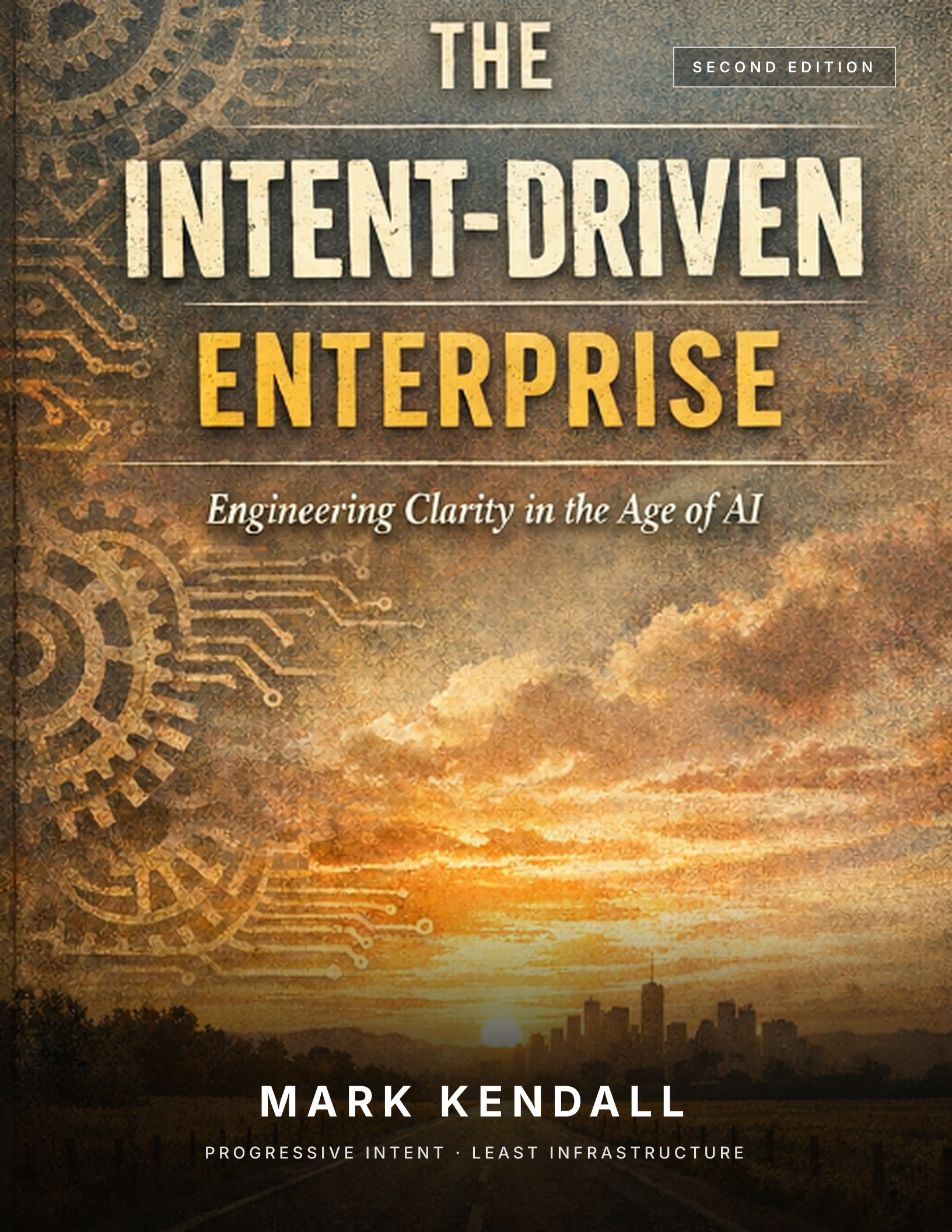
THE INTENT-DRIVEN ENTERPRISE

SECOND EDITION

Engineering Clarity in the Age of AI



MARK KENDALL

PROGRESSIVE INTENT · LEAST INFRASTRUCTURE

SECOND EDITION

The Intent-Driven Enterprise

Progressive Intent, Least Infrastructure: how engineering teams turn agents into measurable outcomes

Mark Kendall
Founder, LearnTeachMaster · Enterprise AI Architect

learnteachmaster.org intent-driven-engineering.com

The Intent-Driven Enterprise — Second Edition
Progressive Intent, Least Infrastructure

Copyright © 2026 Mark Kendall. All rights reserved.
No part of this publication may be reproduced or transmitted in any form without the prior written permission of the author.

Published by LearnTeachMaster · learnteachmaster.org
First edition 2026 · Second edition 2026
ISBN: [to be assigned]

Tool names (GitHub Copilot, Cursor, Claude Code, Spec Kit, Kiro and others) are trademarks of their respective owners and are used descriptively. Tool mechanisms described reflect common practice at the time of writing and will evolve.

Metrics in scenarios and examples are illustrative unless a source is stated. Measure your own baseline.

Contents

Preface to the Second Edition 4

Prologue 6

PART I · THE BREAKING POINT, REVISITED

1 The AI Adoption Illusion, Again 9

2 Agents Everywhere, Results Nowhere 12

3 From the Prompt Trap to the Spec Trap 15

4 The 60-Day Executive Ultimatum 17

PART II · THE SHIFT

5 Why AI Initiatives Still Stall 21

6 From Prompts and Specs to Intent 24

7 What Is Intent-Driven Engineering 27

8 Progressive IDE: The Execution Model 29

9 The Least-Infrastructure Principle 34

10 The Core Mental Model Shifts 38

PART III · THE CORE MODEL

11 Defining Intent 42

12 Intent as a First-Class Artifact 46

13 Intent Across Your Tools 49

14 Determinism Where It Counts 52

15 Measuring What Actually Matters 54

16 The Full Loop: Intent → Agent → Evidence → Outcome 57

PART IV · REUSE WITHOUT A REGISTRY

17 What Happened to the Registry 62

18 Skills, Plugins and Starter Repos 65

19 Turning Knowledge into Systems 68

20 Intent as Code 71

PART V · JUST ENOUGH PLATFORM

21 When You Actually Need a Platform 74

22 Agents, Skills, Tools and Systems 76
23 Orchestration: Start Sequential 79
24 MCP: Connect, Don't Build 81
25 Governance Without Bureaucracy 83

PART VI · ARCHITECTURE IN PRACTICE
26 AI in the Architecture You Already Have 87
27 Observability and the Evidence Loop 89
28 How Teams Actually Work with This 92

PART VII · ENTERPRISE EXECUTION
29 The Pipeline You Already Have 95
30 Scaling Across Many Repositories 97
31 Organizational Adoption 99

PART VIII · ANTI-PATTERNS
32 Prompt Addiction and Spec Bloat 103
33 Platform Before Proof 106

PART IX · REAL SYSTEMS
34 From Idea to Running System 109

PART X · THE FUTURE
35 Owning the Practice 114
Summary 117
The Monday Morning Commitment 118
A Final Word 119
Appendix A · The INTENT.md Template 120
Appendix B · Progressive IDE Quick Reference 121
Appendix C · Readiness Checklist 122
Appendix D · Tool Pointer Snippets 123
Resources 124

PREFACE TO THE SECOND EDITION

What We Learned, and What We Got Wrong

The first edition of this book was written at a particular moment. AI coding assistants were spreading through enterprise engineering teams. Most initiatives were producing demos and dashboards instead of outcomes, and the field needed a way to turn informal AI use into engineered systems.

The diagnosis in that edition was right, and it still is. The prescription went too far.

The first edition proposed an architecture: YAML intent files with typed schemas, an intent compiler, an executor registry, a central intent registry, a five-layer agentic platform and a three-tier measurement system wired into all of it. Every piece made sense on its own. Together they asked teams to build a great deal of infrastructure before they had proven anything.

Then the world moved, faster than any book could.

Within a year, nearly every engineering team had an agent in the editor. GitHub Copilot moved from autocomplete to an agent that opens pull requests. Cursor became the default editor for a large share of developers. Claude Code made the terminal an agentic workspace. Spec-driven tools such as GitHub Spec Kit and Kiro turned "write the spec first" into a product category. Rules files, instruction files, skills, plugins and MCP servers became the ordinary furniture of a repository.

Those tools absorbed much of what the first edition said a team had to build. The agent now reads the intent, plans the work, calls the tools, writes the code, runs the tests and reports back. In the old model that was the compiler and the runtime. You no longer have to build them, because you already have them.

