just be rigorous about what you're learning.

The Disruption Problem

The other thing that kept breaking cycles early on: large clients jerking the roadmap.

We'd go into a cycle with a clean betting table. Then a seven-figure prospect would enter late-stage negotiations and need a specific feature to close the deal. Do we stick to the plan, or do we pivot mid-cycle to capture the revenue?

Most of the time, early on, we pivoted.

That made it nearly impossible to follow Shape Up the way it's written. The cycle would start with four pitches, and by week two, one of them had been swapped out for something that showed up three days ago. Teams were frustrated. They'd say, "This works so much better when we can actually stick to the cycle," and then two weeks later we'd disrupt it again.

This tension — between the disciplined rhythm Shape Up requires and the chaotic reality of enterprise sales — was real. It nearly killed the whole system.

The fix is in Chapter 8. For now, just know that if you're experiencing this, you're not doing it wrong. You're running into a problem that Shape Up, as designed for product companies like Basecamp, doesn't fully address. Enterprise B2B has different constraints. You'll need a different solution. (Spoiler: it's a dual-track model, and it works.)

Learning to Scope: The 80% Problem

Scope hammering is the core discipline of Shape Up. When you hit your appetite and the work isn't done, you cut scope. You don't extend the timeline. You don't add people. You cut.

That's the theory. Here's the practice.

At the beginning, I wasn't great at scoping pitches. I had a rough sense of what a two-week appetite or a four-week appetite could contain, but I hadn't calibrated yet. So I deliberately over-scoped pitches. I'd put in 100% of what I wanted, knowing the team would deliver about 80%.

My thinking: better to aim high and let the team cut the last 20% than to under-scope and leave capacity on the table.

The problem: it demotivated the team.

They'd work hard, deliver 80% of the pitch, and still feel like they didn't get across the finish line. Even though they knew I was over-scoping intentionally, it felt bad. Every cycle ended with a tail of unfinished work. It created a sense of falling short instead of shipping.

That was the wrong move long-term.

What worked better: scoping conservatively and separating must-haves from nice-to-haves ruthlessly. If the must-haves were tight and the nice-to-haves were truly optional, teams could finish the core work and then decide whether to tackle extras. That flipped the psychological dynamic.

> ***Instead of 'we didn't finish,' it became 'we finished and then grabbed a nice-to-have.'***

The rhythm that emerged: early in the cycle, the team and I would have a scoping conversation. We'd look at the pitch together — the problem, the solution sketch, the rabbit holes, the no-gos — and talk through what was actually achievable. Engineers would flag complexity I hadn't seen. I'd clarify which parts were truly essential and which were "would be nice." By the end of that conversation, we'd have a shared understanding of what "done" looked like.

That wasn't grooming. It wasn't estimation. It was collaborative scoping. And it worked because the engineers weren't being asked to predict timelines. They were being asked to solve a problem within a constraint.

Scope hammering happened naturally after that. If something took longer than expected, the team knew exactly which pieces to cut. No negotiation. No drama. Just: "The nice-to-haves didn't fit. We'll pitch them separately if they're still important."

We didn't do it exactly the way Shape Up describes. But the principle — scope is the variable, not time — became muscle memory.

Killing the Sprint Rituals

One of the hardest transitions for teams coming from Agile: the rituals disappear.

No daily standups. No sprint planning meetings. No sprint reviews. No retrospectives. No grooming sessions.

For people who've worked in two-week sprints for years, that feels disorienting. The rituals provided structure. Without them, teams worried they'd lose visibility, coordination, alignment.

Here's what we did instead.

Async Updates Replace Standups

We had one rule: update your status daily. End of day, post three things in Slack: - What got done today - What's up next - Where are the blockers

That's it. No meeting. No round-robin where everyone half-listens while waiting for their turn to talk. Just async text updates that anyone could read when it made sense for them.

This worked especially well during COVID when everyone was remote. You didn't need to synchronize schedules for a fifteen-minute standup. You could catch up on the team's progress in two minutes whenever you had a gap.

Blockers got handled in real-time. If someone flagged a blocker in their update, the person who could unblock them would reach out directly. No waiting until the next day's standup to surface the issue.

Workshops Replace Meetings

If two or three people needed to work through a complex problem together, they'd workshop it. Grab a virtual room (or a physical one, pre-COVID), sketch out the solution, make decisions, move on.

These weren't scheduled rituals. They happened when needed. Sometimes that was twice in one week. Sometimes a team went a full cycle without needing one.

The point: meetings became tools for solving problems, not recurring obligations.

Cool-Down Replaced Grooming

In a sprint-based system, you'd have grooming sessions where the product team walked engineers through upcoming work. The engineers would ask questions, estimate complexity, and then go back to their current sprint. Two weeks later, that groomed work would show up as tickets.

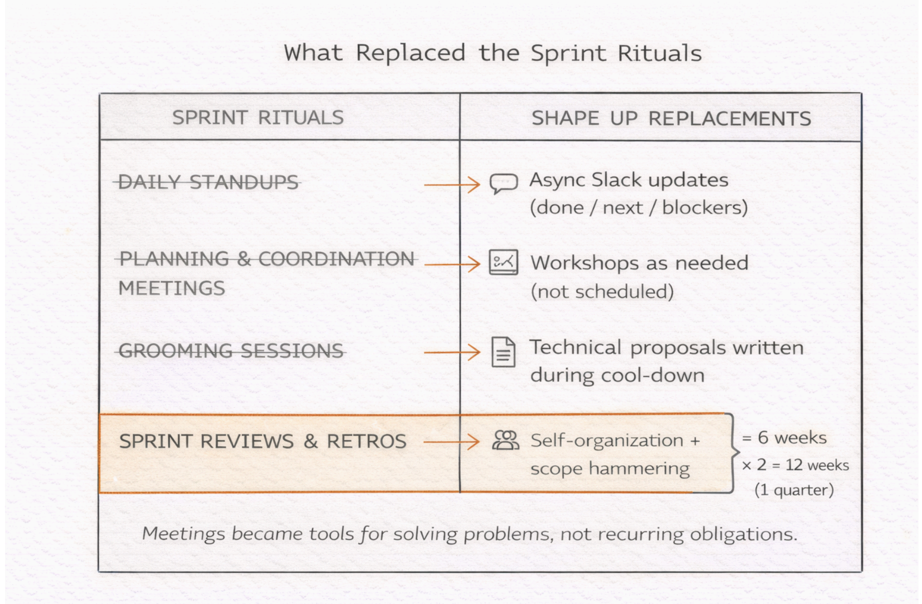
We replaced that with technical proposal writing during cool-down.

Pitches that had been bet on for the next cycle would get assigned to an engineer (or a small group) during cool-down. They'd read the pitch, think through the technical approach, and write up a brief proposal. Sometimes it was just a few paragraphs. Sometimes, for bigger pitches, it was a more detailed design.

This gave engineers proximity to the work right before they started building it. They weren't reviewing tickets that had been written weeks ago. They were engaging with the problem fresh, during the two weeks when they had space to think.

It also meant they could flag issues early. If a pitch had hidden complexity or a technical dead-end, the engineer would surface it during cool-down, and we'd adjust the scope before the cycle started.

No formal grooming meeting. No recurring ritual. Just: "Here's the work we're betting on next cycle. Spend some time in cool-down thinking through how you'd build it."



We return to this in Chapter 12, where we explore how AI accelerates engineering work within the cycle. ### Self-Organization Replaced Coordination Meetings

Here's the thing about giving a team a well-shaped pitch with a clear appetite: they don't need much coordination.

They know what problem they're solving. They know the boundaries (no-gos and rabbit holes). They know the time constraint. They can figure out how to organize the work themselves.

Sometimes that meant one person owned the whole pitch. Sometimes two people split it. Sometimes the team swarmed on it for the first week to break through complexity, then split up to finish individual pieces.

We didn't dictate how they worked. We just said: "Here's the pitch, here's the appetite, ship it."

That level of autonomy only works if the shaping is tight. If the pitch is vague or the problem poorly defined, teams thrash. But if you've done the work in Chapter 3 — clear problem, clear appetite, clear solution sketch — teams run fast without needing much oversight.

How Long Until It Felt Normal?

About six months.

The first couple of cycles were rough. Teams kept asking, "When's the standup?" or "Do we need to groom this?" We'd remind them: no standups, no grooming, just async updates and workshops as needed.

By the third or fourth cycle, people stopped asking. The rhythm started to feel natural. Engineers appreciated having longer uninterrupted stretches to focus. Product people appreciated not having to attend grooming meetings every week. Everyone appreciated the cool-down.

The moment I knew it had clicked: someone asked me, "Can we protect the cool-down better? I don't want this cycle's overflow to eat into it."

> ***"They weren't asking for the rituals back. They were asking to defend the new rhythm because it worked better."***

What This Means for You

Running the cycle isn't about rigid adherence to a six-week schedule. It's about finding the rhythm that works for your context.

Experiment with cycle lengths if you need to, but be honest about what you're learning. If you settle on something other than four weeks build + two weeks cool-down, make sure it's because the alternative actually works better, not because you're afraid to give teams breathing room.

Kill the sprint rituals, but replace them with something lightweight. Async updates. Workshops when needed. Cool-down for technical thinking. Don't just delete the structure — build a new one that's fit for purpose.

And expect the first six months to feel messy. You're not failing. You're learning what actually works when the theory meets your reality.

Try This

For your next cycle:

1. **Kill one sprint ritual.** Pick the meeting that feels least valuable (probably standups or grooming) and replace it with an async alternative. See what happens.

2. **Run a scoping conversation.** Early in the cycle, sit down with the team working on a pitch and talk through must-haves vs. nice-to-haves. Don't estimate. Just clarify what "done" looks like.

3. **Track what breaks**. When something goes wrong mid-cycle (scope explosion, external disruption, unclear pitch), write it down. You're building pattern recognition for what needs to improve in your shaping or betting process.

After three cycles, you'll have real data about what works in your context. Use it.

Chapter 7: Cool-Down: The Sacred Space Everyone Tries to Kill

Here's the math that makes executives nervous:

Six-week cycle. Four weeks building. Two weeks cool-down.

That's 33% of your engineering capacity not directly shipping features.

Every CFO, every board member, every VP of Sales who looks at that number asks the same question: "Why are we paying engineers to do nothing for a third of the year?"

The answer: you're not. And cool-down is exactly why your system doesn't collapse under its own weight.

But that's a hard sell. Enterprise organizations are wired to maximize utilization. Capacity that isn't directly producing output looks like waste. Cool-down looks like slack you can't afford.

Here's why you can't afford not to have it.

What Cool-Down Actually Is

Cool-down is not a vacation. It's not "free time" or "exploration sprints" or "innovation theater."

It's structured recovery and preparation time between cycles of focused, high-intensity work.

During the four-week build period, teams are locked in. They've committed to a pitch. They're solving a specific problem within a fixed appetite. There's no room for detours, experiments, or addressing technical debt that's been accumulating. The goal is to ship the bet.

That intensity works because it's time-boxed. Four weeks of sprint. Then two weeks to breathe.

Here's what actually happens during cool-down:

Technical Proposals for the Next Cycle

Remember Chapter 6? We killed grooming sessions and replaced them with technical proposal writing during cool-down.

Pitches that have been bet on for the next cycle get assigned to an engineer or small team. They read the pitch, think through the technical approach, sketch out a design. Sometimes it's three paragraphs. Sometimes it's a more detailed architecture doc.

This gives engineers proximity to the problem before they start building. They're not reviewing stale tickets from weeks ago. They're engaging with fresh work during a period when they have space to think deeply.

It also surfaces hidden complexity early. If a pitch has a technical dead-end or a dependency no one anticipated, the engineer flags it during cool-down. You adjust the scope before the cycle starts, not three weeks into the build when it's too late.

Bug Fixes and Technical Health

Bugs don't stop appearing just because you're mid-cycle. But during the build period, the team is focused on shipping the bet. Bugs get triaged, logged, and usually deferred unless they're critical.

Cool-down is when you clear the backlog.

Not all of it—you're not trying to reach inbox zero on your bug tracker. But the annoying, friction-inducing issues that have been piling up? The performance bottleneck that's been slowing down a common workflow? The refactoring you've been wanting to do for three cycles? Cool-down is when you tackle them.

This prevents technical debt from becoming technical bankruptcy. You're paying down the interest regularly instead of letting it compound until the system grinds to a halt.

Prototyping and Exploration

Sometimes the best pitches come from experiments that weren't pitches yet.

During cool-down, engineers have space to prototype an idea they've been mulling over. Try a different technical approach. Spike a feature concept to see if it's even feasible.

Some of these experiments go nowhere. That's fine. The cost is a few days during cool-down, not a full cycle bet.

Some of them become the seeds of future pitches. An engineer prototypes a caching layer during cool-down, discovers it's simpler than expected, and two cycles later it becomes a shaped pitch that ships in three weeks.

You can't schedule innovation. But you can create space for it. Cool-down is that space.

