Agile in the Enterprise

Delivering Software at Scale Under Compliance, Governance and AI

FEDRAMP · NIST · CMMC · DEVSECOPS · AI-NATIVE DELIVERY

Chetan Sanghani

FOURTH EDITION

AGILE IN THE ENTERPRISE

Delivering Software at Scale Under Compliance, Governance and AI

Fourth Edition

Chetan Sanghani

Copyright

About the Author

Chetan Sanghani is a Technical Product Manager in enterprise government software, where he leads Agile delivery across 25+ cross-functional teams building government-compliant SaaS platforms for federal and state agencies.

His daily work sits at the intersection of rapid iterative delivery and strict regulatory compliance — FedRAMP authorisation, IL5 and IL6 requirements, NIST 800-53 controls and CMMC obligations — a combination most Agile literature does not address, and the reason this book exists.

He holds Certified Scrum Product Owner (CSPO), Agile Scrum Master (ASM), PMI Agile Certified Practitioner (PMI-ACP) and Certified Associate in Project Management (CAPM) certifications, and is recognised as a C# Corner MVP for his contributions to the developer community, with published articles on enterprise architecture, Agile practice and AI-driven development.

His current work focuses on integrating AI across the software development lifecycle — and on measuring that integration honestly enough to defend the numbers. Chapter 19 sets out what his programme measured, how, and what those measurements do not prove.

He resides in the United States and can be reached through LinkedIn and C# Corner. He welcomes feedback from readers and fellow practitioners.

Scope, Currency and Limitations

Read this page before Part IV.

This book is delivery guidance. It is not legal, regulatory, or certification advice. Nothing in it constitutes an assurance that following its practices will achieve or maintain an authorisation, certification, or finding of compliance. Authorisation decisions rest with your authorising officials and assessors, not with any book.

Regulatory content was verified against primary sources in August 2026. The frameworks discussed here — FedRAMP, NIST SP 800-53, CMMC, FIPS, the NIST AI Risk Management Framework, ISO/IEC 42001, the EU AI Act — are living documents, and several were undergoing active change while this edition was being written. Three examples from the four weeks before this manuscript locked:

- FedRAMP published its Consolidated Rules for 2026, replacing impact levels with certification classes.
- The phased rollout of CMMC was suspended pending a programme review.
- FIPS 140-2 validations moved to historical status.

That rate of change is not unusual, and it shapes how this book is written. **Principles appear in the body text; volatile specifics appear in dated STATUS AS OF boxes** that carry their primary-source reference. Where you find a box whose date has aged, the surrounding argument still holds — verify the specific at the source named in the box.

Corrections and updates between editions are announced through the author's LinkedIn and C# Corner profiles. Where a status box has aged past the point of usefulness, that is where the revision will be noted.

The views expressed in this book are the author's own and do not represent the positions, strategies or opinions of any employer, client or organisation with which the author is or has been associated.

On the author's data. This book reports measurements from one organisation, one technology stack, one regulatory context, and one point in time. Those numbers are offered as evidence of what happened there, with their methodology and limitations stated. **They are not industry benchmarks and must not be read as what your organisation should expect.** Where methodology could not be reconstructed, the text says so.

List of Figures

41 exhibits. Every figure carries alt text; in an ebook reader the description is available to assistive technology.

- Figure 23.4 — Transformation outcomes — improvements and new capabilities, separated

Chapter 24 · The First Ninety Days

- Figure 24.1 — The first ninety days

How to Read This Book

This is a reference, not a narrative. Almost nobody should read it front to back on the first pass.

The book has six parts. Part I establishes the operating model everything else hangs from — read it regardless of your role. After that, follow your track.

IF YOU ARE…	READ	SKIM	SKIP
Executive / VP Engineering	1, 2, 11, 13, 17, 21, 22, 23, 24	3, 8, 20	6, 7, 15
Product Manager / Product Owner	1, 2, 5, 9, 11, 17, 19, 21	3, 4, 13	15, 16
Scrum Master / RTE / coach	3, 4, 6, 7, 8, 10, 22, 24	11, 13	21
Engineer / Architect / Tech Lead	12, 16, 17, 18, 20, 21	2, 11	5, 22
Compliance / Security / ISSO	13, 14, 15, 16, 20	1, 2, 11	5, 7
New to Agile	**Appendix A first**, then 1–8	—	—

Appendix A is a twenty-page Agile baseline. If you can already define a sprint, a backlog and a burndown chart, skip it — this book assumes you can and does not spend its pages re-establishing what you know. If you cannot, read it first and the rest will land.

Every chapter ends the same way: key takeaways, a short list of things you can do on Monday without budget approval, a self-assessment you can answer with evidence rather than opinion, and the mistakes most commonly made. Those sections are designed to be re-read without the chapter.

A note on the callouts. *FROM THE FIELD* is always first-person and always real — it describes something that happened in the author's programme. *ILLUSTRATIVE* means constructed to teach. *ANTI-PATTERN* names a failure mode. *STATUS AS OF* carries a dated regulatory fact and its source. The distinction is deliberate and consistent, and it matters: in a book that mixes lived experience with external research, you are entitled to know which one you are reading at all times.

Foreword: Why This Book Exists

I did not set out to write another Agile textbook. The world has enough of those. If you want a definition of Scrum or a list of the twelve Agile principles, you can find that on the first page of any search engine, for free, in thirty seconds.

I wrote this book because I spent years looking for an Agile guide that addressed the reality I faced every day — and I could not find one.

My reality looks like this. I work in enterprise software. Our products serve government agencies and the professionals who work alongside them. We operate under FedRAMP authorisation, IL5 and IL6 security requirements, CMMC compliance mandates, and NIST 800-53 controls. We have over 500 people across 25 teams, working in quarterly Program Increment cycles with three-sprint iterations, managed through Azure DevOps. Every line of code we ship passes through security scanning, compliance gates, and authorisation boundaries before it reaches a customer.

Every Agile book I picked up told me to embrace change and deliver working software frequently. Great advice. But none of them told me how to do that when every code change requires security impact analysis, when our Definition of Done includes compliance documentation, when our deployment pipeline has to pass through authorisation boundaries that take weeks to navigate. None of them told me how to run Agile ceremonies when the biggest impediment is not a technical problem — it is a regulatory one.

None of them described how to run PI Planning when half your teams are building features and the other half are remediating audit findings. None of them addressed the challenge of coordinating 25 teams across multiple products with shared infrastructure dependencies. And absolutely none of them discussed how AI is transforming the development lifecycle in ways that make traditional velocity metrics obsolete.

So I wrote the book I wish I had found.

Here is the argument the book makes.

Enterprise Agile is not a set of ceremonies, and it is not a choice of framework. It is an **operating model for continuously delivering value under uncertainty** — one that has to satisfy product outcomes, architecture, cross-team dependencies, security, compliance, governance and operations at the same time. Team-level Agile assumes most of those away. That assumption is what breaks when you scale, and no amount of running the ceremonies more diligently repairs it.

That much I could have written for the third edition. What I could not have written is the second half, and it is the reason this edition exists.

AI does not simplify that operating model. AI amplifies it. It raises the rate at which change is produced without raising the rate at which change can be safely absorbed — and the gap between those two rates is where the next few years of software delivery will be decided. Throughput goes up either way. Whether that becomes delivered value or becomes instability depends entirely on the delivery system it lands in.

Which means the organisations that benefit most from AI are the ones that already had sound delivery systems. That is close to the opposite of the story most people are currently being sold, and it is what the available evidence shows. It also happens to be an uncomfortable thing for a regulated enterprise to hear, because we are rarely the ones with the soundest delivery systems.

The discipline this book teaches is building the system that can absorb the acceleration.

What changed in this edition

This is not a revision. It is close to four times the length of the third edition and most of it is new.

The AI material has gone from one chapter to five, because a single chapter could not carry the argument above and because the evidence available now did not exist when I wrote the last one. Regulated delivery

has gone from one chapter to four — it was always the reason people read this book, and it was always the part I under-wrote. The case study has roughly tripled, and it now includes what the transformation cost and what did not improve, both of which the previous edition omitted in the way that transformation accounts usually do.

Everything factual has been re-verified against primary sources, and where a claim in the third edition turned out to have no defensible source, I removed it rather than finding a friendlier citation. Three statistics that appeared in the last edition are simply gone for that reason.

I also changed my mind about something I recommended in print. The third edition advised planning sprint capacity above measured velocity to capture expected AI productivity gains. That advice was wrong, it contradicted other advice in the same book, and it pointed in the most damaging available direction. Chapter 19 explains what I got wrong and why, at more length than is comfortable.

This is not a theoretical overview. It is a practitioner's guide built from real experience leading Agile transformation in an organisation where compliance is not optional, where teams are large and diverse, and where the pressure to deliver faster never stops. Every framework, practice and recommendation in this book has been tested in production — not in a classroom exercise.

If you work in a regulated industry — government contracting, healthcare, financial services, defence — this book was written for you. If you are a developer, tech lead or product manager trying to make Agile work in an enterprise that moves slower than a startup, this book was written for you. If you manage multiple teams and struggle with coordination, consistency and cross-team dependencies, this book was written for you.