We also learned some things the hard way.

The registry never panned out. In practice, engineers do not browse catalogs. Central registries went stale faster than anyone maintained them. The entries that mattered were the ones living next to the code, where the agent could read them. Reuse did happen, but through starter repositories, shared skills and plugins, not through a registry someone had to search.

More specification did not produce better software. Teams that wrote forty-page specs for their agents got forty pages of drift. The agent followed the letter of the spec and missed its point. The teams that won wrote the smallest intent that could succeed, ran it, looked at the result and added detail only where the evidence showed a gap.

Infrastructure before proof is the new tool hoarding. Several organizations built impressive agent platforms that ran almost nothing. The platform was real. The outcomes were not.

This second edition is rebuilt around those lessons. The core is still intent: the declaration of what must be true when the work is done, and how we will know. Around that core, the book now argues for the **least specification and the least infrastructure that reliably produce the desired outcome**, and for a practical execution model that gets there. We call that model **Progressive IDE**. In this edition, Progressive IDE is simply what Intent-Driven Engineering means in practice: the practical execution model for building better software with less specification.

If you read the first edition, you will recognize most of this book. Its structure, its arguments about activity and outcomes, and its insistence on success criteria and evidence are all still here. What is gone is the assumption that discipline requires a platform. In this edition, discipline requires an intent, a test and the willingness to look at the evidence. Everything else is earned.

— Mark Kendall Wylie, Texas · 2026

PROLOGUE

The Same Room, a Year Later

It happened in a steering committee. Again.

The engineering leader had prepared for thirty minutes. This time the slides were better. Agent adoption was at 94% of developers. AI-assisted pull requests were up 3.1× quarter over quarter. Forty percent of merged code was "AI-authored." There was a live demo of an agent taking a ticket from the backlog to an open pull request in eleven minutes.

The CTO waited until the slides were done.

Then she asked the question that ended the presentation, the same question that ended it a year earlier:

> "What measurably changed because of any of this?"

The leader had more data than ever and still could not answer. Lead time to production had not moved. Escaped defects were slightly up. Review queues were longer, because the agents produced pull requests faster than humans could read them. The organization had replaced *prompt activity* with *agent activity* and called it progress.

The tools had changed completely. The question had not changed at all.

That is the point of this edition. Every generation of AI tooling will make activity easier to produce and easier to count. None of them will answer the CTO's question on their own. Only an engineering discipline does that: say what outcome you want, do the least necessary to get it, and prove that you got it.

This book was written from that room. Twice.

Who This Book Is For

Engineering leaders whose teams already have Copilot, Cursor or Claude Code and who are not satisfied with what the adoption dashboard is telling them. By Chapter 4 you will have the language to ask the right questions. By Chapter 15 you will have a measurement model you can stand behind. By Chapter 31 you will have an adoption playbook that does not start with a platform budget.

Senior engineers and architects who will implement what leaders decide. Every pattern in this edition is designed to run in the repository you already have, with the agent you already use and the CI you already operate. Where something genuinely needs a platform, the book says so and explains how to tell.

Engineering teams who sense that their agent use is fast but not compounding. That instinct is correct. Progressive IDE is the structured form of it, and Chapter 8 will have you running the loop the same day.

What This Book Is Not

This is not a book about models. Models are the executors, and they will be better next quarter. The intent you write today should survive that upgrade unchanged.

It is not a book about prompting or about any one tool. Copilot, Cursor, Claude Code, Spec Kit and Kiro all appear here because they are what teams use. The discipline is the same across all of them.

It is not a platform blueprint. Part V explains when a platform earns its place. Most teams will need much less of one than they think.

How to Read This Book

Read Part I even if you think you know the problem. It has shifted: the trap is no longer the prompt but the spec and the platform. Part II is the hinge. Chapter 8 (Progressive IDE) and Chapter 9 (the Least-Infrastructure Principle) are the two chapters everything else depends on. Parts III through VII are operational, so read them with your editor open and an agent running. Part VIII names what will pull you backward. Part IX runs the whole loop on a real problem. The appendix is the reference you will come back to.

A note on numbers: where this edition shows metrics, they are either clearly labeled as illustrative scenarios or presented as measurements you should take yourself. Measure your own baseline, then move it.

PART I

The Breaking Point, Revisited

Why AI initiatives still fail to produce outcomes when every developer has an agent

1 The AI Adoption Illusion, Again
2 Agents Everywhere, Results Nowhere
3 From the Prompt Trap to the Spec Trap
4 The 60-Day Executive Ultimatum

PART I · THE BREAKING POINT, REVISITED

CHAPTER 1

The AI Adoption Illusion, Again

Walk into any enterprise engineering organization today and you will see the same pattern you saw a year ago, at a higher resolution. Every developer has an agent. The editor suggests whole functions. A background agent turns tickets into pull requests. The strategy deck says "AI-native." The adoption dashboard is green.

On the surface, it looks like progress.

Beneath the surface, the same thing is happening that happened before. The tools are being used. Outcomes are not being engineered.

The New Dashboard Problem

The first wave of AI adoption counted prompts, sessions and licenses. The second wave counts things that sound more serious: the percentage of code that is AI-authored, the number of agent-opened pull requests, suggestions accepted, tokens per developer.

These are still activity metrics. They describe how much the machine did. They say nothing about whether anything got better.

An agent that opens fifty pull requests a week is not a capability. If thirty of them are rejected, twelve need heavy rework and eight are merged and quietly introduce regressions, the agent has *increased* the load on the team while every activity metric shows a gain.

> Using AI is an activity. Engineering with AI is a discipline. The industry has become very good at the first and is still mostly unprepared for the second.

What Changed, and What Did Not

A great deal changed. Agents can plan, edit many files, run tests, read failures and try again. They can be pointed at a repository and given a goal. Spec-driven tools generate plans and task lists from a description. The mechanical gap between saying what you want and having working code in front of you has collapsed.

What did not change is the gap between working code and a measurable outcome. That gap was never mechanical. It is closed by three things no tool supplies on its own:

1. **A clear statement of the outcome**, meaning what must be true when the work is done.
2. **A definition of success that can be checked**, not felt.
3. **Evidence**: a record that the outcome happened and keeps happening.

Those three things are what this book calls *intent*. They are just as necessary when the agent writes all the code as when a human does. They may be more necessary, because the agent will cheerfully produce a great deal of something you did not quite mean.

The Questions That Survive Scrutiny

The question is never "how much AI are we using?" The question is "what measurably changed because of it?"

- Did lead time from ticket to production decrease?
- Did escaped defects decrease, or merely move?
- Did review time per change go down, or did review queues grow?
- Did time-to-first-merged-change for new engineers shorten?
- Did the cost of operating a given service change?

Most AI initiatives still cannot answer these questions. That is not because the tools are weak. It is because the initiative was designed to roll out tools, not to move numbers.

Why Smart People Build the Illusion

The people inside these initiatives are capable and hardworking. They are following the pattern the tools reward: try it, show it, scale the rollout. That is the right pattern for exploring a new capability and the wrong one for producing outcomes.

The pattern that produces outcomes is short and unglamorous: **state the intent, do the least that could work, check the evidence, refine.** It does not require a new platform. It requires a change in what the team treats as "done."

The LearnTeachMaster Reference

Throughout this book, the patterns are grounded in work done on and around LearnTeachMaster (learnteachmaster.org), a free, open platform for Intent-Driven Engineering. The first edition described that work through an elaborate reference architecture. This edition describes how the work actually ran: intent files in repositories, agents doing the execution, tests and CI doing the verification, and very little in between.

What This Chapter Has Established

1. Agent adoption creates the same illusion prompt adoption did: activity without compounding outcomes.
2. The missing piece is not capability. It is a stated outcome, a checkable definition of success and evidence.
3. The fix is a discipline, not a purchase: state the intent, do the least that could work, check the evidence, refine.

PART I · THE BREAKING POINT, REVISITED

CHAPTER 2

Agents Everywhere, Results Nowhere

The modern enterprise AI stack looks like abundance from the outside. Inside, it looks like a dozen dialects of the same idea.

How the Stack Gets Built Now

It still accumulates rather than being designed, but the layers are different.

One team standardizes on Copilot because it came with the GitHub enterprise agreement. Another team moved to Cursor because its senior engineers prefer it. A platform group pilots Claude Code for refactoring and migration work. An architecture group adopts a spec-driven toolkit for new services. Someone builds an MCP server for the internal ticketing system, and someone else builds a second one.