Skill Development and Learning

Enterprise software moves fast. Frameworks change. New tools emerge. Your platform evolves.

If your team never has time to learn, they fall behind. Not dramatically. Not all at once. Just a slow, steady drift toward obsolescence.

Cool-down gives teams permission to learn without it feeling like they're stealing time from "real work."

An engineer spends a day learning a new testing framework. Another takes a course on event sourcing. A small group does a deep dive into a competitor's product to understand how they solved a problem you're about to tackle.

This isn't frivolous. It's investment in capability. The returns show up three cycles later when that testing framework cuts QA time in half, or when the team applies event sourcing to a problem and ships it faster because they've already learned the pattern.

Recovery

Four weeks of focused, high-intensity work is draining.

Not in a "we're burning out" way. Just in a normal human way. You've been solving hard problems, coordinating with teammates, making trade-offs under time pressure, hammering scope to fit the appetite. It's mentally taxing.

Cool-down is when you recover.

Some people use it for focused deep work on technical proposals or prototypes. Some people use it to tackle small, low-stakes bugs that feel satisfying to close. Some people use it to catch up on reading, learning, or just having more relaxed conversations with teammates.

The point: you're not asking people to maintain 100% sprint intensity for fifty-two weeks a year. You're building in rhythm. Four weeks on, two weeks to breathe. Repeat.

Teams that have this rhythm don't burn out. Teams that don't, do.

How do you prioritize?

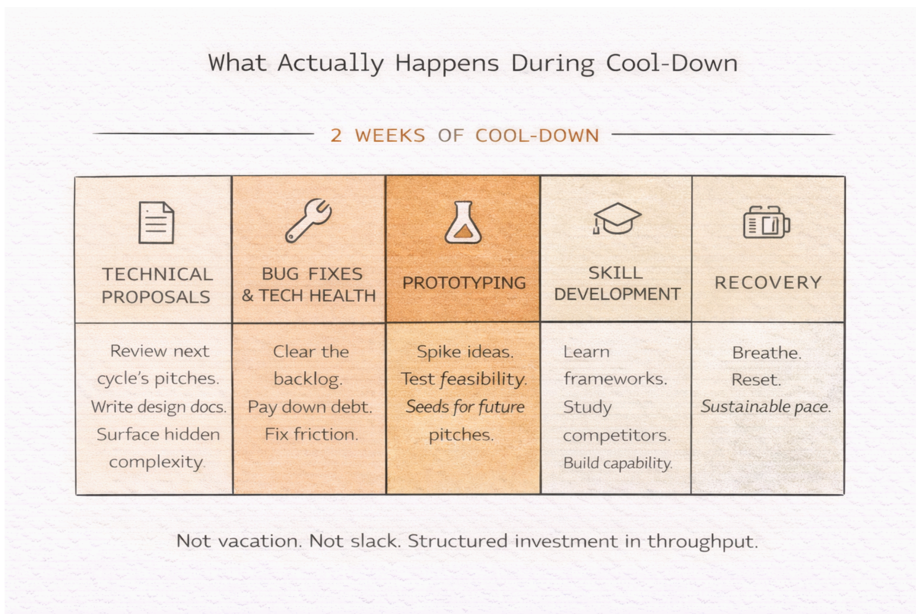
So how do you prioritize these five activities when you only have two weeks?

Non-negotiable (must happen every cycle): - Technical proposals for next cycle's work. If engineers don't engage with upcoming pitches during cool-down, you're back to grooming sessions and stale tickets. This is the foundation. Allocate time for it first. - Recovery. If you don't give teams breathing room, they burn out. This isn't optional. It's maintenance.

High priority (should happen most cycles): - Bug fixes and technical health. You don't need to reach inbox zero, but you can't let debt compound indefinitely. Dedicate at least a few days per cycle to clearing friction and paying down interest.

Opportunistic (happens when there's space): - Prototyping and exploration. Not every cycle will have room for this. That's okay. When teams finish technical proposals early or when cool-down is fully intact, prototyping is where they invest the slack. - Skill development. This compounds over time, but it's flexible. If an engineer wants to spend a day learning a new framework, they can do it when the higher-priority work is done.

The key: technical proposals and recovery are sacred. Everything else is negotiable based on capacity and need. If a pitch bleeds into cool-down and consumes three days, you protect technical proposals and recovery first. Bug fixes and prototyping get scoped down or deferred.



The Battle to Protect It

Cool-down will be under constant attack.

Not maliciously. Just... economically. When you're looking at quarterly targets and a sales pipeline full of deals that need "just one more feature," the question becomes: "Can't we just use cool-down to build this one thing?"

The answer has to be no.

Here's why that's hard.

The Utilization Argument

Finance and operations leaders are trained to think in terms of utilization. If a resource isn't producing output, it's wasted capacity.

Cool-down looks like 33% utilization. That's terrible by traditional metrics.

But engineering capacity isn't a factory line. You're not stamping out identical widgets where maximizing throughput is the goal. You're solving novel problems in a complex system where sustainability and quality matter more than raw output.

The case you have to make: cool-down is an investment in throughput over time, not a reduction in it.

Without cool-down: - Technical debt compounds until velocity collapses - Engineers burn out and leave, costing you months of ramp-up time for replacements - You never surface hidden complexity in pitches, so cycles fail more often - Innovation stops because there's no space to experiment

With cool-down: - Technical health stays manageable - Team retention stays high because the rhythm is sustainable - You catch scope issues before they blow up cycles - Small experiments compound into major improvements

The ROI isn't immediate. It shows up six months later when you're still shipping consistently and your competitors have ground to a halt under the weight of their own technical debt.

The "Just This Once" Trap

The most dangerous threat to cool-down isn't someone trying to eliminate it outright. It's erosion.

A big deal needs one feature to close. It's worth seven figures. Can we just use this cool-down to build it?

Just this once.

Except it's never just once.

You say yes to the seven-figure deal. Then next cycle, there's a six-figure deal. You've already set the precedent that cool-down is negotiable, so saying no feels inconsistent. You say yes again.

By the fourth cycle, cool-down has become de facto overflow time. Teams aren't recovering. Technical debt is piling up. No one's writing technical proposals anymore because there's no time. Prototyping stops. Bugs accumulate.

And then, six months later, velocity collapses. Cycles start failing. Teams are frustrated. People start leaving.

The pattern is predictable. The solution is to hold the line.

Cool-down is sacred. Not because it's untouchable in a crisis, but because treating it as negotiable kills it.

> ***"Cool-down is sacred. Not because it's untouchable in a crisis, but because treating it as negotiable kills it."***

When Cool-Down Gets Invaded (And What to Do About It)

I'm not going to lie to you: sometimes cool-down gets invaded.

In the early cycles we ran, we bled into cool-down fairly regularly. We were still learning to scope pitches. We were still calibrating appetites. We weren't great at holding boundaries with large clients.

A pitch would run over. The team would use part of cool-down to finish it.

That happened. It wasn't ideal. But it also wasn't catastrophic—as long as it was the exception, not the rule.

Here's how to manage it:

Treat Overflow as a Signal, Not a Solution

If a pitch bleeds into cool-down, that's a diagnostic signal.

What went wrong? Was the shaping too loose? Did we under-scope the appetite? Did the team hit unforeseen complexity that we should have caught in the technical proposal phase?

You don't just shrug and say, "Well, we have cool-down for overflow." You investigate and fix the root cause.

Track the Pattern

If the same team bleeds into cool-down multiple cycles in a row, that's not bad luck. That's a systemic problem.

Either: - The shaping for their work isn't tight enough - Their appetite budgets are consistently under-scoped - They're being disrupted mid-cycle in ways that shouldn't be happening - There's a skills mismatch (the team doesn't have the expertise the pitches assume)

You can't fix it if you don't track it. Keep a simple log: which pitches overran, by how much, and why. After three cycles, patterns emerge.

Protect Cool-Down More Zealously Over Time

In the first six months, you're going to make mistakes. Pitches will overrun. Cool-down will get used as a buffer. That's fine—you're learning.

But by cycle four or five, you should be tightening up. Fewer overruns. More disciplined scoping. Clearer boundaries with stakeholders.

By cycle six, cool-down should be mostly intact. Teams should be using it for what it's designed for: technical proposals, bug fixes, prototyping, learning, recovery.

If you're still bleeding into cool-down regularly after six months, something's broken. Don't normalize it. Fix it.

The Moment You Know It's Working

I mentioned this in Chapter 6, but it's worth repeating here.

The moment I knew cool-down had become part of the team's DNA: someone asked me, "Can we protect cool-down better? I don't want this cycle's overflow to eat into it."

They weren't asking for the old sprint rituals back. They weren't complaining about the four-week cycles being too long.

They were asking to defend the rhythm because it worked.

That's the signal. When your team starts protecting cool-down because they value what it gives them—recovery, space to think, time to fix the things that have been bothering them—you've won.

What You're Actually Buying

Cool-down costs you 33% of your engineering calendar.

Here's what you get in return:

Sustainable pace. Teams don't burn out. Retention stays high. You're not constantly onboarding replacements and losing institutional knowledge.

Technical health. Debt gets paid down regularly. The system doesn't degrade into an unmaintainable mess.

Better pitches. Engineers engage with next cycle's work during cool-down and surface complexity early. Fewer cycles fail because of unforeseen technical problems.

Innovation. Small experiments during cool-down compound into major improvements. Features that would never make it through a formal betting process get prototyped and prove their value.

Happier teams. People like working in a system that gives them breathing room. They do better work when they're not constantly sprinting.

Those were a little too qualitative though. Let's put some numbers on it.

Cost of burning out a senior engineer: Conservatively, six months. Three months to recruit and hire a replacement, three months for them to ramp up to full productivity. During that time, you've lost the institutional knowledge, the domain expertise, and the throughput that engineer was providing. If a senior engineer costs $200K fully loaded, you're looking at $100K in lost productivity, plus recruiting costs, plus the opportunity cost of work that didn't happen.

Cost of technical debt accumulation: Harder to quantify, but here's a rough heuristic. If your team is shipping features in 4-week cycles and technical debt isn't getting addressed, velocity will degrade by 20-30% over six months as the system becomes harder to change. That's the equivalent of losing one full-time engineer on a five-person team—except you're still paying for five people.

Cost of failed cycles due to unforeseen complexity: If engineers don't write technical proposals during cool-down, you're betting on pitches blind. Our data at Receeve: pitches without technical proposals had a 40% higher failure rate (either blew the appetite significantly or had to be killed mid-cycle). That's wasted capacity—entire cycles spent on work that doesn't ship.

Now compare that to the cost of cool-down: 33% of calendar time that prevents burnout, keeps technical debt manageable, and surfaces

complexity early. The ROI is real. It's just not immediate. You're optimizing for long-term throughput and sustainability, not short-term utilization.

If you're playing the quarter-to-quarter game, cool-down looks expensive.

> ***"If you're building a product organization that ships consistently for years, cool-down is non-negotiable."***

What This Means for You

You will face pressure to eliminate cool-down. Or to compromise it. Or to treat it as overflow time.

The pressure will come from well-meaning people who genuinely believe maximizing utilization is the right move.

Your job is to hold the line.

Cool-down is not wasted capacity. It's the release valve that keeps your system from exploding under pressure.

Protect it.

Try This

For your next cycle:

1. **Track what happens** in cool-down. Have each team member log what they worked on during the two weeks. Bug fixes? Technical proposals? Prototyping? Learning? You'll see the value when it's visible.

2. **Run a retrospective.** After two or three cycles, ask the team: "What's the most valuable thing that came out of cool-down?" Surface the wins. Use them to build the case when stakeholders

question it. Yes, I mentioned killing rituals earlier in the book, but a retrospective helps surface issues.

3. **Set a boundary.** If a pitch runs over this cycle, decide before the cycle starts whether overflow is allowed or whether you'll cut scope to finish in four weeks. Make it an explicit choice, not a default.

Cool-down only works if you protect it. Start now.

Chapter 8: Shape Up Isn't the Whole System

In Chapter 6, I said the fix was coming. This is it.

Here's the problem we kept hitting in Chapter 6: large clients jerking the roadmap mid-cycle.

A seven-figure prospect enters late-stage negotiations. They need a specific integration to close the deal. It has to be live by a specific date—their date, not yours. The work is completely dependent on when they finish their side of the integration. If they slip by two weeks, you can't start your work.

Try to fit that into a Shape Up cycle.

You can't.

The appetite is meaningless because you don't control the timeline. The scope is fixed because it's driven by their system requirements. The cycle breaks because you're waiting on an external party who doesn't care about your six-week rhythm.

In the second year at Receeve, we tried to make everything fit into Shape Up anyway. We were committed to the methodology. We wanted a unified way of working. One process, one rhythm, one set of rituals for the whole team.

It was a disaster.

Pitches failed because customer deliverables slipped. Engineers had to spend time juggling things instead of focusing on meaningful work. The roadmap looked unreliable because the same items kept getting pushed to the next cycle. Teams were frustrated.