By the end of this book you should be able to choose an operating model and defend the choice with a score rather than a preference. Integrate compliance into delivery without slowing it down, and explain to an assessor why the integrated version is better evidence than the gated one. Give a leadership team a date with a confidence level attached instead of a number you will be held to. Measure AI honestly enough that the figure survives the meeting where someone asks how you know. And lead a transformation without repeating the five mistakes I document in Chapter 23, each of which cost us something.

That last point is the one I would emphasise. Almost everything in this book has a scar behind it. Where something worked, I have tried to say why. Where it did not, I have tried to say that too — including in the places where the honest answer is that we never measured it and I cannot tell you.

Chetan Sanghani 2026

1

Agile Works Until Eight Things Collide

Team-level Agile is largely a solved problem. Everything difficult about enterprise delivery is what happens when eight forces press on that solution simultaneously.

The Problem

A single co-located team of seven, building one product, owning its own deployment, with a product owner who can decide, no regulatory obligations and no shared infrastructure, can be made to work well in about a quarter. The practices are well documented, widely taught, and not seriously contested.

Almost nobody works there.

The practices that make one team effective do not compose. Twenty-five instances of a well-run team do not produce a well-run organisation, and the reason is not that the teams are worse. It is that at scale, a set of forces that a single team can ignore become the dominant constraints — and they interact.

This is the gap that produces the most common complaint in enterprise Agile: *we did everything the books said and it did not work*. Usually that is accurate. The books were describing a context the organisation does not occupy.

> **FROM THE FIELD**
>
> Early in our Agile transformation, we made the mistake of renaming our existing waterfall phases as "sprints." We had two-week sprints, but each sprint was dedicated to a single phase: Sprint 1 was requirements, Sprint 2 was design, Sprint 3 was coding. We had the ceremonies without the mindset. It took a painful PI retrospective — where three teams independently raised the same concern — for the organisation to realise we had changed nothing except the vocabulary. The fix was simple but uncomfortable: every sprint had to deliver a working, testable increment. No exceptions.
>
> That story is usually told as a cautionary tale about cargo-cult adoption, and it is one. But there is a second reading that took me longer to arrive at. **We adopted the vocabulary because the vocabulary was the part that transferred.** The practices we read about assumed a context we did not have — teams that could deliver an increment without waiting on another team, without a security review, and without a documentation obligation. Faced with practices that did not fit, we kept the names and discarded the substance, because the substance was the part that would have required changing the organisation.

The Eight Forces

The Enterprise Collision Model — a proposed model.

Eight forces separate enterprise delivery from team-level delivery. Each is individually survivable with good practice. What makes enterprise Agile hard is that they arrive together and compound.

1 · Scale. Coordination cost grows faster than headcount. Communication paths grow roughly with the square of the number of people who need to talk to each other, and every mechanism that manages that growth — a coordination forum, a planning event, a standard — consumes capacity. Past a threshold, adding people reduces throughput.

2 · Dependencies. With one team, a dependency is a conversation. With twenty-five, dependencies form a graph, and the graph is generated by your organisation chart rather than by your architecture. Waiting becomes the largest single component of lead time, and it is invisible unless you deliberately surface it.

3 · Architecture. One team can let architecture emerge. Twenty-five teams sharing platforms, APIs and data cannot — emergent architecture across many teams produces incompatible local decisions that are individually reasonable and collectively unworkable. Yet dictating architecture centrally removes the autonomy that makes teams effective. The resolution is guardrails rather than directives, and finding the line is genuinely difficult.

4 · Security. Security at team level is a practice. At enterprise level it is a control set with owners, evidence and audit obligations, and it applies to every change regardless of how small or how urgent.

5 · Compliance. The force that most distinguishes this book. Compliance introduces requirements that did not come from a customer, cannot be de-scoped, have external deadlines, and are assessed by people outside your organisation. It is the only force on this list where getting it wrong can remove your right to operate.

6 · Governance. Decision rights, funding cycles, portfolio prioritisation and escalation paths. Governance is what tells a team whether it may make a decision. Most enterprise Agile failures that are described as cultural are governance failures: teams told to self-organise while decision rights remained where they always were.

7 · Operations. Production exists and it interrupts. Support work, incidents, patching and toil arrive on their own schedule and do not respect sprint boundaries. Organisations that plan as though operational load were zero discover it as chronic overcommitment.

8 · AI. The newest force and the one that changes the others. AI increases the rate at which change is produced without increasing the rate at which change can be safely absorbed. It does not add a new category of problem — it raises the pressure on all seven of the others simultaneously.

Why It Is the Interactions That Break Teams

Any single force can be managed with a known practice. The difficulty is that they multiply.

Scale × Dependencies. Coordination mechanisms are the standard answer to dependency load. But every mechanism is itself coordination overhead, so scale makes the remedy for dependencies more expensive at exactly the rate it makes dependencies worse.

Compliance × Governance. Compliance produces requirements. Governance decides what gets funded. When these are not connected, compliance work is invisible to prioritisation until it becomes an emergency — which is the normal state in most regulated organisations.

Architecture × Dependencies. Your dependency graph is a function of your architecture and your org chart. Most dependency-management effort treats the symptom. The remedy is usually architectural or organisational, and both are slower and more political than adding a coordination forum.

Operations × Scale. Operational load is roughly proportional to the amount of software in production, which grows continuously. Development capacity does not grow to match. Every mature organisation therefore experiences a slow erosion of feature capacity that nobody planned for.

AI × everything. More change means more review, more integration surface, more compliance evidence, more architectural drift, more operational surface. This is Chapter 17's argument and the reason AI appears in Part V rather than as an appendix: it is a load test of the other seven.

The practical consequence: a fix aimed at one force in isolation frequently makes another worse. Add a coordination forum to manage dependencies and you have consumed capacity that operational load was already claiming. Tighten architectural governance and you slow the teams whose autonomy you were

relying on. **Every enterprise Agile decision is a trade among these eight, and the mistake is treating any one of them as a standalone problem.**

When Agile Is Not the Answer

A short section, included because its absence in most Agile books is a credibility problem.

Iterative delivery pays off when requirements are uncertain, feedback is available, and the cost of change is low relative to the cost of being wrong. Where those conditions do not hold, the ceremony is overhead.

SITUATION	WHY ITERATION HELPS LESS
A one-shot data migration with a fixed cutover	There is no increment. Feedback arrives once, at the end, by definition
Hardware-coupled delivery	The feedback loop is bounded by manufacturing, not software
A fixed-date regulatory submission with defined content	Scope is externally specified and non-negotiable; the risk is schedule, not learning
Genuine commodity work with no uncertainty	Nothing to learn; flow-based management fits better than iteration

Even here, the engineering practices usually still apply — version control discipline, automated testing, continuous integration, small batches. **It is the iteration and the ceremony that stop earning their cost, not the discipline.** Being able to say this out loud makes the rest of the book more credible, not less.

What This Book Argues

The rest of this book is one claim, developed:

> Enterprise Agile is not a set of ceremonies or a choice of framework. It is an **operating model for continuously delivering value under uncertainty** — one that must simultaneously satisfy product outcomes, architecture, cross-team dependencies, security, compliance, governance and operations. AI does not simplify that model. **AI amplifies it: it raises throughput and, unless the operating model is sound, raises instability in the same motion.** The discipline is building the system that can absorb the acceleration.

Chapter 2 makes the operating model explicit. Everything after it is a treatment of one or more of the eight forces, and the compliance and AI forces — the two that distinguish this book — get a part each.

> **ANTI-PATTERN — ADOPTING PRACTICES WITHOUT THEIR PRECONDITIONS**
>
> **What it looks like:** an organisation adopts a practice that works elsewhere — two-week sprints, self-organising teams, continuous deployment — without the conditions that made it work. Sprints without the ability to deliver an increment. Self-organisation without decision rights. Continuous deployment without automated testing.
>
> **Why it fails:** the practice was the visible part of a system. Transplanted without the system, it produces the ceremony and none of the benefit — which is precisely how "we did Agile and it did not work" happens.
>
> **The tell:** the vocabulary changed and the outcomes did not. See the callout above; this is our own story.
>
> **What to do instead:** for each practice, ask what has to be true for it to work. If those things are not true, build them first or choose a different practice. A practice adopted without its preconditions is worse than none, because it consumes capacity and discredits the change.