Each of those decisions is defensible. Each brings its own place to put instructions:

Tool	Where the instructions live
GitHub Copilot	`.github/copilot-instructions.md`, prompt files
Cursor	`.cursor/rules/`
Claude Code	`CLAUDE.md`, skills, plugins, hooks
Cross-tool convention	`AGENTS.md`
Spec-driven toolkits	spec, plan and task files per feature

Eighteen months in, a single repository may hold four instruction files that disagree with each other, a spec folder nobody has updated since the first sprint, and a rules directory copied from a blog post.

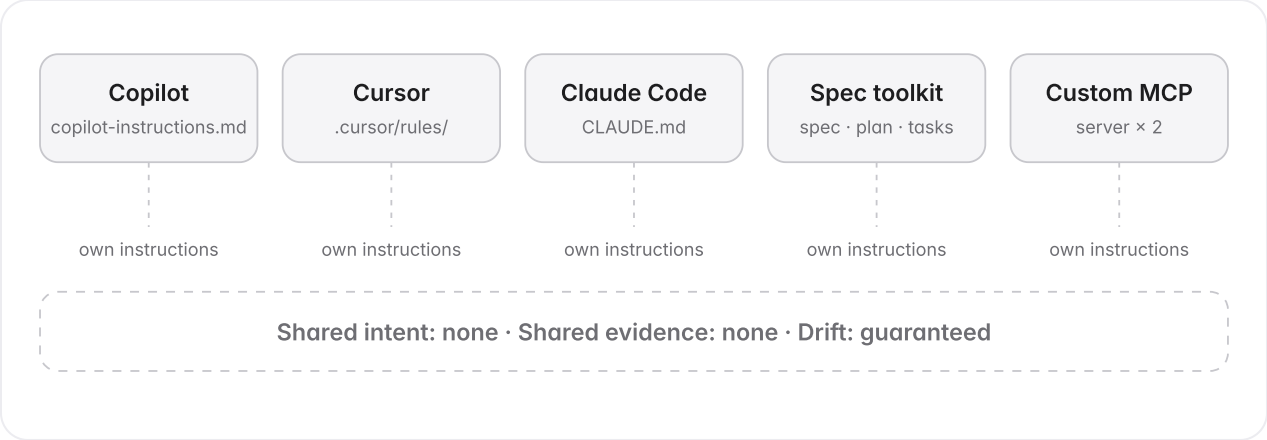


Figure 2.1 — The fragmented agent stack: many tools, many instruction formats, no shared intent

The Instruction-Drift Tax

The first edition described an *integration overhead tax*: every tool added without a shared model raised the cost of connecting it to everything else. Agents add a quieter version of that tax, *instruction drift*.

When the same expectation ("all public endpoints require authorization tests") lives in three tool-specific files, it will be updated in one and forgotten in two. Developers on different tools get different behavior from what is supposedly the same codebase. Nobody notices until an incident review finds that the Cursor users' changes skipped a check the Copilot users' changes included.

The Knowledge Silo Problem, Relocated

The deeper problem is still knowledge. It has moved from people's heads into private chat histories, personal rules files and agent sessions that end when the terminal closes. That is better than nothing, but it is not shared, versioned or checkable. The knowledge exists, and the system cannot use it.

What Intent-Driven Engineering Changes

The answer is not fewer tools. Teams should use the agent that makes them effective. The answer is one shared layer above the tools: **the intent lives in the repository, in plain markdown, once.** Tool-specific files point to it rather than restating it.

> Stop writing instructions for each tool. Write the intent once, next to the code, and let every tool read it.

That one change removes most of the drift. It needs no platform, no registry and no new product. It is a file and a habit. Chapter 13 shows exactly how to wire it into each tool.

PART I · THE BREAKING POINT, REVISITED
CHAPTER 3

From the Prompt Trap to the Spec Trap

The first edition called it *the prompt trap*: the belief that the cure for inconsistent AI output was a better, longer, more elaborate prompt. That trap is still open. It has been joined by a more sophisticated one.

The Prompt Trap, Briefly

A prompt is a natural-language instruction to a probabilistic system. It carries the author's intent implicitly, so when someone else runs it, the context that made it work is missing. The first edition named three failure modes, and they still apply:

1. **Context collapse.** The prompt works for its author and fails for everyone else, because the real requirements were never written down.
2. **Silent regression.** A model or context change degrades the output quietly instead of breaking loudly.
3. **Non-composable knowledge.** Natural-language output passed between steps accumulates ambiguity at every boundary.

The Spec Trap

The industry's response to the prompt trap was the right instinct taken too far: *write the specification first.* Spec-driven development gave agents a requirements document, a design document, a plan and a task list before any code was written.

For many teams this was a genuine improvement. For many others it became a new trap with three symptoms of its own.

Specification as a substitute for thinking. The tool generates a spec from a one-line request. The spec is long, plausible and largely unread. It feels like rigor and contains the agent's guesses about what you meant, now formatted as requirements.

Specification drift. The spec is written once and the code evolves. By the third iteration the spec describes a system that no longer exists, and the agent is reading it as the truth.

Over-constraint. The spec prescribes implementation details the author did not care about: library choices, file layouts, function names. The agent follows them faithfully, the result is worse than if it had been given room, and nobody can tell which constraints were essential and which were accidental.

> A forty-page spec is not a stronger intent. It is a weaker one, buried under thirty-eight pages of guesses.

What a Specification Is For

None of this argues against writing things down. It argues for writing down the *right* things, in the *right amount*, at the *right time*.

The things that must be written down are the ones nobody else can infer: what outcome is wanted, what constraints are non-negotiable, and how success will be checked. Everything else, including the plan, the file structure and the approach, the agent can usually propose, and you can review what it proposes.

That is the difference between a spec and an intent. A spec tries to describe the solution. An intent describes the problem, the boundaries and the test.

The Escape Route

The way out of both traps is the same. Stop at the point where your knowledge ends and the agent's competence begins. Write the outcome, the constraints and the success criteria. Let the agent do the rest. Then look at what it did, and add specification *only where the evidence shows the intent was not enough*.

That loop is Progressive IDE, and Chapter 8 makes it precise.

PART I · THE BREAKING POINT, REVISITED

CHAPTER 4

The 60-Day Executive Ultimatum

The conversation still arrives without warning, in a steering committee, a budget review or a direct message from a CTO who has been patient longer than planned. The message is still simple: *show me what we built, or we are changing direction.*

The Anatomy of the Ultimatum

The sequence is predictable, and with agents it simply runs faster.

Period	Phase	What it looks like
Months 1–2	Rollout	Licenses provisioned, enthusiasm high
Months 2–4	Proliferation	Agents everywhere, activity metrics climb
Months 4–6	Plateau	Demos impressive, review queues longer, outcomes unclear
Months 6–9	Reckoning	Leadership asks what changed
Months 9+	Ultimatum	Sixty days to show outcomes, or pivot

What Leadership Actually Needs to See

Leadership is asking a systems question: *is there a durable, repeatable capability that we now have and did not have before?*

- Can it run without the person who set it up?
- Do we know when it degrades?
- Is it in our normal delivery path, or off to the side?
- Can we show the metric it moves, and how far it has moved?
- If it goes wrong, can we see what changed and roll it back?

A demo answers none of these. A dashboard of agent usage answers none of these.

The Two Failure Responses

The demonstration acceleration. More demos, more impressive agent runs. Leadership has already seen what the tools can do. Capability is not the question.

The retroactive metric. Developer surveys, estimated hours saved, "percent of code written by AI." These are estimates and activity, and they do not discharge the obligation that came with the budget.

The Intent-Driven Answer

The first edition answered the ultimatum with a registry: a central catalog of deployed intents, each with a version, an owner and an outcome metric. The idea was right and the mechanism was too heavy. Most organizations never built the registry, and the ones that did struggled to keep it current.

The answer that works is lighter and more convincing: **a short list of intents, each with its evidence.**

Intent	Lives in	Owner	Baseline → Now	Evidence
PR review preparation	`platform/pr-review/INTENT.md`	Platform	6.4 h → 2.1 h review cycle	CI check history, PR analytics
Incident first-pass analysis	`sre/incident-triage/INTENT.md`	SRE	47 → 31 min MTTR	Incident tool export
New-service scaffold	`starter/INTENT.md`	Architecture	3 days → 4 h to first deploy	Repo creation log

Illustrative example.