The methodology wasn't broken. We were just using the wrong tool for the job.

The Dual-Track Model

Here's what worked: Shape Up for product increments. Kanban for service-layer work.

Two parallel streams. Two methodologies. Same capacity pool.

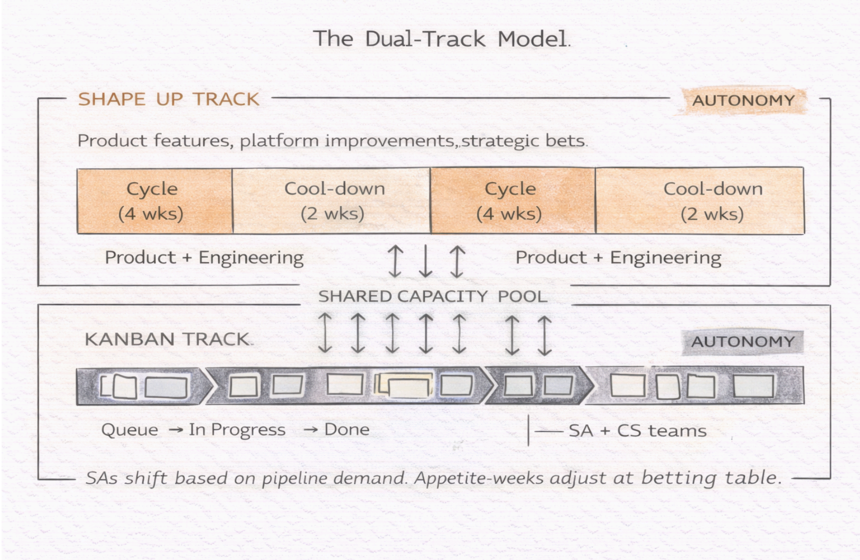
Shape Up governed product work—features, platform improvements, anything we controlled end-to-end. Cycles, pitches, appetites, betting tables. The full system.

Kanban governed service-layer work—client integrations, onboarding projects, anything driven by external timelines or customer-specific requirements. No cycles. No pitches. Just prioritized work flowing through as it arrived, managed by the Solutions Architecture and Customer Success teams.

The SA and CS teams owned their Kanban board with full autonomy. They decided what to prioritize. They managed timelines with customers. They moved work through their pipeline without asking product or engineering leadership for permission.

The product team owned the Shape Up side with the same autonomy.

Two tracks. Two methodologies. No stakeholder fights about capacity.



What Belongs in Kanban vs. Shape Up

The decision framework is simple: who controls the variables?

Kanban: Externally-Driven Work

If the work is dependent on external parties or timelines you don't control, it goes into Kanban.

Examples: - Client integrations: A bank is launching a new payment service. They have a go-live date. They're building their side of the integration in parallel with you. If they slip, you can't proceed. That's Kanban. - Onboarding projects: A new customer needs a custom configuration or data migration before go-live. Timeline is driven by their contract start date and their readiness, not your cycle rhythm. Kanban. - Customer-specific requests: An existing customer needs a one-off integration with their internal system. Scope is fixed by their requirements. Timeline is fixed by their business needs. Kanban.

The common thread: you don't control when the work can start, when it can finish, or what "done" looks like. The customer or external partner controls those variables.

Trying to force this into a Shape Up cycle creates chaos. The appetite doesn't mean anything when the timeline is externally imposed. Scope hammering doesn't work when the requirements are fixed by someone else's system. The cycle breaks when external dependencies slip.

Kanban handles this gracefully. Work arrives, gets prioritized, flows through when the external dependencies are ready. No artificial batching into cycles. No pretending you have control you don't actually have.

Shape Up: Internally-Controlled Work

If you control the problem definition, the timeline, and the scope, it goes into Shape Up.

Examples: - New product features: A fraud detection improvement that you've shaped, scoped, and bet on. You control what gets built and when it ships. Shape Up. - Platform improvements: Refactoring the notification system to reduce latency. You define the problem, set the appetite, decide what "good enough" looks like. Shape Up. - Technical debt: Upgrading a core dependency or improving test coverage. You control the scope and timeline. Shape Up.

The common thread: you own the variables. You can set an appetite, hammer scope if needed, and ship when the cycle ends—regardless of what any external party is doing.

This is where Shape Up shines. Fixed time, variable scope, clear bets.

How the Dual-Track Model Worked in Practice

SA/CS Teams Had Full Autonomy

The Solutions Architecture and Customer Success teams ran their Kanban board without interference from the product organization.

They decided: - Which customer projects to prioritize - How to sequence work based on contract timelines and dependencies - When to escalate issues or negotiate scope with customers - How to staff projects (which SAs worked on which accounts)

Product didn't attend their planning meetings. Engineering leadership didn't review their priorities. They owned their domain.

This eliminated a huge source of organizational friction. In most enterprise B2B companies, there's constant tension between product (wanting to build the roadmap) and customer-facing teams (needing custom work to close deals or retain accounts). That tension usually gets resolved in a meeting where someone with authority makes a call, and half the room walks out frustrated.

We didn't have that problem. The SA/CS team had their lane. The product team had theirs. Autonomy replaced negotiation.

The Capacity Pool Was Shared

Here's where it gets interesting: the people were shared, even though the processes weren't.

We had Solution Architects who spent most of their time on the Kanban track—working with customers on integrations, onboarding, and custom projects. But when the Kanban pipeline was light, they could shift to the Shape Up track and contribute to product work.

How this worked:

At the Horizon 2 betting table (planning the cycle nine weeks out), we'd look at the sales pipeline and the onboarding queue. If we had visibility that the next cycle would be light on customer projects, we'd plan for one or two SAs to be available for product work.

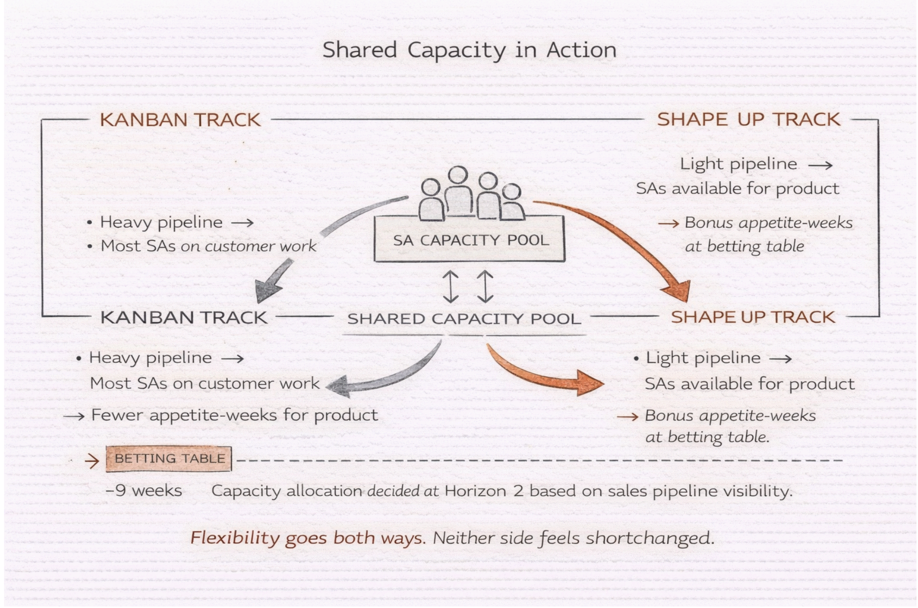
That gave us bonus capacity on the product side. Maybe we could take on an extra pitch. Maybe we could tackle a technical debt item that had been parked for a while. Maybe we could allocate more appetite-weeks to a complex platform improvement.

If the sales pipeline was heavy, we'd plan with fewer SAs on the product side. The betting table would adjust the appetite-weeks budget accordingly.

The SAs tracked their time in both systems. If they were 100% on customer projects for a cycle, they contributed zero appetite-weeks to the Shape Up side. If they were 50/50, they contributed two appetite-weeks (half of the four-week build period).

This flexibility went both ways. Occasionally, a major customer project would spike mid-cycle—a deal closing faster than expected, or an integration hitting unexpected complexity. An SA working on a product pitch could shift back to the Kanban side. The product pitch would adjust scope (hammer the nice-to-haves) or use cool-down to finish.

It wasn't disruptive because the work on both sides was managed to accommodate shifts. Product pitches were scoped with must-haves and nice-to-haves. Customer projects were staffed with some buffer for the unexpected.



Why It Didn't Create Chaos

You might be thinking: "This sounds like constant context-switching and thrashing."

It wasn't. Here's why.

First, the shifts were predictable. We had sales pipeline visibility. We knew which customers were onboarding. We knew which integrations were queued. The betting table nine weeks out gave us time to plan capacity intelligently.

Second, the SAs were skilled at both modes. They weren't product managers pretending to be engineers or engineers pretending to be customer success. They were technical people who could work or core product features and manage a customer integration. For them it was a

manner of managing their time more strictly than anyone dedicated fully to one process or the other.

Third, the autonomy made it psychologically safe. SAs weren't being yanked around by competing priorities and political fights. They understood: customer projects come first (that's the business model), but when there's space, contribute to product work. No drama. No resentment. Just clarity.

Fourth, neither side felt shortchanged. The product team got bonus capacity when it was available and didn't count on it when it wasn't. The SA/CS team got full control of their domain and didn't have product trying to dictate their priorities.

Autonomy plus transparency equals low friction.

What Would Have Broken

We tried the "one methodology for everything" approach early on. Everyone was excited about Shape Up. The team wanted a unified way of working. It felt cleaner.

It failed for predictable reasons.

Customer integrations blew up cycles. A pitch would be scoped for four weeks, but it depended on the customer delivering their API spec. The customer would slip by two weeks. The engineer assigned to the pitch couldn't proceed. The cycle would end with the pitch incomplete, pushed to the next cycle.

This happened repeatedly. Not once or twice. Consistently.

The roadmap became unreliable. Stakeholders—sales, CS, the board—would look at the thematic roadmap and see items that kept getting pushed. "Didn't we say this would ship last cycle?" Yes. But the customer dependency slipped. "Why did we commit to it then?" Because we tried to force it into Shape Up when it didn't belong there.

Teams were frustrated. Engineers hated having pitches fail due to external factors they couldn't control. SAs hated being held to cycle deadlines when the work was inherently driven by customer timelines. Product people hated the unpredictability.

It undermined trust in the system. When Shape Up worked (on internally-controlled work), teams loved it. When it failed (on externally-driven work), they blamed the methodology. "Shape Up doesn't work for enterprise." That's not true. Shape Up doesn't work for everything in enterprise. Big difference.

The fix: stop trying to unify everything. Use the right tool for the job. Shape Up doesn't work for everything in enterprise.

When to Exit Shape Up Entirely (Temporarily)

There's one more scenario where we'd set Shape Up aside: when we needed to focus entirely on existing customer satisfaction for a period.

This happened once. We'd spent two quarters heavily focused on new customer acquisition. New features to win deals. Integrations for prospects. The roadmap tilted toward growth.

Existing customers started grumbling. Not loudly. Not threatening to churn. But the feedback was clear: "You're focused on shiny new things. The stuff we use every day has friction."

We made a deliberate decision: pause Shape Up for one quarter and shift to Kanban for the whole product team.

Why Kanban? Because the work was already identified. Customer Success had a list of improvements, performance issues, and usability friction points that existing customers cared about. We didn't need to shape pitches or run a betting table. We needed to execute on a known backlog.

For three months, the product team operated in Kanban mode. Prioritized list. High-throughput execution. No cycles. No pitches. Just: “Here’s what needs to improve. Go.”

It worked because it was time-boxed. We said: “One quarter. Then we return to Shape Up.”

That time-box was critical. Without it, you drift into permanent project mode. You lose the strategic discipline that Shape Up provides—the appetite-setting, the scope-hammering, the bets on what’s worth building. You start just reacting to requests instead of shaping the future.

But for a focused, time-boxed period, exiting Shape Up to tackle a known body of work made sense.

The product team also used that quarter to reset. Reassess the roadmap. Recalibrate priorities. Figure out if we’d drifted too far in one direction.

Then we came back to Shape Up with fresh energy and clearer direction.

The Decision Framework

How do you decide what belongs in each track?