Metrics

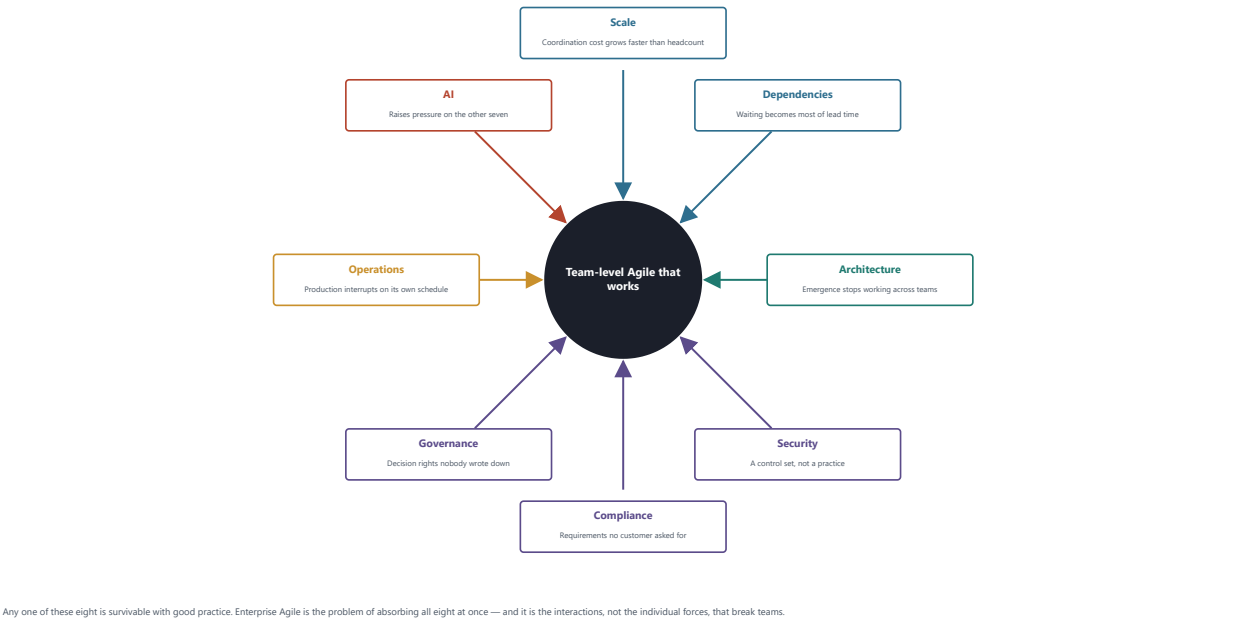
At this altitude, four numbers tell you which forces are actually binding.

METRIC	DIAGNOSES
End-to-end lead time, decomposed by stage	Where you are actually constrained. The single most valuable measurement in this book
Wait time as a proportion of lead time	Dependency and gate load. Above 50% is common and always worth attacking
Unplanned work as a proportion of capacity	Operational force. Above 30% means your plan is fiction
Cross-team dependencies per increment	Whether the graph is growing faster than your ability to manage it

If you measure nothing else from this book, measure the first two.

FIGURE 1.1

The Enterprise Collision Model



Any one of these eight is survivable with good practice. Enterprise Agile is the problem of absorbing all eight at once — and it is the interactions, not the individual forces, that break teams.

Figure 1.1 The Enterprise Collision Model

Key Takeaways

- **Team-level Agile is largely solved. Enterprise Agile is a different problem** — and it is not the first problem repeated twenty-five times.
- **Eight forces separate the two:** scale, dependencies, architecture, security, compliance, governance, operations and AI.
- **The interactions do the damage, not the individual forces.** A fix aimed at one commonly makes another worse.
- **Practices do not transplant without their preconditions.** The vocabulary transfers easily and the substance does not, which is exactly how cargo-cult adoption happens.
- **AI is the newest force and it raises pressure on the other seven** rather than adding a new category of problem.
- **Sometimes iteration is not the answer.** The engineering discipline usually still is.
- **Enterprise Agile is an operating model, not a methodology.** Chapter 2 makes it explicit.

Apply This Monday

1. **Score the eight forces for your organisation** — one to five for how much each currently constrains you. The top two are what the rest of this book should help you with; read those chapters first.
2. **Decompose one feature's lead time by stage** and calculate wait time as a proportion. Most people are surprised by the result.
3. **Measure unplanned work as a proportion of capacity** for the last three sprints. If it exceeds 30%, your planning is describing a system you do not have.

4. **Pick one practice your organisation adopted that did not stick.** Ask what precondition was missing. It is usually identifiable in ten minutes and it usually explains several other failures too.

Self-Assessment

1. Which two of the eight forces constrain you most? Would your engineers agree?
2. What proportion of your lead time is wait time?
3. Can a team deliver a working increment without depending on another team? For how many of your teams is that true?
4. Where are decision rights actually held — as observed, not as documented?
5. What proportion of your capacity goes to unplanned operational work?
6. Which practices have you adopted whose preconditions you never built?
7. Is there work in your organisation where iteration genuinely is not the right model? Are you forcing it anyway?

Common Mistakes

1. **Assuming enterprise Agile is team Agile at volume.** It is a different problem with different constraints.
2. **Fixing one force in isolation.** The coordination forum that solves your dependency problem consumes the capacity your operational load needed.
3. **Adopting the vocabulary because the substance would require organisational change.** Ours, and almost everyone's.
4. **Treating governance failures as cultural failures.** Teams told to self-organise while decision rights stay put are not resisting; they are correctly reading the situation.
5. **Planning as though operational load were zero.**

Chapter 2 makes the operating model explicit — eleven layers, what each decides, and the feedback loops that make it a system rather than a hierarchy.

2

The Enterprise Agile Operating Model

Eleven layers, what each one decides, and the three feedback loops that make it a system rather than an organisation chart. This is the model the rest of the book hangs from.

The Problem

Every enterprise has an operating model. Most have never written it down, which means nobody can point to the part that is broken.

The symptoms are recognisable. Work arrives at teams without anyone being able to explain how it was prioritised. A team makes a decision and it is overturned by someone whose right to overturn it is unclear to both parties. Compliance requirements appear late because no layer was responsible for surfacing them early. An executive asks why something took nine months and receives four different answers, all true. Escalation happens by finding whoever will listen.

None of these is a team-level problem, which is why team-level remedies do not touch them. They are properties of the system the teams sit inside — and an undocumented system cannot be debugged.

What the Frameworks Say

Scaling frameworks provide an answer: a defined structure with named roles, cadences and artifacts. That is genuinely useful, and it is also where most of the criticism lands, usually fairly. Adopting a framework's full configuration imports decisions that were made for someone else's context, and the roles tend to arrive before the problems they solve.

The deeper issue is that a framework diagram shows **structure** — who exists and how they are arranged. What an organisation actually needs to reason about is **decision rights and feedback** — who decides what, and how evidence returns to where decisions are made. Two organisations can implement the same framework structure and behave completely differently depending on where decisions actually sit.

The model below is therefore not a structure. It is a decision-rights and feedback model, and it is deliberately framework-neutral: it describes what must happen, not what the boxes are called. It works whether you run SAFe, LeSS, a hybrid, or nothing at all.

The Model

The Enterprise Agile Operating Model — a proposed model.

Eleven layers. Intent flows down; evidence flows up. Security and compliance are not layers between delivery and production — **they are vertical bands crossing every delivery layer**, which is the

visual argument of Part IV planted at the start of the book.

LAYER	DECIDES ALONE	CADENCE	PRIMARY METRIC	FAILURE MODE IF ABSENT
1 · **Strategy**	What business outcomes matter and what we will not pursue	Annual, revisited quarterly	Business outcome attainment	Teams optimise locally with no shared direction
2 · **Portfolio**	What is funded, and what is stopped	Quarterly	Value delivered per investment; work-in-progress at portfolio level	Everything is funded, nothing is finished
3 · **Product**	What we build, in what order, for whom	Continuous, formalised per increment	Outcome metrics (Ch 5)	Output with no theory of value
4 · **Program / ART**	How teams coordinate; cross-team sequencing and dependency resolution	Per increment	Dependency resolution time; predictability	Teams collide at integration
5 · **Teams**	How the work gets done; sprint scope against the goal	Per sprint	Flow metrics; sprint goal success	Work assigned rather than pulled; no ownership
6 · **Engineering**	Technical approach, architecture within guardrails, quality practice	Continuous	Change failure rate; test effectiveness	Architectural drift; quality as an afterthought
7 · **Security**	Control implementation approach; risk identification	Continuous, crossing all layers	Finding discovery stage; time to remediate	Security discovered at the gate
8 · **Compliance**	Control applicability, evidence sufficiency, assessment readiness	Continuous, crossing all layers	Evidence automation rate; POA&M ageing	Compliance becomes a terminal gate
9 · **DevOps / Platform**	Pipeline, deployment mechanism, environment provisioning	Continuous	Deployment frequency; lead time for change	Every team solves delivery separately
10 · **Production**	Operational response; what constitutes an incident	Continuous	Availability; MTTR; change failure rate	Operational load invisible to planning
11 · **Customer outcomes**	Nothing — this layer only *reports*	Continuous	Adoption; task success; business impact	The organisation cannot tell whether any of the above worked

Layer 11 decides nothing, and that is the point. It is the only layer that tells you whether the other ten are working. Organisations that lack it are running an open loop: they can tell you what they shipped and not whether it mattered.

The Three Feedback Loops

The layers are the easy part. **Most enterprise Agile failures are a missing return arrow, not a missing ceremony**, and the three loops run at deliberately different speeds.