Each row points to a file in a repository and a source of evidence that already exists: CI history, the incident tool, the analytics system. Nothing in the table required building a platform. It required that someone wrote down the intent, defined success before starting and kept the evidence.

> Teams with intent and evidence can answer the ultimatum. Teams with only agents and activity cannot.

The End of Part I

Chapter 1 showed the illusion, which is still with us: agent activity without outcomes. Chapter 2 showed the fragmentation: many tools, many instruction dialects, no shared intent. Chapter 3 showed the traps: the prompt trap and its successor, the spec trap. Chapter 4 showed the reckoning, and that it is answered with evidence, not infrastructure.

Part II introduces the shift: from prompts and specs to intent, and from building platforms to doing the least that works.

> The breaking point is not the end of the initiative. It is the beginning of the discipline.

PART II

The Shift

From prompts and specs to intent, and from platforms to the least that works

5 Why AI Initiatives Still Stall
6 From Prompts and Specs to Intent
7 What Is Intent-Driven Engineering
8 Progressive IDE: The Execution Model
9 The Least-Infrastructure Principle
10 The Core Mental Model Shifts

PART II · THE SHIFT
CHAPTER 5

Why AI Initiatives Still Stall

Enterprises have watched several waves of transformative technology arrive, generate enormous activity and retreat from their promises. Cloud transformation stalls became a genre. Digital transformation stalls followed. AI transformation stalls are now on their second generation.

The causes are structural, so they repeat. The first edition named five. Two of them still stand as written. Three have to be restated, because the first edition's remedy for them was itself a cause of stalling.

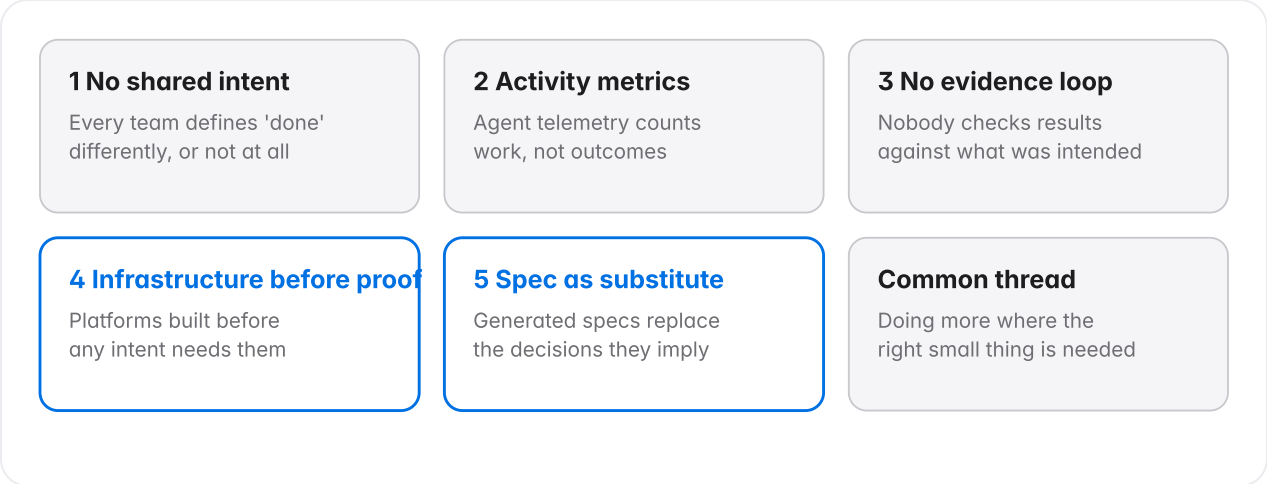


Figure 5.1 — Five root causes of AI stall, second edition

Root Cause 1: No Shared Intent

Engineering organizations have shared models for almost everything: APIs for how services talk, tickets for how work is tracked, pull requests for how code is reviewed. They rarely have a shared way of saying *what an AI-assisted piece of work is supposed to achieve and how anyone will know.*

The first edition proposed a typed YAML schema to fix this. The schema was not the essential part. The essential part was the habit of answering a handful of questions before work begins. A shared intent can be a markdown file with six headings. What makes it shared is that everyone uses the same six headings.

Root Cause 2: Activity Metrics

This cause is unchanged and has become more seductive. Agent telemetry produces impressive numbers automatically. Outcome numbers take deliberate effort. Teams measure what is easy, and what is easy is activity.

Root Cause 3: No Evidence Loop

The first edition called this "no feedback loop" and proposed a performance store, a criteria evaluator and automated recompilation. The real gap is simpler. Teams do not *look*. The agent produces a change, the change merges, and nobody checks it against what was intended. The evidence already exists in tests, CI results, review comments and production metrics. It is not connected back to the intent.

Root Cause 4: Infrastructure Before Proof

This is new, and it is partly this book's fault. The first edition listed "no agentic platform" as a root cause. Organizations that read that as a mandate built platforms first. They stood up orchestration layers, executor registries and intent APIs, then went looking for work to run on them. A platform with no proven intents is a cost center waiting for its own ultimatum.

The corrected root cause is the reverse: **building infrastructure before the intent has proven it needs it.** Platforms should be extracted from success, not installed in anticipation of it.

Root Cause 5: Specification as a Substitute for Judgment

Also new. When the agent can generate a plausible spec in seconds, teams accept the spec instead of making the decisions it pretends to contain. The organization ends up with a great deal of documentation and very little shared understanding of what matters.

The Common Thread

All five causes are failures of discipline, not of technology. Three of them have the same shape: doing *more* (more metrics, more infrastructure, more specification) in place of doing the *right small thing* and checking it.

> Stalling is not a technology problem. It is a discipline problem. And the discipline is mostly subtraction.

PART II · THE SHIFT
CHAPTER 6

From Prompts and Specs to Intent

The shift to intent is still the central move of this book. What has changed is what we are shifting *from*. In the first edition the opposite of intent was the prompt. Now there are two opposites: the prompt, which says too little, and the bloated spec, which says too much.

Three Artifacts, Three Jobs

	Prompt	Spec	Intent
Describes	An instruction	The solution	The outcome, boundaries and test
Typical length	A sentence to a page	Many pages	Usually under one page
Written by	Whoever is at the keyboard	Often generated	The person accountable for the result
Survives a model upgrade	Rarely	Partly	Yes
Checkable	No	Sometimes	By design
Lives	In a chat	In a docs folder	Next to the code it governs

A prompt carries intent *implicitly*. It is the author's intent compressed into an instruction, with the context left in their head. A spec carries intent *diluted*: the essential decisions are present but mixed with dozens of incidental ones. An intent carries only the decisions that must be made by a human and checked by evidence.

What Intent Is

In Intent-Driven Engineering, **intent is a short, explicit statement of what must be true when the work is done, what must not be violated along the way and how success will be checked.**

It answers six questions:

1. **Outcome.** What is different in the world when this is done?
2. **Inputs.** What does the work start from?
3. **Outputs.** What is produced, and in what shape?
4. **Success criteria.** How will we know, checkably, that it worked?
5. **Constraints.** What must not be violated?
6. **Evidence.** What will we keep that shows it worked, and keeps working?

The first edition asked four questions (inputs, outputs, success, boundaries). This edition adds *outcome* at the front, because without it the other answers optimize the wrong thing, and *evidence* at the back, because without it success is a one-time claim rather than a durable fact.

The Compilation Analogy, Updated

The first edition compared intent files to source code and the IDE compiler to a real compiler. The analogy still helps, but the parts have moved.

Traditional software	First edition	Second edition
Source code	Intent file (`.intent.yaml`)	`INTENT.md` in the repo
Compiler	IDE Compiler	The agent you already use
Type checker	Schema validation	Tests, linters, CI checks
Runtime	Agentic platform (WEGA)	Your existing services and pipelines
Unit tests	Success criteria evaluator	Success criteria *as* tests
Version control	Intent registry with semver	Git

> The agent is the compiler. The tests are the type checker. Git is the registry. You already own all three.

This is the most important change in the book. Everything the first edition asked you to build exists in the tools your team uses every day. The discipline is what was missing, and the discipline costs a file and a habit, not a platform.