Ask three questions:

1. Who controls the timeline?

- You do: Shape Up
- External party does: Kanban

2. Who defines “done”?

- You define scope and can hammer it: Shape Up
- Requirements are fixed by external needs: Kanban

3. Is this a bet or a commitment?

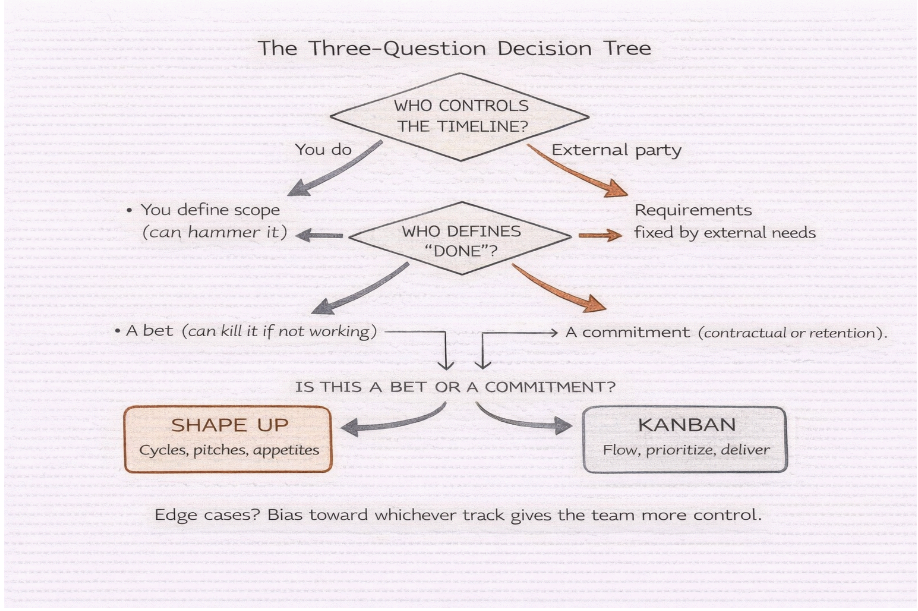
- We’re betting this is worth building (and we can kill it if not): Shape Up

- We've committed to deliver this (contractually or for retention): Kanban

If you answer "us/us/bet," it's Shape Up.

If you answer "external/external/commitment," it's Kanban.

Most work is clear-cut. Some sits in the middle. For edge cases, bias toward the track that gives the team more control. Control enables better execution.



What This Means for You

You don't need to force everything into one methodology.

Enterprise B2B has two fundamentally different types of work: the work you control (product increments, platform improvements, strategic bets) and the work you don't (customer integrations, onboarding, externally-imposed timelines).

Shape Up is exceptional for the first type. It's a disaster for the second.

Run two tracks. Give each team autonomy over their domain. Share capacity intelligently. Adjust appetite-weeks at the betting table based on what's coming.

And if you need to pause Shape Up for a time-boxed period to tackle a focused body of work, do it. Just make sure you set a date to return.

Methodology isn't religion. It's a tool. Use the right one for the job.

Try This

For your next quarter:

1. **Audit your work.** Go through the last three cycles and identify pitches that failed or ran over. How many were due to external dependencies or customer-driven timelines? If it's more than 20%, you're forcing the wrong work into Shape Up.

2. **Define the boundary.** Create a simple decision tree: "Does this go in Shape Up or Kanban?" Use the three questions above. Share it with the team so everyone knows how to route work.

3. **Give one team autonomy.** If you have a customer-facing team (SA, CS, implementation), let them own their Kanban board for one quarter without product approval. See what happens. You might be surprised how much organizational friction disappears.

Two tracks. Two tools. One system.

Chapter 9: When Shape Up Breaks

Shape Up will break.

Not might. Will.

You'll run into situations where the six-week cycle doesn't fit. Where the appetite is meaningless. Where a pitch that looked tight during shaping turns out to be full of holes. Where a commitment gets made outside the betting table and lands on your team like a grenade.

This isn't a sign you're doing it wrong. It's a sign you're doing real work in a real organization.

The question isn't whether Shape Up will break. The question is: what do you do when it does?

This chapter is the honest one. The failure modes, the edge cases, the adaptations we made when the pure methodology couldn't survive contact with enterprise reality.

The Biggest Failure: Forcing Everything Into One System

I covered this in Chapter 8, but it's worth revisiting from the failure lens because it was the most painful mistake we made.

We wanted a unified system. One methodology for all work. Shape Up everywhere. It felt cleaner. More consistent. Easier to explain to the organization.

It failed spectacularly. For six hard months, every single cycle failed for one reason or another to finish all the pitches.

Here's why it hurt more than just "some pitches didn't ship":

It broke trust in the system.

When product work failed (the work we controlled), we could explain it: "We scoped too aggressively. We'll tighten it next cycle." The team understood. They'd learned something. The system was working as designed—scope is the variable, and we were calibrating.

When customer-driven work failed (the work we didn't control), the explanation didn't land: "The customer slipped their timeline, so we couldn't finish." To the broader organization—sales, CS, finance, the board—that sounded like an excuse.

They'd look at the roadmap and see the same items cycle after cycle. "Didn't we commit to this last quarter?" Yes, but the customer dependency slipped. "Why did we commit if it depended on them?" Because we tried to force it into Shape Up when it didn't belong there.

The roadmap looked unreliable. Not because we were shipping slowly, but because we kept replanning the same work. That eroded confidence.

Stakeholders stopped trusting the process. If the methodology couldn't deliver predictably, why should they respect it? Why not just ask for features mid-cycle? Why not pressure the team to "just get it done"?

The fix—dual-track model—solved the mechanical problem. But the trust damage took longer to repair. We had to re-earn credibility by shipping consistently for three or four cycles before stakeholders believed the system worked.

> ***"Lesson: Methodology failures compound into organizational failures. A broken cycle is fixable. Broken trust takes months to rebuild."***

Multi-Team Dependencies: When Cycles Collide

Here's a failure mode Shape Up doesn't address well: what happens when one team's cycle depends on another team's output?

Example: The platform team is building a new API. The product team has a pitch that depends on that API being live. Both teams are running six-week cycles, but they're not synchronized.

The platform team bets on the API for Cycle 3. The product team bets on their feature for Cycle 4, assuming the API will be ready.

The platform team hits unexpected complexity. They ship 80% of the API in Cycle 3, but the remaining 20% bleeds into cool-down. It's not fully documented and tested until two weeks into Cycle 4.

Meanwhile, the product team is already in Cycle 4, trying to build their feature. They're blocked for the first two weeks waiting for the API.

What do you do?

We tried a few approaches:

Option 1: Buffer Dependencies Into the Pitch

When shaping a pitch that depends on another team's work, assume the dependency will slip and build buffer into the appetite.

If you think a feature needs four weeks once the API is ready, shape it as a six-week pitch with the assumption that the first two weeks might be blocked.

This works if dependencies are rare. It doesn't scale if your teams are highly interdependent.

Option 2: Synchronize Betting Tables

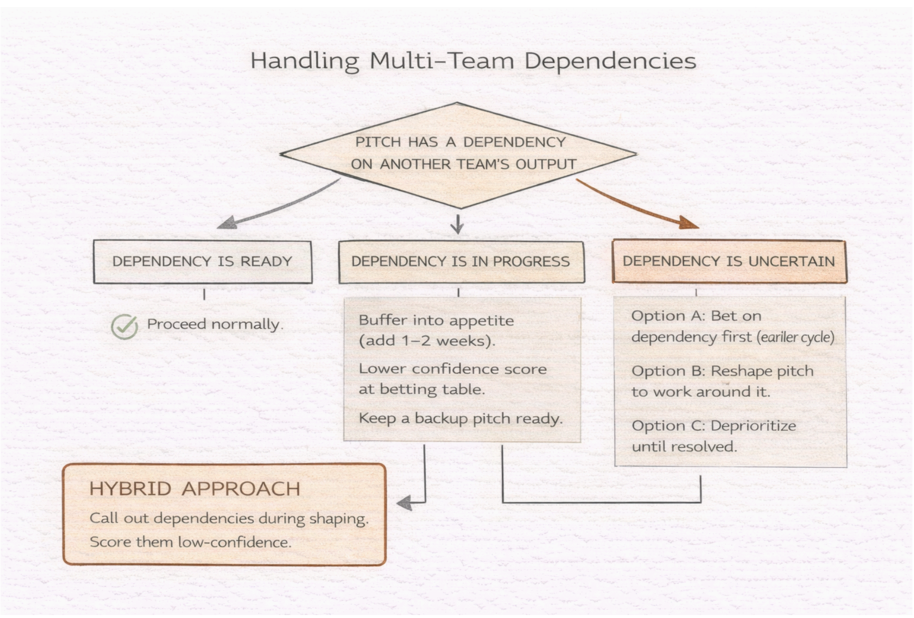
Run both teams' betting tables at the same time. Dependencies get surfaced during betting. If Team A's pitch depends on Team B's output, you either: - Bet on Team B's work first (in an earlier cycle) - Deprioritize Team A's pitch until the dependency is resolved - Reshape Team A's pitch to work around the dependency

This works better but requires tight coordination. It also makes the betting table longer and more complex.

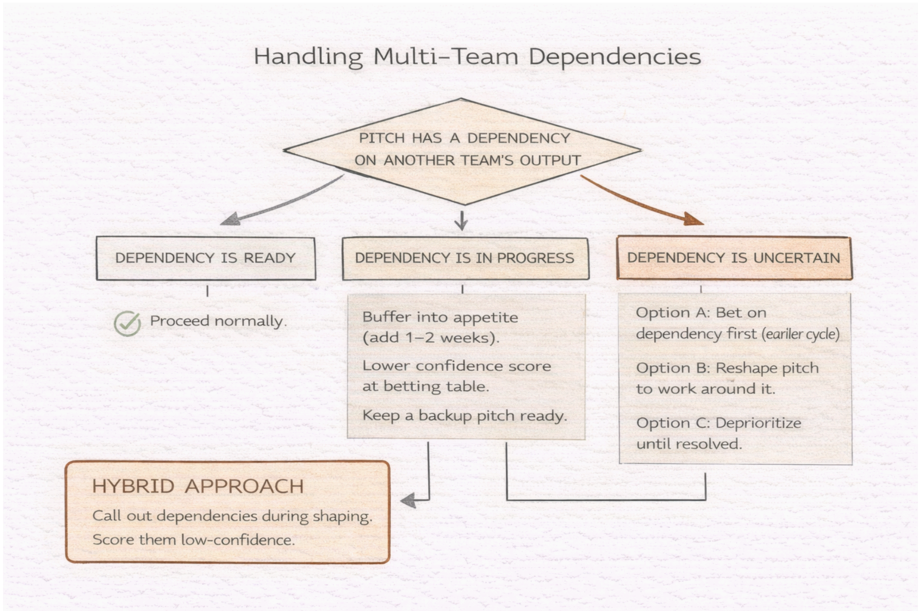
Option 3: Accept Some Cycle Failures

Sometimes a pitch just fails because the dependency didn't land. You don't try to save it mid-cycle. You let it fail, learn from it, and reshape it for the next cycle.

This is honest, but it's demoralizing if it happens too often.



What worked best for us: A hybrid. We'd explicitly call out dependencies during shaping and give those pitches lower confidence scores at the betting table. If we bet on them anyway, we'd keep backup pitches ready—



smaller, lower-risk work that a team could pivot to if the dependency didn't materialize.

Not perfect. But it prevented total cycle failures.

Work That Doesn't Fit Six Weeks

Some work is just too big for a six-week cycle, and breaking it down feels artificial.

Examples: - Platform migrations: Moving from one database to another, or from monolith to microservices - Infrastructure overhauls: Upgrading core dependencies across the entire stack - Regulatory compliance projects: Implementing GDPR, SOC 2, or financial reporting requirements with fixed scope

You have two choices: force it into cycles, or run it outside the cycle system.

Forcing It Into Cycles (Usually a Mistake)

You can try to break a twelve-week migration into two six-week pitches. Pitch 1: "Migrate the authentication service." Pitch 2: "Migrate the billing service."

This works if the work is truly decomposable. But migrations often aren't. You can't migrate half a database and call it done. The work has hidden dependencies, shared infrastructure, and non-linear complexity.

What happens: Pitch 1 takes seven weeks. Pitch 2 reveals unexpected issues discovered during Pitch 1. By the third cycle, you're still in the migration and everything feels like it's dragging.

The team hates it. Stakeholders see the migration eating multiple cycles and wonder why it's taking so long. The cycle rhythm—which is supposed to create momentum—starts feeling like friction.

Running It Outside the Cycle System

For genuinely large, unavoidable work, run it as a parallel project outside Shape Up.

Assign a dedicated team (or part of a team). Give them a realistic timeline—twelve weeks, sixteen weeks, whatever it takes. Don't pretend it fits into cycles.

The rest of the organization keeps running Shape Up. The migration team operates in project mode with clear milestones and checkpoints.

This only works if: - The work is truly unavoidable (regulatory deadline, critical technical debt) - You can afford to pull people off the cycle system temporarily - The scope is well-defined enough that project mode makes sense

We did this once for a major infrastructure upgrade. It took three months. The team hated being outside the cycle rhythm—they missed the cool-

down, the clear finish line every six weeks. But forcing it into cycles would have been worse.

The key: time-box it. Set a deadline for when the project ends and the team returns to cycles. Don't let "project mode" become permanent.

The CEO-Promised-a-Feature Problem

This one's brutal: the CEO is in a sales meeting and promises a feature to close a deal. Then they walk into your office and say, "We need this in four weeks."