The fast loop — Production → Teams, continuous to daily. Errors, performance, usage, incidents returning to the people who wrote the code. Where this loop is missing, teams learn about their own

defects from a support ticket weeks later, and the feedback arrives too late to change behaviour. This is the cheapest loop to build and the one whose absence is most commonly tolerated.

The medium loop — Customer outcomes → Product, per sprint to per increment. Whether shipped work produced the intended effect. Where this is missing, the roadmap becomes a list of outputs with no evidence of value, and prioritisation degrades into whoever asked most recently. This is the loop most organisations claim to have and least often actually operate — see Chapter 5.

The slow loop — Outcomes → Strategy, per increment to annual. Whether the strategy itself is correct. This loop is almost always missing. Strategy is set, cascaded, and then not revisited against evidence until the annual cycle regardless of what was learned. An organisation without this loop can execute a wrong strategy with great efficiency for a year.

> **The diagnostic:** for each loop, name the artifact that carries the evidence and the meeting where it changes a decision. If you cannot name both, the loop does not exist — you have a report, not a feedback loop.

Decision Rights

The most common enterprise Agile dysfunction is not the absence of decision rights. It is their ambiguity — and specifically, **layers that hold vetoes without holding accountability.**

A veto is not a decision right. A function that can stop work but is not accountable for delivery will optimise for its own risk, correctly and rationally, because that is the only thing it is measured on. Every gate in your organisation that can say no without owning the consequence of delay is a structural source of delay, and no amount of goodwill fixes it.

DECISION	DECIDED BY	CONSULTED	VETOES
Funding a product line	Portfolio	Product, Strategy	—
What is in this increment	Product	Program, Teams	Compliance (applicability only)
How a story is implemented	Team	Engineering	Security (control-relevant only)
Architectural standard	Engineering	Teams, Security	—
Whether a control applies	Compliance	Security, Product	—
Whether evidence is sufficient	Compliance	—	—
Release to production	Product + Program	Security, Compliance, Ops	Compliance (for authorisation-relevant releases)
Accepting a security risk	Named accountable executive	Security, Compliance	—

Two rules make this workable.

Every veto is scoped. Compliance can veto on control applicability and evidence sufficiency. It cannot veto on design preference. Security can veto a control-relevant implementation. It cannot veto an architectural choice with no security consequence. **Unscoped vetoes expand**, reliably and without anyone deciding they should.

Risk acceptance is a named person, never a committee. If nobody's name is on the acceptance, the risk has not been accepted — it has been deferred, and it will resurface at the least convenient moment.

Escalation

Escalation paths in most organisations are informal, which means they work through relationships and fail for people without them.

A functioning path has three properties. **It is defined per decision type** — a technical disagreement, a priority conflict and a compliance dispute escalate to different places. **It has a time limit** — a dependency unresolved after N days escalates automatically rather than waiting for someone to be sufficiently annoyed. And **escalation is normal**, not a failure signal. Where escalating is treated as an admission that you could not handle it, people stop, and problems sit.

CONFLICT	ESCALATES TO	TRIGGER
Cross-team dependency unresolved	Program / RTE	2 days
Priority conflict between products	Portfolio	Immediately; do not let teams arbitrate
Technical disagreement within guardrails	Engineering	1 day of blocked work
Control interpretation dispute	Compliance, then the assessor	Immediately
Risk requiring acceptance	Named executive	Immediately

The compliance row matters. Teams argue interpretation among themselves for weeks. The assessor is the authority on interpretation and asking early is cheap (Ch 14).

One Decision, Traced Through All Eleven Layers

Models are easier to argue with than to use. Here is a single ordinary decision moving through the whole stack — *should this feature ship in the next increment?* — with what each layer contributes and where it typically breaks.

[ILLUSTRATIVE — constructed to show the model working. The failure modes are real; the feature is not.]

Suppose a large customer has asked for bulk export, it touches data handling, and someone wants it in the next increment.

LAYER	WHAT IT CONTRIBUTES	WHERE IT BREAKS
1 Strategy	Is expanding this customer segment something we are pursuing? If not, the request may be a distraction with a loud advocate	Strategy is unwritten, so nobody can answer, and the loudest advocate wins by default
2 Portfolio	Is this area funded to take on more? If yes, what stops to make room?	Nothing ever stops, so the answer is "add it", and every commitment gets quietly weaker
3 Product	What outcome is this meant to move, and how would we know? Which rung of the ladder are we targeting? (Ch 5)	The request is accepted as a feature rather than as a hypothesis, and nobody instruments it
4 Program / ART	Who else does this touch? Bulk export needs the platform team and the data model	Dependencies surface during the increment rather than before it
5 Teams	Can we deliver a usable slice in the increment, and what does it displace?	The team is told rather than asked, and the displacement is invisible until something else slips
6 Engineering	Does this fit the architecture, or does it need a new pattern? Does it create debt?	Decided implicitly by whoever writes it first, and the pattern propagates
7 Security	Bulk export of customer data — what is the risk, and what controls apply?	Consulted after the design is fixed, so its input arrives as rework
8 Compliance	Does this change the authorisation boundary? Is it a significant change? Does it need reassessment?	**Asked last, or not at all.** This is the layer whose late involvement causes the most expensive surprises
9 DevOps / Platform	Can it be deployed with existing mechanisms, or does it need new infrastructure?	The team builds something the platform cannot deploy without manual intervention
10 Production	What is the operational load — support volume, monitoring, failure modes?	Nobody asks, and the on-call rota absorbs it silently (Ch 7)
11 Customer outcomes	Did they use it? Did it change anything?	Never revisited. The feature ships and the organisation moves on

Three things this makes visible that a process diagram does not.

The decision is made at layer 3, but it cannot be made well without inputs from 4, 7, 8 and 10. A Product Owner deciding alone is not empowered — they are under-informed, and the difference matters because the remedy is different. Empowerment problems are cultural; information problems are structural, and structural ones are fixable in a week.

Layer 8 is where the cost of lateness compounds fastest. If compliance is asked at the point of the decision, the answer is a constraint and it shapes the design. Asked after the code is written, the same answer is a rework order. Same information, radically different cost, and the only variable is when it was requested. This is the entire argument of Chapter 13 arriving through the operating model rather than through the compliance function.

Layer 11 has no arrow back into layer 3 in most organisations. Everything above happens, the feature ships, and the loop is never closed. Which means the next decision of exactly this kind is made with the same information as this one — no better. **An organisation that runs this sequence a hundred times and never closes the loop has not learned anything from ninety-nine of them.**

> **TRY THIS**
>
> Take a decision your organisation made in the last month — a real one, ideally one that went badly — and walk it down the eleven layers. For each, ask two questions: *who contributed here, and when?* In my experience you will find two or three layers that contributed nothing, and at least one that contributed after the decision was effectively made. Those are not communication failures. They are the operating model, working exactly as it is currently built.

Applying the Model

Three uses, in increasing order of value.

As a diagnostic. Walk the eleven layers and ask, for each: who decides here, what is the cadence, and what evidence do they have? Most organisations find two or three layers with no clear owner, and those are usually where the observable symptoms are coming from.

As a mini-map. Each part of this book operates at particular layers. Part II is layers 3–6; Part III is 2–6 and 9–10; Part IV is 7–8, crossing everything; Part V crosses all of them.

As an argument. When someone proposes a change, ask which layer it addresses and which feedback loop it strengthens. Proposals that do neither are usually adding coordination overhead to compensate for a missing return arrow — which is the most common and most expensive category of enterprise Agile intervention.

> **ANTI-PATTERN — THE STRUCTURE WITHOUT THE LOOPS**
>
> **What it looks like:** an organisation adopts a scaling framework faithfully. The roles exist, the cadences run, the artifacts are produced. Delivery does not improve.
>
> **Why it fails:** the framework supplied the structure and the organisation did not build the feedback. Intent flows down through a well-defined hierarchy and nothing meaningful flows back. Planning events are attended and no decision changes because of evidence from the last increment.
>
> **The tell:** ask what decision changed as a result of the last increment's outcome data. If the answer is a scope adjustment rather than a priority or strategy change, the medium and slow loops are not running.
>
> **What to do instead:** build the fast loop first — it is cheapest and its absence is most damaging. Then the medium loop, which requires outcome instrumentation and is the real work (Ch 5). The slow loop is a leadership behaviour more than a mechanism, and it will not appear until the first two produce evidence worth acting on.

Metrics

The model's own health, as distinct from delivery metrics:

METRIC	WHAT IT TELLS YOU
Layers with a named decision owner	Structural completeness. Below 11 is a gap list
Loop latency — evidence generated to decision changed	The real speed of your organisation. Usually far slower than the ceremony cadence suggests
Escalations resolved within trigger time	Whether escalation works or is theatre
Unscoped vetoes — functions that can block without defined scope	Structural delay sources
Decisions overturned above the deciding layer	Whether stated decision rights are real

Loop latency is the one to track. An organisation with a two-week sprint cadence and a nine-month loop latency is not operating at two-week speed, whatever its calendar says.