The Question That Still Matters

The first edition ended its version of this chapter with a question. It is still the right one, with one word changed:

> If the person who set this up left tomorrow, could the work continue without them, and could we tell whether it was still working?

If the answer is no, you have a prompt or a spec. If the answer is yes, you have an intent.

PART II · THE SHIFT
CHAPTER 7

What Is Intent-Driven Engineering

The first edition defined the discipline this way:

> *A software engineering discipline in which AI capabilities are expressed as formal, versioned intent artifacts, compiled into executable systems, executed through an agentic platform, and continuously measured against defined outcomes.*

Each part of that definition had a reason, and two of them were wrong for most teams. "Compiled into executable systems" and "executed through an agentic platform" described an architecture, not a discipline. The second edition definition is shorter:

> **Intent-Driven Engineering is a software engineering discipline in which the desired outcome, its constraints and its success criteria are stated explicitly, before the work begins, in the smallest form that can succeed, and the work, whoever or whatever does it, is accepted on evidence.**

Every clause still carries weight.

"A Software Engineering Discipline"

IDE is not an AI strategy or a prompt methodology. It inherits the full apparatus of software engineering: version control, review, testing, continuous integration and monitoring. AI-assisted work belongs in the same delivery path as any other work, held to the same standard. It does not belong in an innovation lab with lower standards.

"The Desired Outcome, Its Constraints and Its Success Criteria Are Stated Explicitly"

The intent is written down in words a reviewer can read and an agent can act on. It is not implied by a ticket title or carried in someone's head. *Explicit* does not mean *formal*. A well-written markdown file is explicit. A schema nobody reads is formal and useless.

"Before the Work Begins"

Success criteria written after the fact are rationalizations. The criteria must exist before the agent runs, so the result can be judged against something that was not shaped by the result.

"In the Smallest Form That Can Succeed"

This is the clause the first edition lacked, and it is the one that most changes practice. The goal is not a complete specification. It is the *minimum* intent that lets the work succeed, with more detail added only when evidence shows it is needed. Chapter 8 turns this clause into a method.

"Whoever or Whatever Does It"

The intent does not care whether Copilot, Cursor, Claude Code, a spec-driven toolkit or a person does the work. The same intent should produce acceptable work from any competent executor. If it only works with one tool, the tool's quirks have leaked into the intent.

"Accepted on Evidence"

Work is done when the evidence says so: the tests pass, the success criteria are met and the outcome metric moves. It is not done because the agent says it is, or because the diff looks reasonable. Evidence is what turns AI output into engineering.

What Is Not in the Definition

There is no compiler, registry, platform or schema in the definition. Those may appear later, when a specific intent proves it needs one. They are not part of the discipline. They are implementation choices, and the discipline's job is to keep them from being made too early.

PART II · THE SHIFT
CHAPTER 8

Progressive IDE: The Execution Model

This is the hinge of the second edition.

Chapter 7 said the intent should be stated *in the smallest form that can succeed*. Progressive IDE is how you find that form in practice. You don't guess it up front. You discover it through a short loop that adds specification only where the evidence demands it.

Progressive IDE was developed through the work behind this book as a practical specification model. In this edition it is simply how Intent-Driven Engineering runs: **the practical execution model for building better software with less specification.**

The Loop

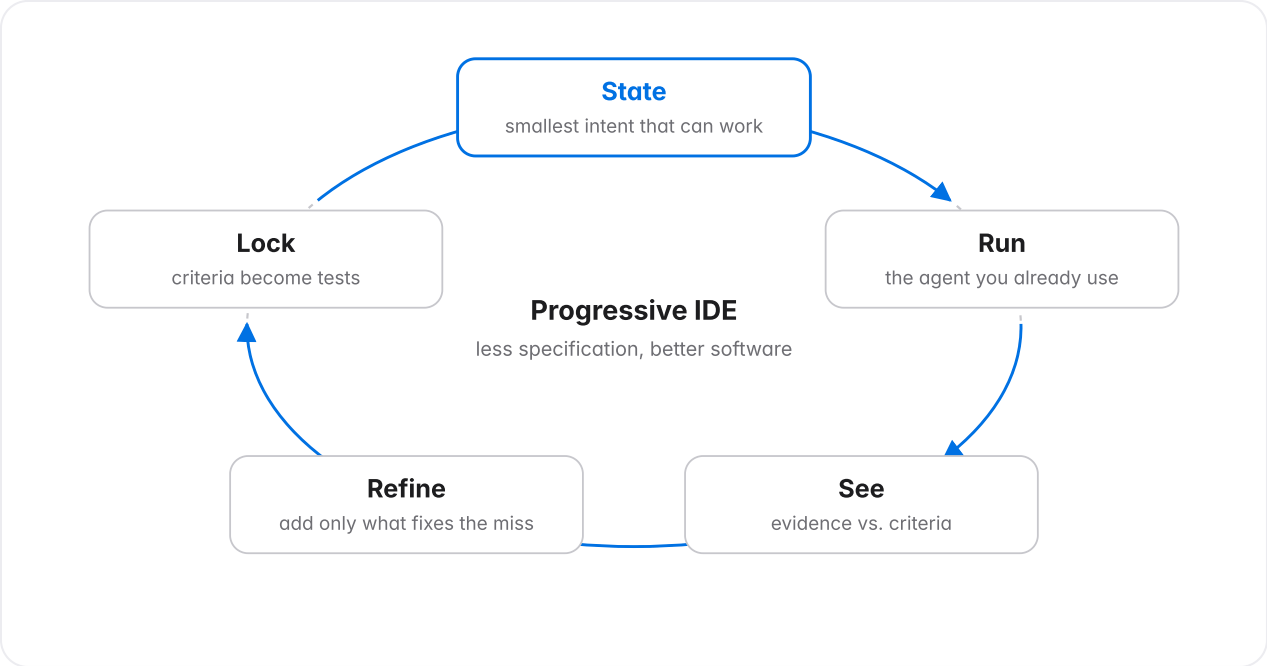


Figure 8.1 — The Progressive IDE loop: state, run, see, refine, lock

1. State. Write the smallest intent you believe could succeed. For many tasks that is a paragraph: the outcome, one or two hard constraints and how you will check it.

2. Run. Hand it to the agent you already use. Do not pre-plan its approach. Let it propose.

3. See. Look at the evidence: the diff, the test results, the behavior. Compare it to the intent, not to what you would have written yourself.

4. Refine. Where the result missed, ask *why*. If the intent was ambiguous, sharpen the intent. If the agent lacked context, add the context. If a constraint was missing, add the constraint. Add only what explains the miss.

5. Lock. When the result meets the success criteria, turn the criteria into tests and checks so the result stays true. The intent and its evidence are committed together.

Most tasks finish in one or two passes. Some take five. Very few need the forty-page spec they would have received up front.

The Specification Ladder

Progressive IDE treats specification as a ladder you climb only as far as the evidence requires.

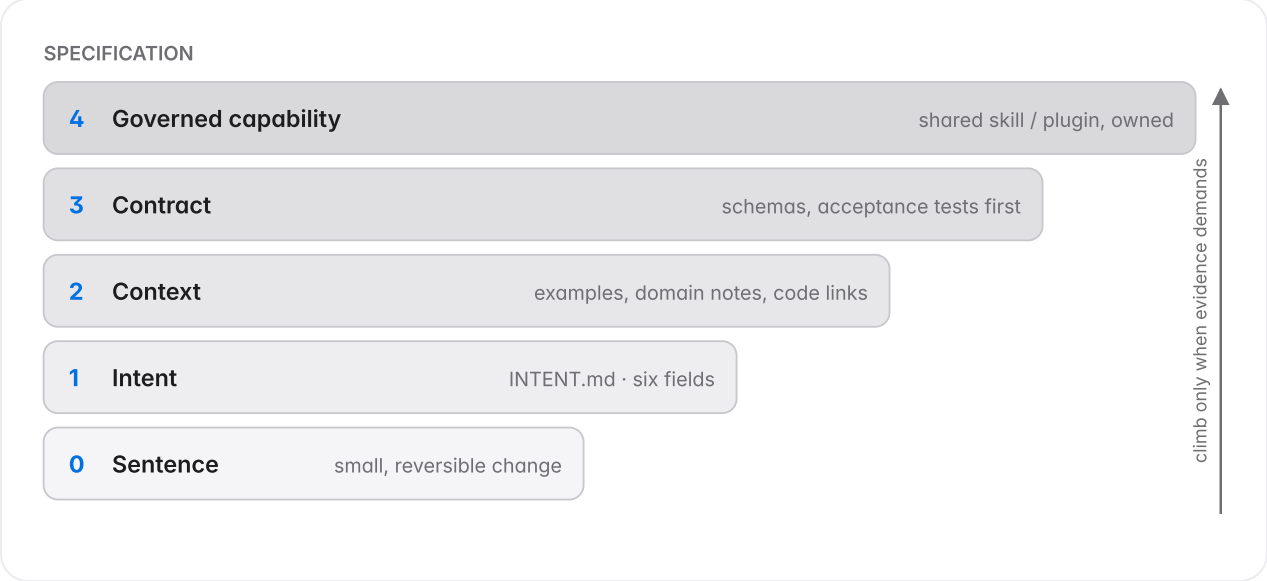


Figure 8.2 — The specification ladder: climb only as high as the evidence requires