It's not shaped. It's not in a pitch. It didn't go through the betting table. But it's a commitment, and it's attached to revenue.

What do you do?

Option 1: Run It in the Kanban Track

If the feature is customer-specific or driven by a contract deadline, it might belong in the Kanban stream (Chapter 8) rather than Shape Up.

Pull an SA or two off the product track, run the feature as a customer project, deliver it on the timeline the CEO promised.

This keeps the Shape Up cycles clean and stable while still honoring the commitment.

What didn't work: Saying "no" to the CEO without offering alternatives. You can't ignore revenue-critical commitments. But you can shape how they get delivered.

Option 2: Treat It as a Horizon 1 Disruption

Remember the disruption rules from Chapter 4? Large deals (ARR-based threshold) can disrupt Horizon 1 if they're rare and predictable.

If this feature fits that criteria, you invoke the disruption process: pull a pitch from the current cycle, replace it with the CEO's commitment, communicate the swap to stakeholders.

This works if: - Disruptions are rare (twice in six years for us) - The CEO understands they're spending political capital when they do this - The feature actually fits in the time they promised

Option 3: Negotiate the Timeline

Sometimes the CEO promises a feature in four weeks, but it's actually a six-week piece of work (or more).

Your job: make the trade-offs visible.

"We can do this in four weeks if we cut X, Y, and Z. The customer gets a minimum viable version, and we ship the rest later. Is that acceptable?"

Or: "This is realistically a six-week build. If we commit to four weeks, we'll blow the cycle and destabilize the next one. Can we negotiate the timeline with the customer?"

CEOs often don't know how long things take. They're optimizing for closing the deal. If you can show them the trade-offs clearly—scope, timeline, or risk to other work—they'll often negotiate.

> ***"You can't ignore revenue-critical commitments. But you can shape how they get delivered."***

When Shaping Fails

Sometimes a pitch looks tight during shaping, passes the betting table, gets assigned to a team... and then falls apart in week two.

The problem was harder than you thought. The solution sketch had a fatal flaw. The scope was way off.

What now?

Kill the Bet

Shape Up says: if a pitch isn't working, kill it. Don't extend the cycle. Don't throw more people at it. Let it fail, learn from it, and move on.

This is the right call when: - The problem is genuinely harder than anticipated - Continuing would mean bleeding into cool-down every cycle - There's a better approach that requires reshaping from scratch

We killed bets a few times. It felt bad in the moment, but it was the right move. The team used cool-down to write up what went wrong, and we reshaped the pitch for a future cycle with better information.

Scope Hammer Aggressively

If the core of the pitch is still viable, cut everything that's not essential and ship the must-haves.

This works when: - The problem definition was right, but the solution was over-scoped - There's still value in shipping a smaller version - The team can identify a clean line between must-haves and nice-to-haves

We did this more often than killing bets. A four-week pitch would hit complexity in week three, and we'd ruthlessly strip it down to the core. Ship it. Evaluate. Reshape the rest later if it's still worth doing.

Post-Mortem the Shaping Process

If shaping keeps failing, something's broken in the shaping process.

Common issues: - Not enough upfront exploration: The shaper didn't de-risk the unknowns before betting - Poor technical proposals: Engineers didn't flag complexity during cool-down - Wishful thinking on scope: The pitch assumed everything would go smoothly

We'd do lightweight post-mortems after failed pitches: "What did we miss during shaping? What should we do differently next time?"

Over time, shaping got tighter. Fewer failed bets. More accurate appetites.

What to Keep Pure vs. What to Adapt

Here's the question everyone asks: "How much can I change Shape Up before it stops being Shape Up?"

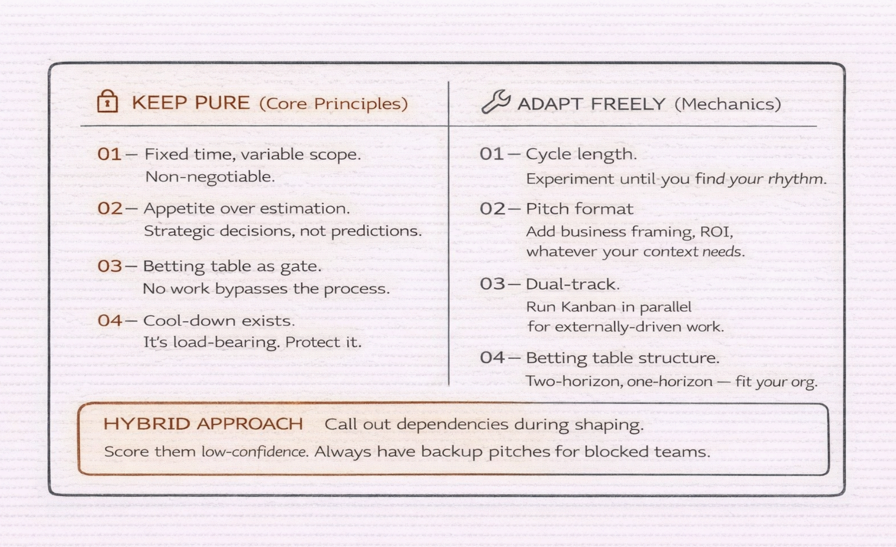
There's no universal answer, but here's what we learned:

Keep Pure (Core Principles):

- Fixed time, variable scope: This is non-negotiable. If you start extending cycles to fit scope, you've lost the discipline.
- Appetite over estimation: Don't go back to estimating story points. Appetite is a strategic decision, not a prediction.
- Betting table as gate: Don't let work bypass the betting process (except rare, pre-defined disruptions).
- Cool-down exists: Don't kill it. Protect it. It's load-bearing.

Adapt Freely (Mechanics):

- Cycle length: We experimented with 3+1, 4+1, settled on 4+2. Adjust to your context.
- Pitch format: Add business framing, ROI fields, whatever makes sense for enterprise.
- Dual-track: Run Kanban in parallel for work that doesn't fit Shape Up.
- Betting table structure: Two-horizon model worked for us. One-horizon might work for you.



The principles are what make Shape Up work. The mechanics are how you apply it to your reality.

If you preserve the principles and adapt the mechanics, you're fine.

If you start compromising the principles—extending cycles, skipping shaping, letting work bypass betting—you'll drift into something that looks like Shape Up but doesn't deliver the benefits.

What This Means for You

Shape Up will break. That's normal.

When it does: - Diagnose the root cause. Is it a failure of shaping? A dependency issue? Work that doesn't belong in cycles? - Adapt the mechanics, not the principles. Run dual-track. Time-box large projects. Adjust cycle lengths. But keep fixed time, variable scope. - Rebuild trust through consistency. If the system fails and erodes stakeholder confidence, you earn it back by shipping reliably for multiple cycles. - Don't bend the system for every exception. Disruptions should be rare. CEO promises should go through a process. If you're constantly making exceptions, the system has no integrity.

Shape Up is resilient, but it's not infinitely flexible. Know where the boundaries are.

Try This

After your next failed cycle or pitch:

1. **Run a lightweight post-mortem**. Gather the team and ask: "What broke? Was it shaping, dependencies, external factors, or something else?"
2. **Categorize the failure.** Does this belong in Shape Up at all? Should it run in Kanban instead? Was it shaped poorly? Did a dependency fail?

3. **Decide**: Kill the bet, reshape it, or scope hammer and ship a smaller version. Don't let it linger indefinitely.

Failures are data. Use them to tighten your system.

Chapter 10: The Organisational Change Nobody Warns You About

Shape Up isn't a process change.

It's a cultural change that happens to use a process as its vehicle.

You can copy the mechanics—six-week cycles, pitches, betting tables, appetite-weeks—and still have it fail if the culture underneath doesn't shift. Because Shape Up requires trust, autonomy, and comfort with ambiguity that most organizations don't have by default.

This chapter is about what has to change beneath the surface. Not the rituals or artifacts, but the way people work, the way you hire, the way teams collaborate, and what happens to roles that don't fit the new model.

How Different Groups Reacted

The cultural shift wasn't uniform. Different parts of the organization adapted at different speeds and with different levels of resistance.

Engineers: Immediate Buy-In

Engineers loved it.

Almost universally, in every organization where I've introduced Shape Up, the engineering team jumped on it immediately.

Why? It removed the estimation friction.

Engineers hate estimating. Not because they're lazy or uncooperative, but because they're being asked to predict the future based on incomplete information. "How long will this take?" is an impossible question when you haven't seen the code yet, don't know what edge cases you'll hit, and can't control how many times the requirements will shift.

Shape Up replaces that with scope negotiation. Instead of "How long will this take?" it's "Here's the appetite. What can we solve in that time?"

That's a conversation engineers can have honestly. They can say, "In four weeks, we can build X and Y but not Z. If you want Z too, it's a six-week appetite or we cut something else."

That's collaborative problem-solving, not defensive estimation theater.

So engineers bought in fast. The friction came later—when the first few cycles didn't work perfectly, or when we had to adapt the process. Some engineers wanted to be dogmatic about Shape Up, insisting we follow it exactly as written in the Basecamp book.

I had to remind them: the framework is there to assist us, not to dictate. We're using the principles to build a process that fits our people and our context. We're not copy-pasting someone else's process and forcing it to work.

When I introduced Shape Up in my last company, we had 7 engineers. Quick read of the book and everyone was in. Resistance came in those first six months, but not because they wanted to get rid of it, but because they wanted us to adhere to it more dogmatically. As I adapted it to our needs and explained that it wasn't going to be copy/paste as we developed it for ourselves, it became smoother.

Product People: Mixed Reception

Product teams were harder to win over.

The resistance depended on their background and habits.

If they came from spec-heavy environments—where product managers wrote detailed feature specs and handed them to engineers as tasks—Shape Up felt like a loss of control. They were used to defining the solution in detail. Shape Up asks them to define the problem and appetite, then let the team figure out the solution.

That's a shift. It requires trust. It requires letting go.

If they came from customer-facing roles or had engineering backgrounds themselves, the transition was smoother. They already thought in terms of problems and constraints rather than detailed solutions. Appetite-based scoping felt natural.

The other shift for product people: business framing.

I asked them to get closer to the front line—to understand not just what customers wanted, but why it mattered economically. Revenue potential. Cost savings. Strategic value. Expected ROI.

Some product people loved this. It elevated their role from “feature factory manager” to “business strategist.”

Others resisted. “I’m not a finance person. I just build what customers need.”

But in enterprise B2B, “what customers need” has to connect to business outcomes. The pitch template forced that connection. Over time, the product team got better at it. But it took a few cycles—and some uncomfortable conversations—to break old habits.

There were only two product people when I rolled this out. Here there was a lot more resistance initially, because having to come up with an ROI instead of a wish list of features was foreign to them. I actually had one leave because of it. But as time went on and new people were introduced to it, they saw the benefits of proving why we needed something.

The Rest of the Organization: Mostly Unaffected

Sales, customer success, finance, marketing, the board—they didn’t need to understand Shape Up in detail.

What they needed was visibility into what was being built and why.

That’s where the thematic roadmap (Chapter 1) did most of the heavy lifting. Stakeholders could see: - Here are the themes we’re focused on this quarter - Here’s the value proposition for each theme - Here’s what’s shipping and when

They didn't care about pitches, appetites, or betting tables. They cared about outcomes.

The pitch format helped because it included the framing—the business case, the ROI drivers, the strategic rationale. Sales could take a pitch and understand the value prop well enough to sell it. Finance could look at the expected ROI and track whether we delivered. Marketing could see the USPs and start messaging.

> ***"The organization unified around the artifacts, not the methodology."***

That's intentional. You don't need everyone to understand the process. You need the process to produce artifacts that everyone can use.

What Has to Change Culturally

Beyond the team-by-team reactions, there are deeper cultural shifts required. If I tried to fill this next section with how to get these attributes, then the content of this book would explode. So I will give you this list from my perspective, but I can't provide you medicine for these. A couple of them will be alleviated by going this path. A few others not really.

Autonomy and Trust

Shape Up only works if teams have real autonomy.

You can't run a betting table where the team bets on pitches, and then have leadership override those bets mid-cycle because a stakeholder complained. You can't give teams a six-week cycle and then interrupt them in week three with "just one quick thing."

If you don't trust teams to make decisions, Shape Up will feel like theater. The rituals will happen, but the benefits won't materialize.

This means: - Trusting engineers to scope solutions instead of micromanaging implementation - Trusting product people to shape good

pitches instead of second-guessing every bet - Trusting SA/CS teams to manage their Kanban board (Chapter 8) without product approval

If your organization defaults to top-down control, Shape Up will struggle. You'll need to actively build trust before the process can take root.

Comfort with Ambiguity

Shape Up runs on incomplete information.

Pitches are shaped at the right level of abstraction—detailed enough to be credible, but not so detailed that they constrain the team's problem-solving. That ambiguity is a feature, not a bug.