FIGURE 2.1

The Enterprise Agile Operating Model

	DECIDES ALONE	CADENCE	PRIMARY METRIC	FAILURE MODE IF ABSENT	
1 · Strategy	What outcomes matter; what we will not pursue	Annual, revisited quarterly	Outcome attainment	Teams optimise locally with no shared direction	FAST LOOP — production to teams, daily
2 · Portfolio	What is funded, and what is stopped	Quarterly	Value per investment; portfolio WIP	Everything funded, nothing finished	
3 · Product	What we build, in what order, for whom	Continuous	Outcome metrics	Output with no theory of value	
4 · Program / ART	Cross-team sequencing and dependency resolution	Per increment	Dependency resolution time	Teams collide at integration	
5 · Teams	How the work gets done; sprint scope	Per sprint	Flow; sprint goal success	Work assigned rather than pulled	MEDIUM LOOP — outcomes to product, per sprint
6 · Engineering	Technical approach within guardrails	Continuous	Change failure rate	Architectural drift	
7 · Security	Control approach; risk identification	Continuous — crosses all layers	Finding discovery stage	Security discovered at the gate	
8 · Compliance	Control applicability; evidence sufficiency	Continuous — crosses all layers	Evidence automation rate	Compliance becomes a terminal gate	
9 · DevOps / Platform	Pipeline, deployment, environments	Continuous	Deployment frequency	Every team solves delivery separately	SLOW LOOP — outcomes to strategy, per increment
10 · Production	Operational response; what is an incident	Continuous	Availability; MTTR	Operational load invisible to planning	
11 · Customer outcomes	Nothing — this layer only reports	Continuous	Adoption; task success; impact	You cannot tell whether any of it worked	

Intent flows down; evidence flows up. The three feedback loops run at different speeds. Most enterprise Agile failures are a missing return arrow, not a missing ceremony — and layer 11, which decides nothing, is the only one that tells you whether the other ten worked.

Figure 2.1 The Enterprise Agile Operating Model

FIGURE 2.2

Decision rights across the operating model

	DECIDED BY	CONSULTED	SCOPED VETO
Funding a product line	Portfolio	Product, Strategy	—
What is in this increment	Product	Program, Teams	Compliance — applicability only
How a story is implemented	Team	Engineering	Security — control-relevant only
Architectural standard	Engineering	Teams, Security	—
Whether a control applies	Compliance	Security, Product	—
Whether evidence is sufficient	Compliance	—	—
Release to production	Product + Program	Security, Compliance, Ops	Compliance — authorisation-relevant releases
Accepting a security risk	A NAMED executive	Security, Compliance	—

A veto is not a decision right. Every veto here is scoped, because unscoped vetoes expand — reliably, and without anyone deciding they should. Risk acceptance is a named person, never a committee: a committee acceptance is a deferral with a signature block.

Figure 2.2 Decision rights across the operating model

Key Takeaways

- **Every enterprise has an operating model; most have not written it down**, which is why nobody can point at the broken part.
- **Eleven layers, but the layers are the easy part.** Most failures are a missing return arrow, not a missing ceremony.
- **Three loops at three speeds** — production to teams, outcomes to product, outcomes to strategy. The third is almost always missing.
- **Layer 11 decides nothing and is the only one that tells you whether the rest worked.** Without it you are running an open loop.
- **Security and compliance cross every delivery layer.** Drawing them as gates between layers is where regulated delivery goes wrong.
- **A veto is not a decision right.** Any function that can block without owning the delay is a structural source of delay.

- **Unscoped vetoes expand.** Scope every one explicitly.
- **Risk acceptance is a named person.** A committee acceptance is a deferral.

Apply This Monday

1. **Walk the eleven layers and name the decision owner for each.** Unnamed layers are your gap list, and there will be two or three.
2. **For each of the three loops, name the artifact and the meeting where evidence changes a decision.** If you cannot name both, the loop is a report.
3. **List every function that can block a release.** For each, write the scope of its veto. Anything unscoped is a delay source.
4. **Measure loop latency once:** pick a piece of outcome evidence from last quarter and trace whether any decision changed because of it, and how long that took.

Self-Assessment

1. Can you name the decision owner for all eleven layers?
2. What decision changed as a result of last increment's outcome data?
3. Which functions can veto, and is each veto scoped in writing?
4. When a team's decision is overturned, does everyone agree that the overturning party had the right?
5. Does your escalation path work for someone without strong internal relationships?
6. Is anyone accountable for customer outcomes as distinct from delivery?
7. Are security and compliance participants in every layer, or gates between some of them?
8. What is your loop latency, and how does it compare to your sprint length?

Common Mistakes

1. **Adopting structure without feedback.** The most common and most expensive framework failure.
2. **Treating layer 11 as optional** because it does not decide anything.
3. **Leaving vetoes unscoped.** They expand, and nobody decides that they should.
4. **Committee risk acceptance.** A deferral with a signature block.
5. **Informal escalation.** It works through relationships and fails everyone without them.
6. **Confusing ceremony cadence with organisational speed.** Two-week sprints and nine-month loop latency is a two-week theatre over a nine-month system.

Chapter 3 addresses the question this model deliberately left open: how the teams inside it should actually work.

3

Choosing How You Will Work

Framework selection is a fit problem, not a ranking problem. This chapter gives you an instrument instead of an opinion.

The Problem

"Which framework should we use?" is asked constantly and answered badly, in one of three ways.

By popularity. Scrum is the most widely adopted, so Scrum. This tells you what other organisations chose, in contexts you know nothing about.

By size. Small team, Scrum; many teams, SAFe. This is the answer most decision trees produce, including the one in the previous edition of this book, and it is close to useless — it reduces a ten-dimensional decision to one dimension, and not the most important one.

By whoever attended the training. More common than anyone admits, and it explains a remarkable proportion of enterprise framework adoption.

None of these examines the context that determines whether a framework will work. And there is a fourth failure that is worse than all three: **treating this as a decision made once.** The right answer changes as an organisation matures, and most organisations never revisit it.

What the Frameworks Say About Themselves

Every framework describes the conditions under which it works. Almost none describes the conditions under which it does not, because that is not what framework documentation is for.

This produces a genuine information gap. A reader can find out what Scrum requires and what SAFe provides. Finding out when SAFe is the wrong choice requires either experience or a book willing to say it. So, from the top:

Scrum works when work is decomposable into two-week increments, the team can deliver something usable in that window, and a Product Owner can decide. It fails where work arrives unpredictably (Ch 7), where increments cannot be completed because of external dependencies, or where nobody can actually prioritise.

Kanban works when work arrives continuously, priorities shift faster than an iteration, or the work is genuinely heterogeneous. It fails where an organisation needs a predictable planning rhythm to coordinate with others, and where a team uses "no iterations" to avoid commitment altogether.

SAFe works when many teams share a product or platform and need synchronised planning, and when the organisation genuinely will fund and staff the coordination roles. It fails when adopted wholesale

without the problems its roles solve, when it is used as a governance layer over teams that are not actually Agile, and when the cadence is imposed on work that does not fit it.

LeSS works when a genuinely unified product is built by teams with high engineering maturity and strong organisational trust. It fails where teams are geographically or organisationally separated, or where the organisation is not prepared to have one backlog and one Product Owner — which is a much harder constraint than it sounds.

XP practices work everywhere and are not a scaling framework. Pair them with whichever of the above fits.

Hybrid — different frameworks for different work within one organisation — is what most mature organisations actually run, and it is almost never presented as a legitimate terminal choice. It is.

The Framework Fit Score

A proposed model. Ten dimensions. Score your context 1–5 on each, then read the profile.

#	DIMENSION	1	5
1	**Teams sharing one product**	One team	Fifteen or more
2	**Work predictability**	Arrives unpredictably, priorities shift daily	Plannable weeks ahead
3	**Dependency density**	Team can deliver alone	Most work needs two or more teams
4	**Regulatory burden**	None	Authorisation at risk; external assessment
5	**Product complexity**	Single coherent product	Many products, shared platform
6	**Release frequency needed**	Quarterly acceptable	Multiple times daily
7	**Operational load**	Under 10% of capacity	Over 40%
8	**Engineering maturity**	Manual testing, infrequent integration	Full automation, trunk-based, continuous deployment
9	**Organisational maturity**	Command-and-control, low trust	High autonomy, decisions at the edge
10	**Co-location / timezone overlap**	Fully distributed, minimal overlap	Co-located or heavily overlapping

Reading the score — dimension weights, not a total. The point of the instrument is that different dimensions dominate different decisions, so a single number would hide the reasoning.