Rung	What you add	Add it when
0 — Sentence	One-line outcome	Small, well-understood, reversible change
1 — Intent	`INTENT.md`: outcome, inputs, outputs, success, constraints, evidence	Anything worth reviewing or repeating
2 — Context	Examples, domain notes, links to relevant code	The agent's miss came from missing knowledge
3 — Contract	Schemas, interface definitions, acceptance tests written first	Other systems depend on the output's shape
4 — Governed capability	A shared skill, plugin or hook, with tests and an owner	Several teams need the same intent to behave identically

Each rung is justified by a specific failure observed on the rung below. You do not write a contract because the template has a contract section. You write it because a consumer broke when the output shape changed.

> Specification is earned. Every line of intent should be able to point to the miss that made it necessary, or to the risk that makes it non-negotiable.

Readiness Before Running

Progressive does not mean careless. Before the first run, an intent should pass a short readiness check:

- Is the **outcome** stated as a change in the world, not as an activity?
- Is there at least one **success criterion** that a machine or a reviewer could check without asking the author?
- Are the **non-negotiable constraints** written down, such as security, data handling and compatibility?
- Is it clear **where the evidence will come from**?

If any answer is no, fix that first. Adding detail beyond these four is optional until the evidence says otherwise.

What Progressive IDE Is Not

It is not "no spec." Some work starts at rung 3 because the risk is known: a payment integration, a regulated data flow, a public API. Progressive means you start where the risk is, not at the bottom by default.

It is not "iterate until it looks right." The loop is anchored to success criteria written before the first run. "Looks right" is not a criterion.

It is not tool-specific. The loop runs the same in Copilot's agent mode, Cursor, Claude Code or a spec-driven toolkit. With a spec-driven tool, Progressive IDE simply means you keep the generated spec as short as the evidence allows and prune what the agent invented.

A Twenty-Minute Example

A team needs a CSV export on an admin report page.

State (rung 1):

```
# Intent: Admin report CSV export

## Outcome
Admins can download the currently filtered report as CSV without
asking engineering for a data pull.

## Success criteria
- Export respects the active filters
- Column headers match the on-screen table
- 50k rows export in under 10 seconds

## Constraints
- Admin role only; no new permissions
- No PII columns beyond those already shown on screen

## Evidence
- Integration test for filter parity
- Load test result attached to the PR
```

Run. The agent adds an endpoint and a button, and writes tests.

See. The filter-parity test passes. The load test fails at 50k rows because the agent built the CSV in memory.

Refine (rung 2). Add one line of context: *"Use the existing streaming response helper in `lib/http/stream.ts`."*

Run → See. The load test passes in 4.2 seconds.

Lock. The tests stay in the suite. The intent is committed beside the feature.

The intent never grew past one page. The one line of context it gained was the one line that was actually needed.

PART II · THE SHIFT
CHAPTER 9

The Least-Infrastructure Principle

If Progressive IDE governs how much *specification* you write, the Least-Infrastructure Principle governs how much *machinery* you build around it.

> **Use the least infrastructure that reliably produces the desired outcome. Add infrastructure only when a proven intent cannot be sustained without it.**

The first edition got this backwards. It described the full machinery up front, including compiler, registry, platform, shared services and event triggers, and presented the simple version as a temporary starting point on the way to the real thing. In practice, the simple version *is* the real thing for most work, and much of the machinery was never needed.

The Infrastructure Ladder

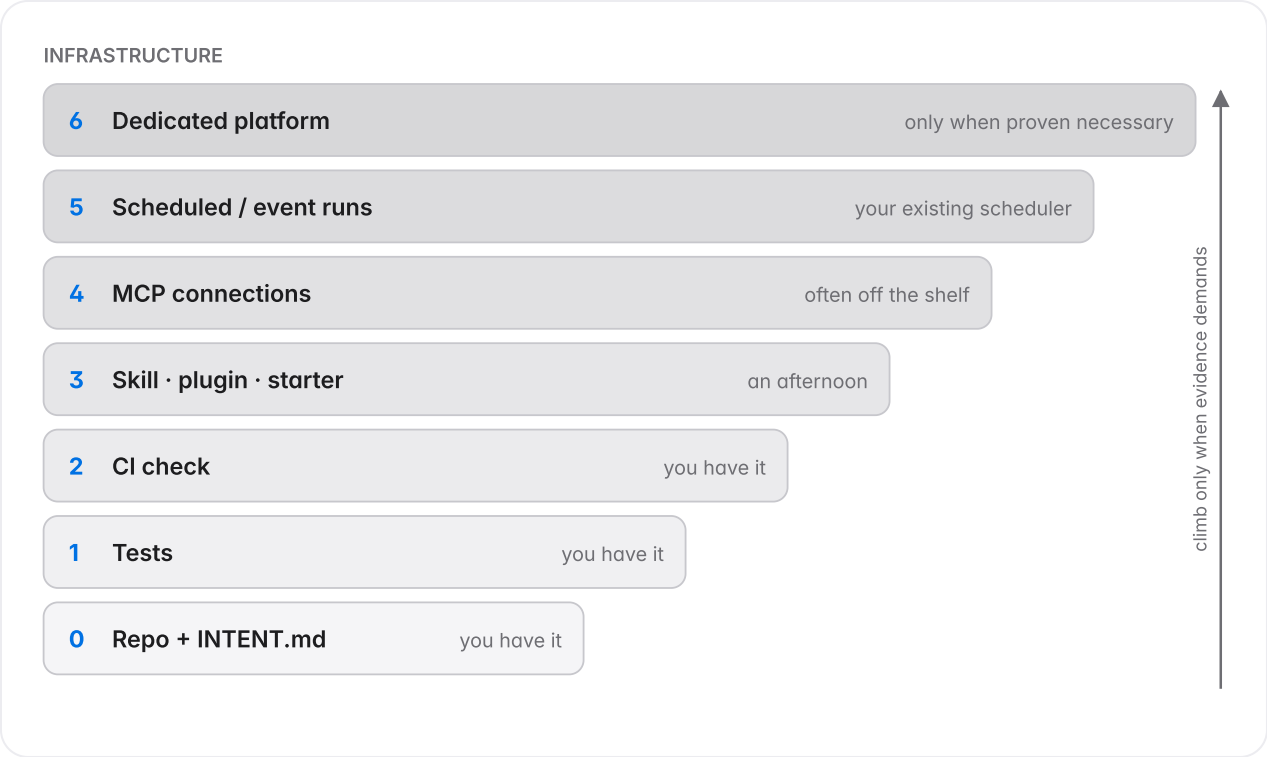


Figure 9.1 — The infrastructure ladder: each rung is earned by a proven intent

Rung	Infrastructure	You already have it?
0	A repository and an `INTENT.md`	Yes
1	Tests that encode the success criteria	Yes
2	A CI check that runs them on every change	Yes
3	A shared skill, plugin, rules file or starter repo	Usually, or one afternoon of work
4	MCP connections to the systems the intent needs	Often, increasingly off the shelf
5	Scheduled or event-triggered agent runs	Your existing scheduler or pipeline
6	A dedicated agent platform with central governance	Only if rungs 0–5 have been proven insufficient

The rule is simple: **you may not build rung *n* until an intent running on rung *n − 1* has shown, with evidence, that it needs more.**

The Questions That Earn a Rung

Before adding infrastructure, answer these honestly:

1. **Which specific intent needs this?** If you cannot name one that is already running, stop.
2. **What evidence shows the current rung is insufficient?** For example: a failure mode, a scale limit, an audit requirement that is actually being imposed.
3. **What is the smallest addition that closes that gap?**
4. **Who will own it?** Infrastructure without an owner is future drift.