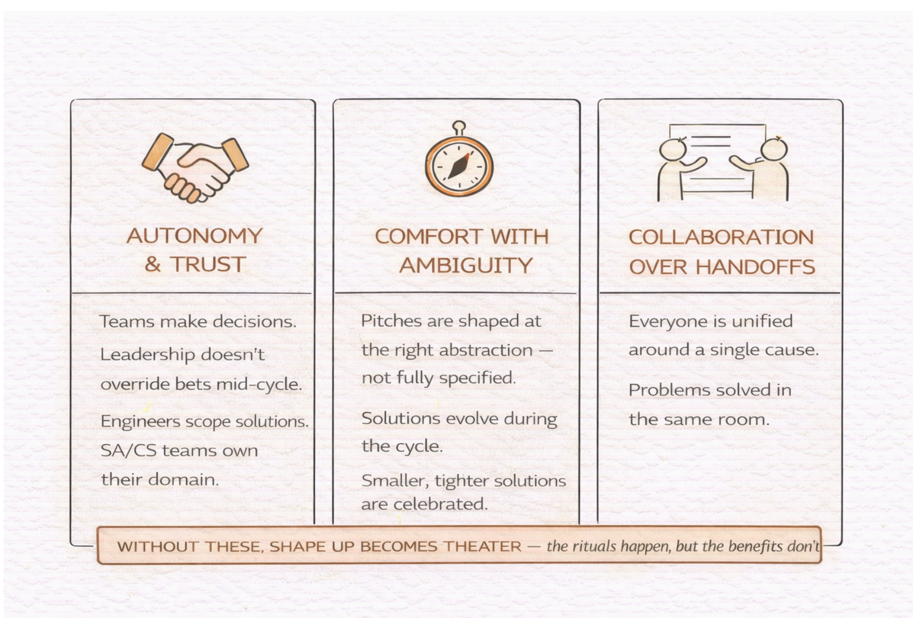
But it makes some people uncomfortable.

Stakeholders who want Gantt charts with every task pre-defined will hate it. Engineers who want perfectly specified requirements before starting will resist. Product people who want certainty about what's shipping will feel anxious.

You can't eliminate the ambiguity. You have to teach the organization to be comfortable with it.

That means: - Communicating clearly that scope is the variable, not time - Setting expectations that pitches will evolve during the cycle as teams discover complexity - Celebrating when teams ship a smaller, tighter solution instead of the original pitch

Over time, the organization learns that ambiguity doesn't mean chaos. It means flexibility.



Collaboration Over Handoffs

This flavor of Shape Up changes how everyone works together.

In a traditional process, everyone keeps pushing the parts in front of them down the assembly line then moves on to the next one. Engineers build what was designed. If there's a mismatch, it gets escalated. The assembly line stops. There's a fight.

In the original Shape Up, designers and engineers collaborate throughout the cycle. The pitch provides a solution sketch—rough wireframes, key flows, critical decisions—but the team figures out the details together.

What I'm proposing is doing this across the whole organization. Product may sit in the middle, but the pitches unify the groups behind a shared understanding. Each contributes to the whole. You throw away the assembly line and build a unified team.

If your culture is built on strict role boundaries and handoff points, this will feel messy at first. But it's also where the best solutions come from.

People Who Won't Adapt

Standard change management rule: some people won't adapt.

They can't or won't shift their habits. They resist the new process. They keep trying to revert to the old way of working.

What do you do?

First, give it time. Some people need to see the system work before they trust it. Let them experience a few successful cycles. Show them the benefits: less estimation friction, clearer priorities, more autonomy, better outcomes.

Second, coach them through the transition. If a product person is struggling with business framing, sit with them during shaping. If an engineer is uncomfortable with ambiguity, walk them through how to navigate a loosely-scoped pitch.

Third, make the tough calls. If someone actively undermines the process—complains constantly, refuses to participate in good faith, drags the team down—you address it like any other performance issue.

This isn't unique to Shape Up. It's change management. Some people adapt. Some don't. If they can't adapt and the process is working for everyone else, they probably aren't a fit for the organization long-term.

Leading the Change, Not Installing a Process

Here's what worked: I didn't install Shape Up. I led a change toward better ways of working, and Shape Up was the vehicle.

That meant: - Selling the benefits to each stakeholder group, not forcing compliance - To sales: "You'll have more reliable visibility into what's shipping and why" - To engineering: "You'll stop defending estimates and start solving problems" - To product: "You'll make strategic bets instead of

managing a backlog" - To finance: "You'll see ROI framing on every piece of work we commit to"

Selling Shape Up to Each Stakeholder Group

ROLE	CORE CONCERN	SHAPE UP BENEFIT
SALES	*When will features ship? Can I make commitments?*	Reliable visibility into what's shipping and why. *Thematic roadmap speaks their language.*
ENGINEERING	*Stop asking me to predict the future.*	Scope negotiation replaces estimation. Autonomy to solve problems within constraints.
PRODUCT	*I'm drowning in feature requests and backlog management.*	Strategic bets on shaped pitches. Business framing elevates the role.
FINANCE	*What's the ROI on this engineering spend?*	Every pitch includes revenue potential, cost savings, and expected ROI.

- Adapting the mechanics to fit the organization, not cookie-cutter implementation
    - Two-horizon betting table instead of one
    - Business framing in pitches instead of pure product thinking
    - Dual-track for customer work instead of forcing everything into Shape Up
- Preserving the principles even when we changed the process
    - Fixed time, variable scope
    - Appetite over estimation
    - Betting table as the decision gate
    - Cool-down exists and must be protected

The principles are what make Shape Up work. The mechanics are just how you apply those principles to your context.

If I'd tried to copy-paste the Basecamp process exactly, it would have failed. Enterprise B2B is not a product company. We had different constraints, different stakeholders, different types of work.

But the principles held. And by holding to the principles while adapting the mechanics, we built something that worked.

What Surprised Me (Nothing, Actually)

Honestly? Nothing surprised me about the organizational change.

Every time I've introduced Shape Up, the pattern has been the same: - Engineers excited immediately - Product people take a few cycles to adjust - The rest of the organization doesn't need to care about the methodology as long as the artifacts make sense

The fact that it's predictable is a good sign. It means the resistance isn't about Shape Up specifically. It's about change in general—and the resistance maps to role-based habits, not fundamental flaws in the approach.

> ***"The Biggest Lesson: Organizational change is always more than throwing a process at people."***

You have to lead it. You have to sell the benefits. You have to link the value to what each group cares about. You have to show people how this makes their work easier, not harder.

If you do that, the process takes root. If you don't, it becomes one more failed methodology that people shrug off and ignore.

What This Means for You

Before you introduce Shape Up, ask: - Do we have trust and autonomy? If not, build that first. Shape Up won't create trust; it requires it. - Can our culture handle ambiguity? If not, start small. Run one team on Shape Up and let the results speak. - Are we willing to adapt the mechanics? If you're

looking for a cookie-cutter process, Shape Up isn't it. You'll need to shape it to fit your organization. - Can we hold to the principles? Fixed time, variable scope. Appetite over estimation. Those are non-negotiable. Everything else is flexible.

Shape Up is resilient, but it's not magic. It works when the culture underneath supports it.

Build that culture first. Then the process will thrive.

Try This

Before rolling out Shape Up broadly:

1. **Map stakeholder concerns.** For each group (engineering, product, sales, finance, CS), write down: "What do they care about? What problem does Shape Up solve for them?" Sell the benefits in their language, not methodology jargon.

2. **Run a pilot with one team.** Don't go organization-wide immediately. Let one team run Shape Up for three cycles. Document what works and what breaks. Use those lessons to adapt before scaling.

3. **Identify your principles vs. mechanics.** Write down: "What must stay true?" (principles) and "What can we change?" (mechanics). This clarity will save you from dogmatic arguments later.

Change is leadership, not installation. Lead it well.

Chapter 11: Getting Started Monday Morning

You've read the book. You're convinced. Monday morning, you're going to start running Shape Up.

Here's your 90-day playbook.

This isn't theory. It's the concrete, actionable steps I'd take if I had to implement Shape Up in a new organization tomorrow, knowing everything I know now from doing it multiple times.

Some of this will work for you. Some of it won't. Your context is different from mine. But this gives you a starting point—a path through the first three months that avoids the most common failure modes.

Let's go.

Before You Start: Get Two Things in Place First

If I had to start over, I wouldn't begin with cycles and betting tables.

I'd start with two foundational pieces that make everything else easier:

1. The Thematic Roadmap (Chapter 1)

Get the roadmap conversation started. Move from feature lists to themes. Get stakeholders—sales, CS, finance, the board—aligned on the strategic direction for the quarter.

This does two things: - It creates the context for shaping. You're not shaping random features. You're shaping work that serves a strategic theme. - It gets stakeholders bought in before you change the process. They see value in the roadmap shift, which makes them more willing to trust the process changes that follow.

You don’t need a perfect thematic roadmap on day one. Just start the conversation. “Here’s what we’re focusing on this quarter and why.” Make it about outcomes, not deliverables.

2. Business Framing for the Product Team (Chapter 3)

Train your product team to think in terms of business framing. Why does this work matter? What’s the revenue potential? What’s the cost savings? What’s the strategic value?

This takes time. Product people who are used to writing feature specs need to shift to writing business cases. Give them a few weeks to practice before you run your first betting table.

Why start here? Because shaping is the hardest part of Shape Up to get right. If your product team is still learning how to frame the “why,” their pitches will be weak. Weak pitches make betting tables frustrating. Frustrating betting tables kill momentum.

Get the framing muscle built first. Then layer in the rest of the process.

Image: Chapter11.1

Week 1: Communicate, Align, Read

Day 1: Communicate the Change

Gather your teams—product, engineering, design—and tell them what’s changing and why.

Keep it simple: - “We’re moving to six-week cycles with two-week cool-downs. Here’s why.” - “We’re replacing estimation with appetite. Here’s what that means.” - “We’re running a betting table to decide what we work on. Here’s how it works.”

Don’t oversell it. Don’t promise that Shape Up will solve all their problems. Just say: “We’re trying something new. It’s designed to reduce

friction around estimation and give us clearer priorities. We'll run it for three cycles and see if it works."

Set the expectation: this is an experiment, not a mandate.

Day 2-3: Align with Stakeholders

Meet with sales, customer success, finance, and anyone else who depends on the product roadmap.

Explain: - The thematic roadmap (if you've already started that conversation, reinforce it) - The cycle rhythm: "Every six weeks, we ship a batch of work. You'll have visibility into what's coming." - The betting table: "We're making strategic trade-offs explicit. You'll see why we chose to build X instead of Y."

Ask for patience. The first cycle or two might feel bumpy. Pitches might not ship perfectly. The roadmap might shift as we learn. That's expected.

Day 4-5: Read and Absorb

Have your core team read the original Shape Up book by Basecamp. It's free online. It's short. It's well-written.

You don't need everyone to agree with everything in it. You just need them to understand the principles so they can engage with the process intelligently.

After they've read it, have a discussion: "What resonates? What feels like it won't work in our context? What do we want to adapt?"

This primes the team to think critically about the process rather than blindly following it.

Picking the Pilot Team

Don't roll out Shape Up to the entire organization on day one.

Start with one small team. Four to five people. Run it for three cycles. Learn. Adjust. Then scale.

Criteria for Picking the Pilot Team:

1. High trust, low politics. Pick a team where people already work well together. You don't want interpersonal friction to confound the process experiment.

2. Experienced, not junior. Shape Up requires autonomy and problem-solving. A team of junior engineers will struggle. Pick people who can navigate ambiguity.

3. Product work, not customer work (at least initially). Start with internally-controlled work where you can set appetites and control scope. Don't pilot Shape Up on customer-driven projects with external dependencies.

4. Willing participants. Don't force this on a skeptical team. Find people who are curious and open to trying something new.

5. Visible but not critical. Pick work that stakeholders care about, so success is visible. But don't bet the company on the first cycle. You need room to fail safely.

The First Shaping Session

Gather your product team (or whoever will be doing the shaping). Pick one problem to shape.

Remember, this is the first step. As you move forward, you will be filling in a lot more details than at the beginning. You want to grow into the right amount of detail. And at the beginning, you're getting used to using appetite instead of estimations.

What to Expect:

It will feel loose. Shaping isn't writing detailed specs. It's defining the problem, setting an appetite, sketching a solution at the right level of abstraction, and identifying no-gos.

Walk through the pitch template from Chapter 3: - Problem: What are we solving and why does it matter? - Appetite: How much time is this worth? - Solution sketch: Rough wireframes, key flows, critical decisions - Rabbit holes and no-gos: What could go wrong? What are we explicitly not doing? - Business framing: Revenue potential, cost savings, strategic value, expected ROI

The first pitch will take longer than you expect. Maybe two to three hours. That's normal. You're building the muscle.

Example First Pitch: Customer Data Import

Here's what a first pitch might look like, using the bank data import example from Chapter 3:

Business framing: - Revenue potential: $450K (one deal in final stages) - Cost savings: N/A — this is a revenue-driven bet - Strategic value: Opens up a new market and segment for us - Expected ROI: Break-even after first customer goes live

Problem: Mid-tier bank customers can't onboard without manual data entry. Their existing systems export CSV files with transaction data, but we don't have an import tool. This is blocking deals in final stages—one customer ($450K ARR) identified data import as a blocker in the technical review.