SIGNAL	INDICATES
D1 ≤ 2 and D3 ≤ 2	You do not have a scaling problem. **Do not adopt a scaling framework.** Scrum or Kanban at team level
D2 ≤ 2 or D7 ≥ 4	Flow-based management for that work. Kanban, regardless of what the rest of the organisation runs
D1 ≥ 4 and D3 ≥ 4	You need synchronised planning. SAFe or LeSS — decide with D9 and D10
D1 ≥ 4, D5 = 1, D8 ≥ 4, D9 ≥ 4, D10 ≥ 4	LeSS is viable. All five must hold; LeSS is unforgiving of the exceptions
D1 ≥ 4 and (D9 ≤ 3 or D10 ≤ 3 or D5 ≥ 3)	SAFe. It is more tolerant of distribution, multiple products and lower organisational maturity — which is the honest reason most large organisations land there
D4 ≥ 4	**Compliance overlays apply regardless of framework.** Part IV. This is not a framework choice
D8 ≤ 2	**Fix this before choosing anything.** Low engineering maturity makes every framework perform badly, and a framework will be blamed for it
Mixed D2/D7 across teams	**Hybrid is correct.** Do not force one model on heterogeneous work

Two of these deserve emphasis because they are the most commonly ignored.

D8 is a precondition, not a dimension. An organisation without automated testing and reliable integration will fail with any framework, and will conclude the framework was wrong. Engineering maturity is not something a framework provides.

D4 is not a framework input at all. Regulatory burden changes what you must do within whatever framework you pick. The third edition of this book drew compliance as a dashed footnote on the decision tree, which was exactly the wrong treatment — it is not a branch, it is a layer over the whole diagram.

Four Scenarios, Scored

[ILLUSTRATIVE — constructed to show the instrument working, not descriptions of real organisations.]

The instrument is worth more when you see it resolve a case that looks obvious and turns out not to be.

Scenario 1 · The startup that thinks it has a scaling problem

Five teams, one product, growing fast. Leadership has been told they need SAFe.

> D1 **3** · D2 **4** · D3 **2** · D4 **1** · D5 **1** · D6 **4** · D7 **2** · D8 **4** · D9 **4** · D10 **4**

Read: D3 is 2. Teams can mostly deliver alone. **They do not have a scaling problem — they have a growth anxiety.** Adopting a scaling framework here buys coordination machinery for dependencies that do not exist, and the overhead will be blamed on Agile within two quarters.

Answer: stay at team level. Scrum, with XP practices given D8 is already strong. Revisit when D3 reaches 4 — and the trigger to watch is integration defects, not headcount.

Scenario 2 · The regulated enterprise at twenty-five teams

Multiple products, shared platform, FedRAMP authorisation, distributed teams, moderate organisational maturity.

> D1 **5** · D2 **4** · D3 **5** · D4 **5** · D5 **4** · D6 **2** · D7 **3** · D8 **3** · D9 **2** · D10 **2**

Read: D1 and D3 both at the ceiling — synchronised planning is genuinely required. D5 at 4 rules out LeSS immediately, and D9 and D10 at 2 rule it out twice more. **D4 at 5 is not a framework input at all** — it is Part IV, and it applies whatever you pick.

Answer: SAFe Essential, adopted selectively (Chapter 8), with compliance overlays. Note D8 at 3: the engineering maturity is adequate but not strong, and that will limit what the framework can deliver. Fixing it is the higher-return investment, and it will not feel like a framework decision.

Scenario 3 · The platform team drowning in interrupts

Eight engineers, one shared platform, everything is urgent, roadmap constantly displaced.

> D1 **2** · D2 **1** · D3 **3** · D4 **3** · D5 **2** · D6 **5** · D7 **5** · D8 **4** · D9 **3** · D10 **3**

Read: D2 at 1 and D7 at 5. This is the clearest signal the instrument produces, and it is routinely misread as a performance problem. **Sprint commitments here will fail for structural reasons no estimation coaching can fix.**

Answer: Kanban with WIP limits and classes of service (Chapter 7). Report cycle time and throughput, not sprint goal success. Expect the demoralisation to lift immediately and cycle time to improve as a side effect of the WIP limit.

Scenario 4 · The distributed organisation with almost no overlap

Twelve teams across three continents, two products, four hours of shared working time on a good day.

> D1 **4** · D2 **3** · D3 **4** · D4 **2** · D5 **3** · D6 **3** · D7 **2** · D8 **3** · D9 **3** · D10 **1**

Read: D1 and D3 say synchronised planning is needed. D10 at 1 says the synchronous events that normally deliver it are close to impossible. This is the genuinely hard case, and no framework solves it off the shelf.

Answer: SAFe cadence with asynchronous adaptation (Chapter 6) — objectives and known dependencies circulated as pre-reads, the scarce overlap window spent exclusively on negotiation and disagreement, and planning events split across two shorter days rather than one long one. **The real intervention is architectural:** at D10 of 1, every dependency costs a day of latency minimum, so reducing the dependency count (Chapter 10) buys more than any coordination mechanism will.

What the four have in common. In three of them the presenting question — "which framework?" — was the wrong question. Scenario 1 needed no framework, scenario 3 needed a different model of work entirely, and scenario 4 needed architecture. **Only scenario 2 was actually a framework decision**, and even there the instrument flagged engineering maturity as the higher-return investment.

That ratio matches my experience. Most framework questions are something else wearing a framework question's clothes, and the value of scoring the ten dimensions is that it tells you which.

The Hybrid Case

> **FROM THE FIELD**
>
> We use Kanban for our operations and support teams, where work arrives continuously and unpredictably.
>
> Trying to force support tickets into two-week sprints was a disaster — we would plan a sprint, and then
>
> a critical production issue would blow up the plan on day two. The team felt constant failure despite
>
> doing important work. Switching to Kanban with WIP limits of 3 per developer brought sanity and reduced
>
> our average ticket resolution time from 5 days to 3 days — a 40% improvement.
>
> The phrase I would draw out is *"constant failure despite doing important work."* That team was not
>
> underperforming. It was being measured against a model that did not describe its work, and the
>
> resulting demoralisation was a measurement artifact. We had imposed sprints for consistency, which is
>
> the most common reason organisations force a single framework, and consistency of *model* is worth very
>
> little when the work genuinely differs.

Scored on the instrument, that support team looked nothing like our feature teams: D2 of 1 against 4, D7 of 5 against 2. Different scores should produce different answers.

Our actual configuration — and this is what "hybrid" means concretely, rather than as a way of avoiding a decision:

WORK TYPE	MODEL	WHY
Feature development	Scrum, two-week sprints	Plannable work, decomposable increments
Operations and support	Kanban with WIP limits	Unpredictable arrival, interrupt-driven
Cross-team coordination	SAFe PI Planning and ARTs	25 teams, high dependency density
Engineering practice	XP-derived, everywhere	Orthogonal to all of the above

Hybrid needs one thing to work: agreement on the interfaces. Teams may run different internal models provided they meet the same cadence for cross-team commitments, use the same Definition of Done, and report the same flow metrics. Chapter 8 covers standardising interfaces while leaving interiors alone — it is the discipline that makes hybrid coherent rather than chaotic.

Migration and Exit

Framework choices should change. Almost no organisation revisits them.

FROM → TO	TRIGGER	COST
Scrum → Kanban	Sprint commitments missed repeatedly due to interrupts; D2 or D7 has moved	Low. Mostly a measurement and expectation change
Kanban → Scrum	Need to coordinate with a cadence-based organisation; work has become plannable	Low–medium. The commitment skill has to be built
Team-level → SAFe	D1 and D3 both rose; integration failures are now the dominant defect source	**High.** Roles, training, event overhead, and a real productivity dip
SAFe → lighter	Coordination overhead exceeds coordination benefit; dependencies have been structurally reduced	Medium. Politically hard — roles exist and people hold them
Anything → hybrid	Work types have genuinely diverged	Low, if interfaces are already standardised

Exit criteria are worth writing at adoption time, because they are unwriteable later. The question "what would tell us this framework is no longer right?" is answerable on day one and becomes politically impossible once roles are filled. Suggested triggers: coordination overhead exceeding a stated share of capacity; planning events where no dependency is discovered (the coordination is no longer needed); or a sustained fall in dependency density.

> **ANTI-PATTERN — THE FRAMEWORK AS GOVERNANCE LAYER**
>
> **What it looks like:** an organisation adopts a scaling framework primarily to give leadership visibility and predictability. Teams are not meaningfully Agile; the framework supplies planning events, roll-up reporting and a cadence.
>
> **Why it fails:** scaling frameworks coordinate Agile teams. Applied to teams that cannot deliver an increment, they produce an expensive reporting apparatus over unchanged delivery — and they discredit
>
> the framework, which is then blamed for outcomes it never had the ability to affect.
>
> **The tell:** teams' sprint outcomes are reported upward in detail, and nothing about how teams work has changed.
>
> **What to do instead:** if D8 or D9 is low, fix those first. A scaling framework is an amplifier of team-level capability, and amplifying zero yields zero — at considerable cost.

Metrics

METRIC	WHAT IT TELLS YOU	ACTION THRESHOLD
Coordination overhead — capacity in cross-team coordination	Whether the framework is paying for itself	Above 15% warrants review
Dependencies discovered per planning event	Whether coordination is still needed	Approaching zero means you may be over-coordinating
Sprint goal success by team	Whether the model fits that team's work	Below 60% sustained suggests wrong model, not weak team
Interrupt rate by team	Which teams should be on flow-based management	Above 30% means Kanban
Framework consistency vs. outcome variance	Whether standardisation is helping	Consistent framework, wildly varying outcomes means the framework is not the variable

Sprint goal success below 60% is worth reading carefully. The instinct is to conclude the team is underperforming. Check the interrupt rate first — it is frequently a fit problem, and the support-team story above is exactly that failure.