Why Teams Overbuild

Teams overbuild for understandable reasons. Platforms are interesting to build. They are visible to leadership. They feel like strategy. And vendors and frameworks make it easy to stand up orchestration in a day.

But every layer has a carrying cost: upgrades, security review, on-call, onboarding and the cognitive load of one more place where behavior is defined. That cost is paid every week, and if no proven intent runs on the layer, there is nothing to offset it.

The Tools Already Did the Hard Part

A year ago, an organization that wanted agents to read intent, plan multi-step work, call tools, edit code, run tests and iterate on failures had to build much of that itself. Today it comes in the box with Copilot's coding agent, Cursor, Claude Code and the spec-driven toolkits. MCP standardized the connection to enterprise systems. Skills and plugins standardized packaging and sharing.

The work that remains is the work no tool can do for you: deciding what outcome you want, defining success, keeping the evidence and refusing to build what you have not yet proven you need.

What This Looks Like in Practice

LocationMaster, an intent-driven application built around governed skills, tools, MCP connections and hooks, was distilled within three days into `intent-driven-starter`, a reusable Claude Code plugin and starter repository. That did not take a platform. It took one working system, an honest look at which parts were reusable and the discipline to package only those parts.

The order matters. A working app came first, and a reusable capability was extracted from it. A platform, if one is ever needed, comes after many such extractions have proven their value. That is the Least-Infrastructure Principle applied to the whole lifecycle.

> Extract platforms from success. Never install them in anticipation of it.

PART II · THE SHIFT
CHAPTER 10

The Core Mental Model Shifts

Understanding Intent-Driven Engineering intellectually is quick. Practicing it is harder, because the agent is right there, it is fast, and it rewards just asking. The first edition named four mental model shifts. All four still apply, and this edition adds two.

Shift 1: From Output to Outcome

An output is what the agent produced: a diff, a document, a pull request. An outcome is what changed because of it: lead time fell, a defect class disappeared, an engineer was unblocked. Optimize for outcome reliability, not output polish. A beautiful diff that does not move the outcome is a failure that happens to look good.

Shift 2: From Individual to System

Agents make individuals dramatically more productive. That is real and insufficient. A system capability means the *team* can do a class of work at a higher level regardless of who is on keyboard that day. Individual gains become system capability when the intent is written down, shared and checked.

Shift 3: From Exploration to Engineering

Explore freely. Prototype with abandon. But know the moment a piece of work crosses from "trying things" to "people will depend on this." At that moment it needs an intent, success criteria and evidence. The first edition called this the *production gate*. The gate still exists, and it is now a file and a test, not a compilation pipeline.

Shift 4: From Knowledge in Heads to Knowledge in Repositories

Tacit knowledge used to live in senior engineers' heads. Now it also lives in private chat histories and personal rules files. Neither is shared. The shift is to put knowledge where both humans and agents will read it: the repository, in intent files, skills and tests.

Shift 5: From More Specification to the Right Specification

When a result is wrong, the instinct is to add more instructions. Resist it. Ask *which* missing piece caused the miss, and add only that. A shorter intent that is entirely load-bearing beats a longer one full of guesses, for humans and agents alike.

Shift 6: From Platform to Practice

When an initiative stalls, the instinct is to build the platform that will fix it. Resist that too. Ask whether the practice is in place: are intents written, is success defined, is evidence kept? If not, a platform will only make the absence of practice more expensive.

The TeamBrain Example

TeamBrain began as a response to the handoff problem, the gap between what sales promises and what engineering receives. The prompt-era approach would have been a better prompt for extracting requirements from sales material. The first-edition approach would have been a typed intent compiled onto a platform.

What worked was in between, and smaller than either. A short intent stated the outcome: *engineering starts every engagement with an explicit list of requirements, risks, assumptions and open questions, traceable to source material.* Success meant no requirement without a source and every assumption surfaced. The capability was packaged as a team-brain framework and a Claude Code plugin that any team could install. There was no registry and no platform, just an intent, a reusable skill and evidence from real engagements.

> TeamBrain did not get better output by writing a better prompt or a bigger spec. It removed a class of problem by stating the outcome and packaging the smallest thing that reliably produced it.

Three Practices to Start Tomorrow

1. **Write the outcome before you open the agent.** One sentence. What will be different?
2. **If you will run it twice, write it down.** An intent file beside the code, with success criteria.
3. **Never accept without evidence.** If you cannot say how you would check it, you are not ready to run it.

None of these needs a new tool, platform or budget. They need a change in what the team treats as done.

What Part II Has Established

Chapter 5 restated the root causes, including two that this book's first edition helped create. Chapter 6 distinguished intent from prompts and specs. Chapter 7 gave the second-edition definition of Intent-Driven Engineering. Chapter 8 introduced Progressive IDE, the execution model at the center of the discipline. Chapter 9 set out the Least-Infrastructure Principle. Chapter 10 named the six mental shifts that make it stick.

Part III builds the core model: how to write intent, where it lives, how it works across your tools and how to measure what it produces.

PART III

The Core Model

Writing intent, placing it, running it across your tools, and measuring what it produces

11 Defining Intent
12 Intent as a First-Class Artifact
13 Intent Across Your Tools
14 Determinism Where It Counts
15 Measuring What Actually Matters
16 The Full Loop: Intent → Agent → Evidence → Outcome

PART III · THE CORE MODEL
CHAPTER 11

Defining Intent

Part III is where the work begins. Every chapter from here is operational.

The most basic act in Intent-Driven Engineering is defining intent precisely enough that someone else, human or agent, can do the work and anyone can check it. The first edition set the bar at "precise enough that a compiler can build an execution plan." This edition sets it at "precise enough that a competent stranger could succeed and a reviewer could verify." That bar is lower in formality and higher in clarity.

The Six Fields

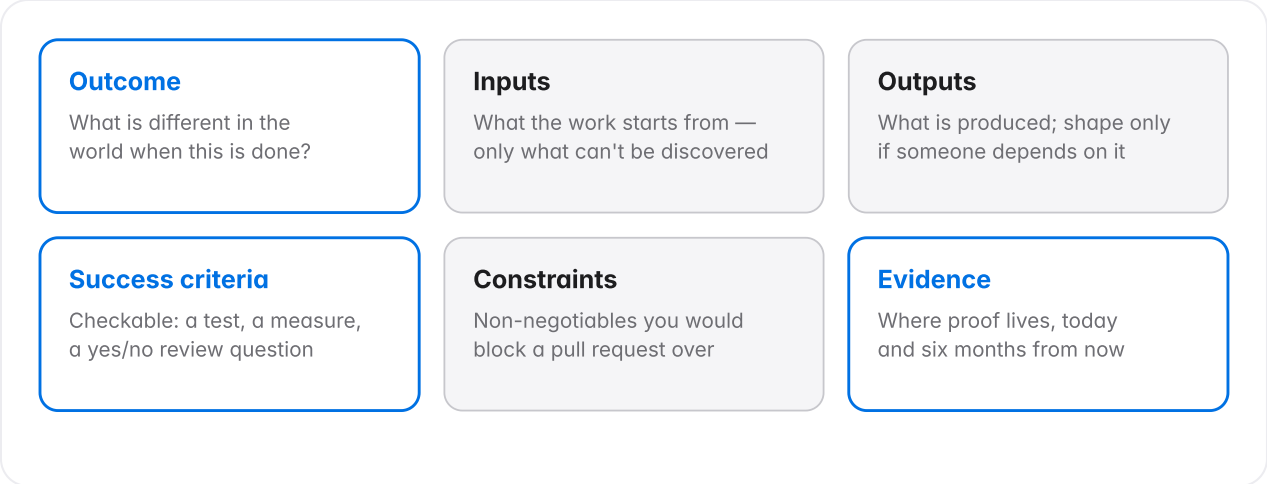


Figure 11.1 — Anatomy of an intent: six fields, each with one job

1. Outcome

What is different in the world when this is done? State it as a change in a user's ability, a system's behavior or a metric, not as an activity.

- Weak: *"Add caching to the product service."*

- Strong: *"Product detail pages render in under 300 ms at p95 under normal load."*

The weak version says what to do. The strong version says what must be true, and leaves the agent free to find out whether caching is the right way to get there.