Appetite: 2 weeks, 1 engineer. This is hygiene, not innovation. Competitors have this. We need it to close deals, but it's not worth more than a small bet.

Solution sketch: - Admin uploads CSV file (we support one standard format initially—the bank's export format) - System validates columns, shows preview of first 10 rows - Admin confirms, system imports transactions in background - No real-time processing—overnight batch is fine for the initial version

Rabbit holes and no-gos: - We're NOT building a universal CSV parser. We support one bank's format for this cycle. - We're NOT handling errors gracefully yet. If the import fails, admin gets an error message. That's it.

Notice what's NOT in this pitch: detailed error handling, multi-format support, real-time validation, progress indicators. Those are nice-to-haves. The appetite is two weeks, so we shaped the simplest version that solves the core problem. That's what "the right level of abstraction" looks like in practice.

Common First-Pitch Mistakes:

- Too much detail. The pitch isn't a spec. Resist the urge to define every edge case.
- No clear problem. "We should build X" isn't a pitch. "Customers are struggling with Y, and here's how X solves it" is a pitch.
- Wishful thinking on appetite. Be honest. If it feels like a six-week problem, don't call it a four-week appetite just to make it fit the cycle.

After the session, circulate the pitch to the team that will build it. Get their feedback during cool-down. Does the solution make sense? Are there hidden complexities? Adjust before you bet on it.

The First Betting Table

Keep it simple.

Who's in the room: Product leadership, engineering leadership, maybe a founder or exec. Small group—three to five people max.

What's on the table: The pitch you just shaped, plus maybe one or two others if you have them.

The decision: Do we bet on this for the next cycle?

How to Run It:

1. **Present the pitch.** Walk through the problem, appetite, solution sketch, and business framing. Five minutes, maybe ten.
2. **Discuss**. Does this solve a real problem? Is the appetite right? Is the business case solid? Is the team confident they can ship it?
3. **Bet or pass**. Either commit to the pitch or don't. No "let's refine it and come back next week." If it's not ready, it's not ready. Move on.

For your first betting table, you probably only have one or two pitches. That's fine. Bet on what you have and accept that you're under-capacity for this cycle. You're learning.

Don't try to fill every appetite-week. Don't scramble to create pitches just to have more options. Start small. Build confidence.

The First Cycle: Setting Expectations

The first cycle will not be perfect.

Set that expectation clearly with the team: "This is a learning cycle. We're figuring out how this works. Some things will break. That's okay."

What Success Looks Like:

Not: "We shipped exactly what was in the pitch."

Yes: "We learned how to navigate ambiguity, we scoped to fit the appetite, and we shipped something valuable."

If the team scope-hammered and cut nice-to-haves to finish in six weeks, that's success. If they used part of cool-down to finish because they misjudged the appetite, that's fine for cycle one. Track it. Learn from it. Tighten it next time.

During the Cycle:

Don't micromanage. Let the team run. Check in async (those three daily questions from Chapter 6). If they hit a blocker, help them unblock. But resist the urge to hover.

Don't add work mid-cycle. This is critical. If a stakeholder asks for "just one small thing," the answer is no. Protect the cycle. Show the team that the six-week boundary is real.

Expect ambiguity to surface. The pitch will evolve as the team digs in. That's normal. As long as they're solving the core problem within the appetite, they're on track.

At the End of the Cycle:

Ship what's done. Even if it's not perfect. Even if there are nice-to-haves left on the table. Ship it.

Run a lightweight retro. Gather the team. Ask: - What worked? - What was harder than expected? - What should we change for the next cycle?

Don't skip this. The retro is where you learn.

Months 2-3: Expanding, Iterating, Building Confidence

After the first cycle, run it again. Same team. New pitch. Second cycle.

This is where the rhythm starts to feel real. The team knows what to expect. Shaping gets faster. The betting table gets easier.

By the Third Cycle:

- You should have a small portfolio of pitches to choose from at the betting table
- The product team should be getting better at business framing
- The engineering team should be getting better at scoping to appetite

- Stakeholders should be seeing reliable output every six weeks

If all of that is true, you're ready to scale.

Add a second team. Run both teams on the same six-week rhythm. Hold one betting table for both.

Or, if you need to tackle customer-driven work, start the dual-track model (Chapter 8). Keep one team on Shape Up, give another team a Kanban board for service-layer work.

What to Iterate:

- Cycle length. Maybe four weeks + two weeks works better than six weeks total. Experiment.
- Pitch format. Add fields that make sense for your business. Remove ones that don't.
- Betting table structure. Maybe you need Horizon 1 and Horizon 2 (Chapter 4). Maybe you don't.

> ***"Don't iterate on the principles. Keep fixed time, variable scope. Keep appetite over estimation. Keep the betting table as the decision gate. Those are load-bearing."***

How to Know If It's Working

Here are the leading indicators that Shape Up is taking root:

Positive Signs:

1. Teams stop asking about estimates. They start asking, "What's the appetite?" instead of "How long will this take?"

2. Pitches get tighter over time. The first few are loose. By cycle three or four, product people are shaping at the right level of abstraction.

3. Stakeholders stop asking, “When will X ship?” They start asking, “What’s in the next cycle?” They’ve internalized the rhythm.

4. The betting table gets faster. First one takes an hour. By cycle three, it’s thirty minutes because the pitches are clear and the trade-offs are transparent.

5. Teams protect cool-down. Someone says, “Can we make sure this doesn’t bleed into cool-down?” That’s the moment you know it’s working (Chapter 7).

When to Expect These Signs

Don’t panic if you’re not seeing all five by day 60. These signs appear on a predictable timeline.

By Cycle 1 (Day 42): Don’t expect any of these signs yet. The first cycle is about learning, not perfection. You’re just trying to ship something and complete the loop. If the team finished the cycle and shipped something—even imperfectly—that’s enough.

By Cycle 2 (Day 70): You should start seeing sign #1. Engineers should be asking “What’s the appetite?” instead of “How long will this take?” If they’re still defaulting to estimates at cycle two, the framing hasn’t landed. Go back and reinforce it—revisit the appetite conversation from Chapter 2.

By Cycle 3 (Day 90): You should see signs #2, #3, and #4. Pitches are getting tighter. Stakeholders are internalizing the rhythm. The betting table is getting faster. If you’re not seeing these by cycle three, something is broken. Either shaping isn’t working (go back to Chapter 3), or the cultural prerequisites aren’t in place (Chapter 10).

Sign #5 (teams protect cool-down): This is the last one to appear. Don’t expect it until cycle four or five. It means the culture has genuinely shifted—people value the rhythm enough to defend it. When someone in the team says “we’re not bleeding into cool-down,” you’ve won.

Warning Signs:

1. Pitches keep failing. If you're killing bets or bleeding into cool-down every cycle, something's wrong. Either shaping is broken, or appetites are systematically under-scoped.

2. The betting table is political. If people are lobbying for their pitches or arguing about priorities, the business framing isn't strong enough. Go back to Chapter 3 and tighten it.

3. Stakeholders are frustrated. If sales or CS keep saying, "We don't know what's shipping," the roadmap visibility is broken. Fix the thematic roadmap first.

4. Teams are burned out. If cool-down is consistently getting invaded or teams are working weekends, you're not holding the boundaries. Protect the cycle. Protect cool-down.

5. People revert to old habits. If engineers start asking for story points or product people start writing detailed specs, the culture shift hasn't happened. Go back to Chapter 10 and reinforce the principles.

Common First-90-Day Mistakes

Mistake 1: Rolling Out Too Fast

Don't go organization-wide on day one. Pilot with one team. Learn. Adjust. Scale when it's working.

Mistake 2: Being Dogmatic

Don't copy-paste the Basecamp process or what I showed you. Adapt it to your context. Preserve the principles, change the mechanics.

Mistake 3: Skipping the Roadmap Work

If you jump straight into cycles without aligning stakeholders on the thematic roadmap, you'll spend the whole cycle defending your priorities. Do the roadmap work first.

Mistake 4: Under-Investing in Shaping

Weak pitches kill the system. Spend time teaching your product team how to shape. This is the hardest skill to build and the most important.

Mistake 5: Not Protecting the Boundaries

If you let work slip into cycles mid-stream, if you let cool-down get invaded, if you extend timelines to fit scope—you've lost the discipline. The system falls apart. Hold the line.

Final Thoughts: What to Do Monday Morning

Here's the checklist:

Week 1: - [ ] Communicate the change to your team - [ ] Align with stakeholders - [ ] Have the team read the Shape Up book - [ ] Start the thematic roadmap conversation - [ ] Train product team on business framing

Week 2-3: - [ ] Pick your pilot team (4-5 people, high trust, experienced, willing) - [ ] Run your first shaping session - [ ] Circulate the pitch for feedback

Week 4: - [ ] Run your first betting table (keep it simple, one or two pitches) - [ ] Kick off the first cycle

Weeks 5-10 (First Cycle): - [ ] Let the team run - [ ] Check in async - [ ] Protect the cycle boundaries - [ ] Ship at the end of week 10

Week 11-12 (Cool-Down): - [ ] Run a lightweight retro - [ ] Shape pitches for cycle two - [ ] Let the team recover

Repeat for Cycles 2 and 3.

By day 90, you'll know if this works for your organization.

If it does, scale it. If it doesn't, figure out why. Maybe the culture isn't ready (Chapter 10). Maybe you're forcing the wrong type of work into Shape Up (Chapter 8). Maybe the principles are sound but the mechanics need more adaptation (Chapter 9).

Shape Up is resilient. But it requires discipline, trust, and a willingness to adapt.

Start small. Learn fast. Build confidence. Scale when it works.

You've got this.

Chapter 12: AI in the Process

I'm known as the worst engineer on the team. By design.

My role is product, strategy, some fundraising, little bit of selling, not day-to-day code. But one afternoon, we needed three payment service provider integrations built. Not one. Three. Different providers, different APIs, different documentation formats.

I pulled down the API specs, stored the documentation locally, opened three parallel sessions, and pointed each one at a different provider. "Here's the documentation. Here are our internal interfaces. Build the integration."

Then I went and did other work.

Twenty minutes later, the first one was done. Forty-five minutes later, all three were more or less complete. They needed integration testing — putting them into an environment, checking that data actually flowed to the sandbox accounts. But the core implementation was finished.

I did this. The worst engineer on the team. Without asking anyone for help.

That's what AI does to the operating system I've described in this book. It doesn't change the system. It accelerates the people within it.

The System Doesn't Change. The Speed Does.

Everything I've described in the previous eleven chapters — shaping, pitching, betting, building, cool-down, the dual-track model — all of that still applies. AI doesn't replace any of it. The betting table still needs humans having real discussions about what's worth building. Appetites are still strategic decisions. Cool-down is still sacred.

What changes is how fast and how thoroughly people can do the work within that system.

We ran Shape Up for several years before generative AI showed up. All the activities I'm about to describe — competitive research, pitch writing, technical proposals, integration work — we did all of that manually. Somebody maintained competitive analysis documents somewhere. Hopefully they were updated. You'd go look, find they were six months stale, and do the research again yourself. Going through sales notes, customer feedback, regulatory documents — all of that was grinding, time-consuming work.

Now you can get most of that information on demand, in real time, synthesized and summarized. The shaping work that used to take four to six weeks for a complicated feature — gathering information across competitors, customers, internal feedback, regulatory requirements, then synthesizing all of that into a pitch with full business framing — has been reduced, in some cases, to under an hour.

That's not an exaggeration. That's what happened when I got the workflow right.

The Document Is the Anchor

Here's the concept that makes AI actually useful in this process, rather than just a novelty: the document is the anchor.

Think about how shaping works. You start with something cloudy — a problem, an opportunity, a customer need — and you gradually refine it into something concrete. A pitch. The pitch document is the memory of everything you've figured out so far.

If you just told an AI "create me a pitch about credit payment plans," you'd get something generic and unusable. It wouldn't know your customers, your competitive landscape, your platform constraints, or your strategic priorities. It would be all over the place.

But if you treat the pitch document as the core — the persistent artifact that you keep coming back to across multiple sessions — everything changes.

Here's how this actually worked on one of the most complex pitches I built: a complete revamp of our credit payment plans. Offers, early repayment options, default handling, the works. A lot of moving parts and a lot of complexity.

I didn't try to do it all in one session. I worked in layers.

Layer one: competitive research. I ran three separate sessions, one per competitor. Each session focused on how that competitor handled payment plans — their feature set, how they positioned it, how they sold it, what problems it solved. Each session produced a focused competitive analysis document.

Layer two: synthesis. I took those three documents into a new session and asked: what's the common ground? Where are they all going? What are they missing? I ended up with a consolidated view of the competitive landscape for this specific feature area.