FIGURE 3.1

The Framework Fit Score — ten dimensions

	SCORE 1	SCORE 5	WHAT A HIGH SCORE PUSHES TOWARD
1 · Teams sharing one product	One team	Fifteen or more	Synchronised planning
2 · Work predictability	Priorities shift daily	Plannable weeks ahead	Iteration-based models
3 · Dependency density	Team delivers alone	Most work needs 2+ teams	Coordination mechanisms
4 · Regulatory burden	None	Authorisation at risk	Compliance overlays — not a framework choice
5 · Product complexity	Single product	Many products, shared platform	SAFe over LeSS
6 · Release frequency needed	Quarterly acceptable	Multiple times daily	Continuous delivery practice
7 · Operational load	Under 10% of capacity	Over 40%	Flow-based management
8 · Engineering maturity	Manual, infrequent integration	Trunk-based, full automation	A PRECONDITION — fix this first
9 · Organisational maturity	Command and control	Decisions at the edge	LeSS viability
10 · Co-location / overlap	Distributed, little overlap	Co-located	LeSS viability

Read the dimensions, not a total. Dimension 8 is a precondition rather than an input: below a threshold every framework performs badly and gets blamed for it. Dimension 4 is not a branch in the decision — it is a layer over the whole thing.

Figure 3.1 The Framework Fit Score — ten dimensions

Key Takeaways

- **Framework selection is a fit problem, not a ranking problem.** The shape that matches your context matters more than any framework's merits.
- **Size is one dimension of ten**, and rarely the decisive one.
- **Engineering maturity is a precondition, not a dimension.** Below a threshold every framework performs badly and gets blamed for it.
- **Regulatory burden is not a branch in the decision.** It is a layer over the whole thing.
- **Hybrid is a legitimate terminal answer**, and it is what most mature organisations actually run. It requires standardised interfaces, not standardised interiors.
- **"Constant failure despite doing important work" is a measurement artifact.** Check the model before concluding the team is underperforming.
- **Write exit criteria at adoption time.** They become unwriteable once roles are filled.

Apply This Monday

1. **Score your context on all ten dimensions.** Ten minutes. Then score your most different team separately and compare — if the profiles diverge sharply, you have a hybrid case you are probably not acknowledging.
2. **Check D8 honestly.** If engineering maturity is low, your framework question is premature.
3. **Measure coordination overhead** as a share of capacity. Above 15%, ask what it is buying.
4. **Count dependencies discovered at your last planning event.** Near zero means the event may have outlived its purpose.
5. **Write exit criteria for your current framework.** If you cannot, you have adopted something you cannot leave.

Self-Assessment

1. Why did you choose your current framework? Is the reason still true?
2. Do all your teams run the same model? Should they?
3. What is your coordination overhead, and what is it buying?
4. Is any team consistently missing sprint goals because of interrupt load rather than capability?
5. What would tell you your framework is no longer right?
6. Is engineering maturity high enough that a framework is your actual constraint?
7. Are your cross-team interfaces standardised, or are you standardising team interiors instead?

Common Mistakes

1. **Choosing by size.** One dimension of ten, and rarely the decisive one.
2. **Adopting a scaling framework to fix team-level problems.** It amplifies capability; it does not create it.
3. **Treating compliance as a branch** rather than a layer over everything.
4. **Forcing one model on heterogeneous work** for the sake of consistency, and reading the resulting demoralisation as underperformance.
5. **Never revisiting the choice.** Contexts change; framework decisions almost never do.
6. **Adopting without exit criteria**, and discovering later that they are politically unwriteable.

Part II turns from the choice of model to the things that determine whether any model works: how teams are designed, how product decisions are made, and what the cadence actually costs.

4

Designing Teams for Cognitive Load and Compliance

Your dependency graph is generated by your org chart. Team design is therefore not an HR exercise — it is the highest-leverage architectural decision you make.

The Problem

Most enterprise team structures are historical. They reflect a product line that was reorganised, an acquisition, a manager who built a group, a specialism that was scarce five years ago. Nobody designed them, and nobody can explain them.

The consequences show up somewhere else entirely. Delivery is slow, so the organisation invests in coordination. Dependencies are painful, so it adds dependency management. Handoffs lose context, so it adds documentation requirements. Every one of these is a treatment for a symptom whose cause is that the teams are drawn wrong.

Conway's Law states the mechanism: organisations produce designs that mirror their communication structures.[1] The practical corollary is the one that matters here — **you cannot manage your way out of a dependency graph your organisation chart generates.** You can only redraw the chart or accept the graph.

What the Textbooks Say

The standard Agile guidance is that teams should be cross-functional and self-organising: they should contain all the skills needed to deliver, and decide how to work.

Both are correct, and neither is sufficient at enterprise scale. "Contains all the skills needed" is straightforward for a team building one thing. It becomes ambiguous when a team's work spans several systems, when specialist skills are scarce, and when some required capabilities — security engineering, compliance, platform operations — cannot be replicated twenty-five times.

The guidance also says nothing about how many teams there should be, where the boundaries fall, or what happens when a team's scope exceeds what its members can reasonably hold in their heads.

What Enterprise Reality Adds

Cognitive load is the binding constraint, not headcount. A team's effective scope is bounded by how much of the system its members can understand well enough to change safely. Two teams of eight with clear, narrow domains outperform one team of sixteen owning both — not because of team size mechanics but because the second arrangement asks each person to hold twice as much context. Team size is a proxy; cognitive load is the thing.[2]

Some capabilities cannot be embedded everywhere. You will not have a security engineer, an ISSO and a platform specialist on each of twenty-five teams. Pretending otherwise produces a design that fails in implementation and defaults to whatever informal arrangement fills the gap.

Team boundaries determine your dependency graph, and the graph determines your lead time. This is the single most under-appreciated fact in enterprise delivery. Most organisations spend heavily managing dependencies and almost nothing on the structure producing them.

In regulated organisations, where compliance sits is a delivery decision. Compliance as an external function creates a dependency on every change. Compliance embedded creates a scarce-resource problem. Neither is obviously right and the choice has to be made deliberately, because the default — external — is chosen by inertia and is usually wrong.

Four Team Types

The most useful vocabulary for this comes from Skelton and Pais, who identify four fundamental team types and three interaction modes.[3] Applied to a regulated enterprise:

Stream-aligned teams own a flow of work for a product or a customer-facing capability. They are the majority of your teams and everything else exists to make them effective. In our organisation these are the feature teams on each product line.

Platform teams provide internal capability as a service — CI/CD, environments, deployment tooling, shared infrastructure. Their measure of success is that stream-aligned teams can self-serve without raising a ticket. A platform team that is a queue is not a platform team; it is a bottleneck with a service catalogue.

Enabling teams raise capability and then leave. Agile coaching is the obvious example; security engineering is often better as an enabling team than an embedded specialist. Their success is measured by their own redundancy, which makes them politically fragile and worth protecting.

Complicated-subsystem teams own something requiring deep specialist knowledge — a search engine, a processing pipeline, a cryptographic component. They exist because the cognitive load genuinely cannot be distributed. Keep the count low; every one is a permanent dependency.

The three interaction modes matter as much as the types. **Collaboration** — two teams working closely for a bounded period, high bandwidth, expensive, appropriate for discovery. **X-as-a-service** — one team consumes another's output with minimal interaction; the default for platform relationships and the cheapest mode. **Facilitating** — one team helps another improve; the enabling mode.

Most dysfunction is a mode mismatch. A platform relationship stuck in collaboration mode is a queue. An enabling relationship that becomes permanent collaboration is a dependency nobody planned.

Where Security and Compliance Sit

The question with the largest delivery consequence in a regulated organisation, and the one most often settled by default.

MODEL	COST	RESPONSIVENESS	CONSISTENCY	SCALING LIMIT
Fully embedded *A specialist on every team*	Very high; usually infeasible	Excellent	Poor — practice diverges	Specialist supply
Shared across 2–3 teams *One specialist rotating*	Moderate	Good until contention	Moderate	Becomes a queue at ~4 teams
Enabling team *Specialists raise team capability, then withdraw*	Moderate, front-loaded	Good once capability transfers	Good — one practice taught	Requires teams willing to own it
Platform-provided *Capability delivered as tooling and guardrails*	High to build, low to run	Excellent	Excellent — enforced, not taught	Only covers what can be automated
External gate *(the default)* *Central function reviews completed work*	Low apparent cost	Poor	Good	Fails immediately at scale

The answer is usually a combination, sequenced. Automate what can be automated into the platform (Ch 16). Use an enabling team to raise capability on what cannot. Retain a small central function for the genuinely irreducible: authorisation decisions, risk acceptance, assessment liaison, control interpretation.

The third edition of this book recommended embedding one specialist across two or three teams. That is the shared model, and it is the usual compromise and the usual bottleneck — it works at three teams and becomes a queue at five. Fine as a transition; not a destination.

The failure mode to name explicitly: the external gate is nobody's decision. It is what exists when nothing was decided, and it is the arrangement Chapter 13 spends a chapter dismantling.

Cognitive Load in Practice

Three questions to size a team's scope:

How many distinct domains must a member understand to work safely? More than two or three and handovers within the team become as costly as handovers between teams.

How long does a new joiner take to make an unsupervised change? Over three months suggests the scope exceeds what can reasonably be held.

When something breaks, how many people can diagnose it? If one, you do not have a team; you have a person and some colleagues.

Three overload signals worth watching: **specialisation inside the team** — members who only touch their part, which is a silo at smaller scale; **rising defect rates in the least-touched area**, the part nobody quite owns; and **estimation accuracy that varies wildly by work area**, which indicates the team understands some of its scope and not the rest.

In regulated environments, controls are part of cognitive load. A team owning components under a substantial control baseline carries the burden of knowing which controls apply, what evidence is needed, and what triggers reassessment. Loading such a team with the same functional scope as an unregulated one overloads it — invisibly, because the extra load has no story points attached to it.

FROM THE FIELD

When we first appointed Scrum Masters, many were former team leads who unconsciously continued

assigning tasks and making technical decisions for the team. The breakthrough came when we reframed the

role publicly in a leadership meeting: *"Your success is measured not by what tasks you assign, but by

what blockers you remove."* We tracked impediment resolution time as a key Scrum Master metric. Within

two PIs, our top Scrum Masters had established relationships with infrastructure, security, and

compliance teams that reduced cross-team dependency resolution time from an average of 5 days to under

2.

What I would add is where that improvement actually came from. Those Scrum Masters were not resolving

dependencies faster through diligence. They were building **standing relationships** with the functions

that generated the dependencies — which is what a good enabling or facilitating interaction looks like,

arrived at informally.

The uncomfortable implication is that we solved an organisational design problem with individual

effort. It worked, and it was fragile: it depended on specific people and would not have survived their

departure. Chapter 10 is about removing dependencies structurally rather than relying on someone being

good at navigating them.

Psychological Safety in a Compliance Culture

Psychological safety — the shared belief that the team is safe for interpersonal risk-taking — is consistently identified as the strongest predictor of team effectiveness.[4] In a compliance environment it faces a specific pressure that the general literature does not address.

Compliance cultures are, correctly, intolerant of certain failures. Some mistakes have external consequences. That intolerance is appropriate, and it creates a gradient: raising a problem you caused carries more risk here than elsewhere.

The result is a specific and dangerous failure mode. **The information most valuable to a compliance programme — "I think we may have a gap" — is the information most costly to**

surface. Teams do not go silent about everything. They go silent about exactly the thing you most need to hear.

Three things help, and none is a poster campaign. **Separate the finding from the finder** — treat self-reported gaps categorically differently from discovered ones, visibly and consistently. **Make raising a concern a recorded, unremarkable act** rather than an escalation; if the only route is escalation, the bar is set at "certain," and by then it is late. And **have a leader surface their own error first**, publicly. Until that happens, nobody believes the policy.

> **ANTI-PATTERN — THE COMPONENT TEAM**
>
> **What it looks like:** teams organised around technical layers — a front-end team, a back-end team, a
>
> database team. Each is efficient within its layer and staffed by specialists.
>
> **Why it fails:** every customer-facing change requires all of them, in sequence, with handoffs. Local efficiency is high and lead time is terrible. Worse, no team owns a customer outcome, so nobody is accountable for whether the feature worked.
>
> **The tell:** most features appear on three or more teams' boards, and estimating anything requires a meeting.
>
> **What to do instead:** organise around streams of change. Accept lower specialist utilisation in exchange for shorter lead time — that trade is almost always correct, and resistance to it is usually an argument about utilisation, which is not a metric worth optimising (Ch 11).

Metrics

METRIC	DIAGNOSES	WARNING SIGN
Cross-team dependencies per feature	Whether boundaries are drawn well	Above 2 consistently
Time for a new joiner to ship unsupervised	Cognitive load	Over 3 months
Bus factor per critical area	Concentration risk	1 anywhere that matters
Interaction mode mismatches	Structural friction	Any permanent collaboration that should be as-a-service
Self-reported vs. discovered compliance gaps	Psychological safety, measurably	Ratio falling
Specialist wait time	Whether the placement model is holding	Rising means the shared model is saturating

The self-reported-to-discovered ratio is the best available proxy for psychological safety in a compliance context, and almost nobody tracks it. It is also uncomfortable to report upward, which is itself informative.

FIGURE 4.1

Team types in a regulated organisation

	PURPOSE	INTERACTION MODE	SUCCESS LOOKS LIKE	FAILURE MODE
Stream-aligned	Owns a flow of work for a product or capability	Consumes as-a-service; collaborates briefly	Delivers end to end without handoff	Becomes a component team with a product name
Platform	Internal capability as a self-service product	X-as-a-service	Teams self-serve without raising a ticket	A queue with a service catalogue
Enabling	Raises capability, then withdraws	Facilitating	Its own redundancy	Permanent collaboration nobody planned
Complicated-subsystem	Deep specialist knowledge that cannot be distributed	X-as-a-service	Kept rare and well-bounded	Every one is a permanent dependency
Security / compliance	Assurance capability — placement is a delivery decision	Embedded, shared, enabling or platform-provided	Provided as a service or embedded	An external gate — chosen by inertia

Most dysfunction at scale is an interaction-mode mismatch — usually a platform relationship stuck in collaboration mode, which is a queue. Team type taxonomy after Skelton and Pais; original rendering.

Figure 4.1 Team types in a regulated organisation

FIGURE 4.2

Where security and compliance specialists sit

	COST	RESPONSIVENESS	CONSISTENCY	SCALING LIMIT
Fully embedded	Very high; often infeasible	Excellent	Poor — practice diverges	Specialist supply
Shared across 2–3 teams	Moderate	Good until contention	Moderate	Becomes a queue at ~4 teams
Enabling team	Moderate, front-loaded	Good once capability transfers	Good — one practice taught	Teams must be willing to own it
Platform-provided	High to build, low to run	Excellent	Excellent — enforced	Only what can be automated
External gate (the default)	Low apparent cost	Poor	Good	Fails immediately at scale

The external gate is not a decision — it is what exists when nothing was decided. The shared model is the usual compromise and the usual bottleneck: it works at three teams and queues at five. The answer is normally a sequenced combination.

Figure 4.2 Where security and compliance specialists sit

Key Takeaways

- **Your dependency graph is generated by your org chart.** You cannot manage your way out of it; you can only redraw the chart or accept the graph.
- **Cognitive load, not headcount, bounds a team's effective scope.** Team size is a proxy for the real constraint.
- **Four team types and three interaction modes.** Most dysfunction is a mode mismatch — usually a platform relationship stuck in collaboration.
- **Where compliance sits is a delivery decision.** The external-gate default is what exists when nobody decided.
- **The shared-specialist model works at three teams and queues at five.** A transition, not a destination.
- **Controls are cognitive load.** Teams under a heavy baseline carry invisible burden that has no story points attached.
- **In a compliance culture, the most valuable information is the most costly to surface.** Separate the finding from the finder, and have a leader go first.
- **Solving structural problems with individual effort works and is fragile.**

Apply This Monday

1. **Count cross-team dependencies for your last ten features.** Above two consistently means the boundaries are wrong, not that coordination is weak.
2. **Classify each of your teams by type**, and each significant relationship by interaction mode. Mismatches are your friction list.
3. **Ask how long a new joiner takes to ship unsupervised.** Over three months is a cognitive load signal.
4. **Name your compliance placement model.** If the answer is "they review things at the end," that is the default, not a decision.
5. **Check the ratio of self-reported to discovered compliance gaps** over the last two quarters.

Self-Assessment

1. Can you explain why your teams are drawn the way they are, other than history?
2. How many teams must cooperate for a typical feature to reach a customer?
3. Which teams are stream-aligned, and which are component teams with a product name?
4. Is your platform team a service or a queue?
5. Where do security and compliance specialists sit, and was that decided or inherited?
6. Do your regulated teams carry a lighter functional scope to account for control burden?
7. When did someone last raise a compliance gap they had caused themselves?
8. Which critical areas have a bus factor of one?

Common Mistakes

1. **Treating team design as an HR exercise.** It is the highest-leverage architectural decision available.
2. **Component teams with product names.** The org chart says stream-aligned; the dependency graph says otherwise.
3. **Optimising specialist utilisation.** High utilisation and long lead time are the same phenomenon.
4. **Loading regulated teams identically to unregulated ones.** Control burden is real and invisible.
5. **Leaving compliance placement to default.** The external gate is chosen by inertia.
6. **Assuming psychological safety generalises.** In a compliance culture the gradient is specific and predictable.

Chapter 5 turns to what these teams are pointed at — and to the difference between shipping things and changing outcomes.