Layer three: gap analysis. I opened a new session with my pitch draft and the competitive synthesis. "Look at what I've got so far. Look at what the competition is doing. Where are the gaps?" The AI identified specific things I was missing. At that point, I made the strategic decisions — which gaps actually mattered, which ones were must-haves versus nice-to-haves, which ones were irrelevant to our positioning.

Layer four: customer feedback. Same process. I pulled customer success notes, support conversations, and feedback related to payment plans. Summarized them into a focused document. Fed that into the pitch and decided what to keep.

Layer five: workshop input. We'd done a virtual whiteboard session with stakeholders. I copied the whiteboard content into a text document and

used it as additional context — what did the workshop participants say was necessary for this feature to succeed?

Each layer was a separate session. Each one injected new context into the pitch document without overloading any single conversation. The pitch got progressively sharper, more grounded, more thorough.

At the end, I had a pitch that covered the problem from every angle — competitive positioning, customer needs, stakeholder input, technical implications. I still did the ROI and strategic importance assessment myself. That's judgment work. But the underlying research, synthesis, and gap analysis that would have taken weeks was done in a fraction of the time.

The key insight: work on the document, not in the chat. The chat is disposable. The document persists. Every time you start a new session, you point the AI back at the document and inject whatever new context is relevant. The document gets better. The chat gets thrown away.

This applies beyond pitches. Technical proposals during cool-down work exactly the same way. Engineers use the pitch as context, layer in their technical stack knowledge, and produce a technical proposal that's been checked against the product requirements. The AI helps them cover edge cases and verify that the proposal addresses everything the pitch describes.

Workslop Will Kill Your Credibility

Now here's the warning.

I made a mistake with this. Early on, I was in a rush. I had the AI generate a pitch quickly, without feeding in much context. I threw some figures into the at-a-glance table for the ROI, but I didn't have time to really work through whether they made sense.

The team caught it almost immediately. The pitch didn't quite make sense. It had logical gaps. No solid link to our platform. And now I had a credibility problem — because if you're the person passing AI-generated

work downstream without putting your expertise into it, people stop trusting the information you send them.

This is the workslop problem, and it's growing across every company that uses generative AI. Someone has AI create a pitch, a slide deck, a report, a proposal. It looks polished. It reads well. They send it along.

The person downstream spends more time and energy cleaning it up, fact-checking it, and figuring out what's actually true than if the original author had just done the work properly. The AI output looks complete, but it lacks the judgment, context, and expertise that only the human can provide.

Here's the rule: your expertise is the glue. AI handles the research, the synthesis, the drafting, the gap analysis. You handle the judgment — what's actually worth building, what the numbers really are, what the strategic implications look like, whether this pitch is honest or wishful thinking.

If you skip that step, you'll erode trust faster than you built it. And in the operating system I've described — where the betting table runs on pitch quality, where transparency reduces politics, where accountability loops prevent gaming — trust is the currency that makes everything work.

Don't let AI debase it.

During the Cycle: The Right Documents Make AI Useful

That's the shaping side. AI changes the build phase too, but not in the way most people expect.

Here's what creates the real acceleration: having the right documents in place before the AI touches any code.

If you've done the work I described above — a solid pitch document with business framing, a technical proposal that maps the solution to the architecture — then the AI has everything it needs to be genuinely useful

during the build. Without those documents, the AI is guessing. With them, it's executing against a clear brief.

If you've invested in a proper design library — reusable components, documented patterns, consistent standards — AI turns that investment into a multiplier. You can say: "I need an entirely new section of the interface. Here's the existing design library. Add the navigation, build me a live prototype." An hour or two of iteration, and you have a working prototype that matches your design standards. That prototype becomes code in the appropriate framework, handed to engineers alongside the pitch and technical proposal.

If your APIs are well-documented and your front end follows consistent standards, the AI can stitch the implementation together — because it has the design, the pitch context, and the technical proposal all available.

Here's what this looked like in practice. I once had to create an entirely new interface for a completely new type of user in on of my platforms. I had already created the pitch, and I had our design library. I literally gave an AI both, and then spent a couple of hours og

One of our engineers had a pitch for a new section of the platform — a reporting dashboard that pulled data from three internal services. He had the pitch document with the business context: what the customer needed to see, why it mattered, how it was positioned. He had the technical proposal he'd written during cool-down: which APIs to call, how to aggregate the data, what the caching strategy should be. And he had the design library with our component standards.

He opened an AI session, pointed it at all three documents, and said: "Build the front-end scaffolding for this dashboard. Use the existing component library. Wire it to these API endpoints."

The first pass took about thirty minutes. It wasn't perfect — the data aggregation logic needed reworking, and there were edge cases around empty states that the AI missed. But the scaffolding was there. The layout matched our design standards. The API calls were structured correctly. The

engineer spent his time on the hard parts — the caching logic, the edge cases, the performance tuning — instead of writing boilerplate for the hundredth time.

That's the shift. The engineer didn't write less code. He wrote less *boring* code. The time he saved on scaffolding went straight into the architectural decisions that actually required his expertise.

Engineers as Craftsmen

That example illustrates the broader role shift. Engineers become craftsmen. They focus on the deep thought work: how do I architect a solution that actually solves the business problem? How do I make sure the AI-generated code meets our standards? How do I steer multiple agents without them going off the rails?

The engineer in the dashboard example was running the AI, reviewing its output, catching its mistakes, and making the decisions the AI couldn't make — what to cache, how to handle failures, where the performance bottlenecks would be. That's steering. That's craft.

The same shift happens for product people. With AI handling the research grinding — competitive analysis, regulatory summaries, customer feedback synthesis — product people can operate at a higher level. They spend their time figuring out what something is actually worth, not digging around trying to assemble the information they need to make that judgment.

We didn't see much resistance. We'd been experimenting with AI models since they first came out, and the framing was always the same: "This will eventually work. We need to stay ahead of it." Engineers saw it as elevation — less boilerplate, more architecture. Product people saw it as liberation — less digging, more deciding.

There's a prerequisite, though: code standards matter more than ever. The more well-standardized your codebase is, the more reliably AI generates code that matches your norms. If your codebase is messy and inconsistent, the AI will produce messy and inconsistent output, and the engineer

spends more time fixing it than they saved. Good engineering hygiene is no longer just a nice-to-have. It's the foundation that makes AI-assisted development work.

The Dual-Track Shift

Remember the dual-track model from Chapter 8 — Shape Up for product, Kanban for service-layer work?

AI hit the Kanban side hard. All those integration pieces — the client integrations, the onboarding work, the service-layer implementations that the SA and CS teams managed — collapsed in terms of time required.

That's the payment service provider integration story I opened with. Three integrations, done in parallel, by the least technical person on the team. The code still needed testing. You still had to verify that data actually flowed correctly to sandbox accounts. But the brute-force implementation work that used to consume days of an SA's time was reduced to minutes.

What did that free up? SA capacity. SAs who used to spend cycles grinding through integration code could now focus on higher-value activities — the features needed for large contracts, the strategic customer work, the relationship management that actually drove revenue.

The Kanban track didn't change structurally. It's still Kanban. Work still flows through based on priority and external timelines. But the throughput increased dramatically because the implementation work got faster.

The Downstream Cascade

Here's the part that I enjoyed the most.

Once you have a solid pitch document and a solid technical proposal — both refined with AI's help — those two documents become the basis for everything downstream.

The pitch document contains the problem being solved, why it's valuable to customers, how it's positioned against the competition. That's

marketing and sales enablement material, already written. AI can take that pitch and generate product marketing copy, sales talking points, and positioning documents.

The technical proposal contains the architecture decisions, the implementation approach, the technical trade-offs. AI can use that to generate release notes, API documentation, and internal technical communications.

This means you can start go-to-market preparation while the feature is still being built. By the time the release ships, the marketing text is drafted, the sales team knows how to position it, and the release notes are ready.

I've had cycles where the pitch and technical proposal were used as the direct basis for release notes. Not rewritten from scratch. Transformed from the source documents that already contained all the relevant information.

It plays through the whole process. The pitch feeds shaping. Shaping feeds the build. The build feeds documentation. Documentation feeds go-to-market. One set of well-maintained documents, cascading through every downstream process, with AI handling the transformation at each stage.

That's not a minor efficiency gain. That's a fundamental change in how much brute-force work disappears from the product development cycle.

What Hasn't Changed

The betting table hasn't changed. It's probably the last place I'd expect AI to show up.

Think about what actually happens at the betting table. A small group of people — product, engineering, leadership — sit in a room and make judgment calls about what's worth building. They're reading body language. They're weighing a pitch's strategic value against the team's current morale. They're navigating the politics of saying no to a stakeholder's pet project. They're making trade-offs that require

understanding the business, the market, the team, and the moment — all at once.

That's not a task you can hand to AI. AI can help you prepare for the betting table — synthesize pitch summaries, model appetite-week scenarios, generate at-a-glance comparisons. But the actual decision? That's where your expertise, your market instinct, and your ability to read the room earn their keep. The betting table is the place where the human operating system does what no algorithm can: make a judgment call that balances strategy, politics, capacity, and timing in a way that the team trusts enough to commit to.

Cool-down hasn't changed much either — yet. Engineers still use it for bug fixes, technical debt, and preparing technical proposals for the next cycle. But I expect this to evolve. AI agents are getting better at handling well-defined technical tasks: running through a backlog of minor bugs, identifying and fixing code quality issues, updating dependencies, generating documentation from code changes. As long as the right context documents exist and the codebase is well-structured enough for the AI to work safely, some of the mechanical cool-down work could increasingly happen in the background. We're not fully there yet for most teams. But the engineers who are already comfortable steering AI during the build cycle will be the first to extend that into cool-down.

The principles haven't changed at all. And this is the part I want to be most explicit about, because it's the thread that runs through every chapter of this book. Fixed time, variable scope. Appetite over estimation. Betting table as the gate. Cool-down as sacred space. These aren't process mechanics that AI makes obsolete. They're decision frameworks that require human judgment to operate.

Fixed time, variable scope is a *choice* — a deliberate trade-off that says "we'd rather ship something valuable in six weeks than ship something perfect in six months." No AI makes that choice for you. Appetite over estimation is a *stance* — it says "we decide what this is worth before we figure out what's possible." That's a strategic act, not a computational one.

The betting table is a *conversation* — it works because people trust each other enough to say "not this cycle" and have that decision stick.

AI makes the people faster. It doesn't make the principles obsolete.

What This Means for You

AI in this process is not about replacing judgment with automation. It's about removing the grinding work so your people can focus on the thinking that actually matters.

Product people stop digging and start deciding. Engineers stop writing boilerplate and start architecting. SAs stop grinding through integration code and start focusing on customer relationships. Everyone operates at a higher level because the lower-level work — research, synthesis, drafting, implementation — gets handled faster.

But only if you maintain the human layer. The expertise. The judgment. The accountability.

The moment you let AI generate a pitch and just send it downstream without putting your brain into it, you've got workslop. And workslop erodes the trust that makes this entire operating system function.

Use AI to go faster. Use your expertise to go right.

Try This

Start with one pitch:

1. **Pick a pitch you're working on** — ideally something with competitive complexity or multiple stakeholder inputs. Don't start with something simple. Start with something where the research burden is real.

2. **Work in layers**. Run competitive research in separate sessions, one per competitor. Synthesize. Pull in customer feedback separately.

Feed each layer into the pitch document incrementally. Keep the document as the anchor.

3. **Do the judgment work yourself**. When the research is synthesized and the gaps are identified, make the strategic calls. What's worth including? What's the real ROI? What's the honest appetite? Don't let the AI decide those things for you.
4. **Track the time**. How long did the AI-assisted pitch take versus how long it would have taken manually? That's your baseline for the speed gain — and your evidence when someone asks whether AI is actually helping.

One pitch. Four layers. Your judgment on top.

And welcome to tomorrow.

Appendix: Templates & Artifacts

The templates and artifacts referenced throughout this book — including the pitch template, betting table agenda, appetite-weeks calculator, thematic roadmap template, board communication script, and FAQ for skeptics — are available online at:

https://www.inflectionadvisory.io/

These are practical, steal-ready documents you can adapt for your own organisation.

c

ABOUT THE AUTHOR

Michael Backes has spent over 25 years building enterprise software products and the organisations that ship them. He has founded and co-founded multiple companies, served as a venture builder at Liquid Labs — where he backed 16 companies with €87 million in capital and supported six exits — and worked across the full spectrum of B2B product leadership, from first sales calls to scaled platform operations.

He spent six years as CTPO at Receeve, an enterprise AI platform serving banks and financial institutions across Europe, where he built and ran the operating system described in this book. During that time he ran Shape Up in a live enterprise B2B environment through customer escalations, seven-figure RFPs, rapid team growth, and the arrival of AI as a development accelerant.

Today Michael leads Inflection Advisory, providing fractional CTPO and advisory services to companies navigating transitions.

Connect with Michael at inflectionadvisory.io or on LinkedIn.