2. Inputs

What does the work start from? Name the data, the context and the existing code it must work with. Be specific about the things the agent cannot discover on its own. If the relevant module is obvious from the repository, you do not need to list it.

3. Outputs

What is produced, and in what shape? When other systems or people consume the output, describe the shape they depend on. When nobody depends on it yet, a short description is enough. This is where the first edition's typed schemas belong, and only here, and only when a consumer exists.

4. Success Criteria

How will we know it worked? This is the field most teams under-specify and the one that matters most.

Good criteria are **checkable**: a test, a measurement or a review question with a yes/no answer.

- *"Handles errors gracefully"* is not checkable.
- *"Returns 4xx with a typed error body for every invalid input in the contract test suite"* is checkable.
- *"Is fast"* is not checkable.
- *"p95 under 300 ms in the load test profile"* is checkable.

If you cannot write a checkable criterion, you do not yet know what you want. That is useful information. Find out before you run the agent.

5. Constraints

What must not be violated? These are the non-negotiables: security, data handling, compatibility, regulatory requirements, budgets and the boundaries of what the agent may touch. Keep this list short and real. Every constraint here should be one you would block a pull request over.

6. Evidence

What will we keep that shows it worked and keeps working? Name the source: a test suite, a CI check, a dashboard, an export from an existing tool. Evidence is what lets you answer the CTO's question six months from now without reconstructing anything.

The Template

```
# Intent: <short name>

## Outcome
<What is different in the world when this is done?>

## Inputs
<What the work starts from. Only what the agent can't discover.>

## Outputs
<What is produced. Shape only if someone depends on it.>

## Success criteria
- <Checkable criterion>
- <Checkable criterion>

## Constraints
- <Non-negotiable>

## Evidence
- <Where proof will live>

## Notes (optional, added by evidence)
- <Context added after a miss, with a one-line reason>
```

The **Notes** section is where Progressive IDE does its work. It starts empty. Each line added later should say what miss it fixed. Six months on, the notes are a precise record of what the team learned about this capability.

The Sprint Retrospective Example, Revised

The first edition showed a 30-line YAML file for processing sprint retrospectives, with typed inputs, an output schema, four success criteria, an executor name and a failure handler. Here is the second-edition version:

```
# Intent: Retro → action list

## Outcome
Every retrospective ends with actions the team will actually close,
not a page of notes nobody reads.

## Inputs
- Retro transcript or notes
- Current team roster (for owners)

## Outputs
- 3–7 actions, each with an owner from the roster and a due date
- Recurring themes, if any, linked to prior retros

## Success criteria
- Every action has an owner on the roster and a due date
- No action duplicates an open action from the last two retros
- Action close rate at next retro ≥ 60%

## Constraints
- Do not assign actions to people who weren't present

## Evidence
- Action close rate, tracked in the team's board
```

It is shorter, it says more about what matters, and its final criterion is an outcome rather than a formatting rule: the actions actually get closed. The first edition's version guaranteed that every action had an owner. This version guarantees the retrospective is worth having.

> The intent is not documentation. It is the definition of done, in words a stranger can act on and a reviewer can check.

PART III · THE CORE MODEL
CHAPTER 12

Intent as a First-Class Artifact

A first-class artifact is one the engineering system treats with full professional discipline: version control, review, testing and ownership. Source code is first-class, and so are infrastructure definitions. Intent must be too. It should not be a wiki page or a comment in a ticket. It should be a file in the repository, reviewed like code.

Where Intent Lives

The first edition placed intents in a central registry. This edition places them **next to the code they govern.**

```
payments-service/
├── AGENTS.md                  # points tools at intents and conventions
├── INTENT.md                  # service-level intent
├── features/
│   ├── refund-automation/
│   │   ├── INTENT.md          # feature intent
│   │   └── ...
│   └── fraud-prescreen/
│       ├── INTENT.md
│       └── ...
└── tests/
    └── intent/                # success criteria as tests
```

A service-level `INTENT.md` states what the service is for and the constraints every change must respect. Feature-level intents state what each piece of work must achieve. Tests under `tests/intent/` encode the success criteria.

The reason is practical. Agents read the repository. Humans review the repository. CI runs against the repository. An intent anywhere else is invisible to all three.

The Lifecycle, Lightened

The first edition described six stages: author, review, version, compile, deploy, measure and evolve. With the compiler and platform removed, five remain, and each is something your team already does.

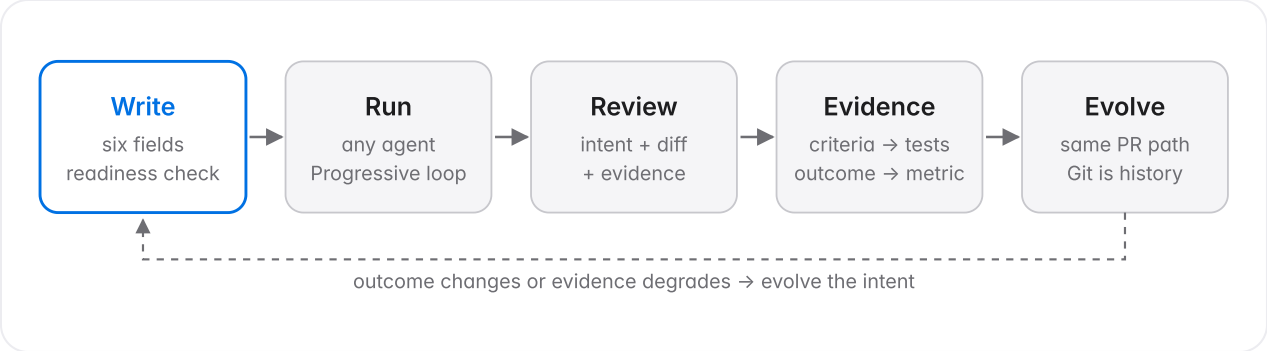


Figure 12.1 — The intent lifecycle: five stages, all in the tools you already use

1. Write. Answer the six questions. Pass the readiness check from Chapter 8. This should take minutes for most work and an hour for significant work. If it takes longer, the scope is too broad.

2. Run. Hand the intent to the agent. Let it propose an approach. Iterate with the Progressive IDE loop.

3. Review. The pull request contains the intent, the change and the evidence. The reviewer checks three things. Is the intent clear? Does the change satisfy it? Does the evidence prove it? Reviewing against a written intent is much faster than reviewing against an unwritten standard, because the reviewer is verifying instead of discovering.

4. Evidence. Success criteria become tests and checks that stay in the suite. Outcome metrics are tied to an existing source.

5. Evolve. When the outcome changes, the evidence degrades or a new miss appears, the intent is updated through the same review path. Git keeps the history.

Versioning Without Ceremony

The first edition prescribed semantic versioning for every intent. Most intents do not need it. They are consumed by one team and evolve with the code, and Git history is the version record.

Semantic versioning earns its place at rung 3 and above, when an intent is packaged as a shared skill or plugin that other teams install. Then the version number tells consumers whether an update is safe. Until then, a commit hash is a version.

Ownership

Every intent has an owner: the team accountable for whether its outcome holds. The simplest mechanism is the one you already use. A `CODEOWNERS` entry for the intent file means changes to it require the owner's review. No registry entry is required.

Ownership means three responsibilities:

1. **Accuracy.** The intent describes what the work is actually supposed to do.
2. **Evidence.** Someone looks at the evidence and acts when it degrades.
3. **Evolution.** When the business need changes, the intent changes first.

The Compounding Effect

In month one, a team has a handful of intent files. They are a useful reference. By month six it has dozens, scattered through the repositories where the work happens, each with its evidence and its notes. New engineers and new agents read them and understand what each part of the system is for and how its success is judged.

This is the organizational capital the first edition attributed to the registry. It turns out the registry was never the asset. **The intents were the asset**, and they are most valuable where the work happens.

> An intent file that lives beside the code is read by every agent, every reviewer and every new engineer who touches that code. An intent in a central catalog is read by whoever remembers to look.

PART III · THE CORE MODEL
CHAPTER 13

Intent Across Your Tools

Every team in your organization already uses an agent, and they probably do not all use the same one. This chapter shows how one intent serves all of them without being rewritten for each.

The Principle: Write Once, Point Everywhere

Each tool has its own instruction mechanism. Do not restate your intents and conventions in each one. Put them in shared, tool-neutral files and make each tool-specific file a short pointer.

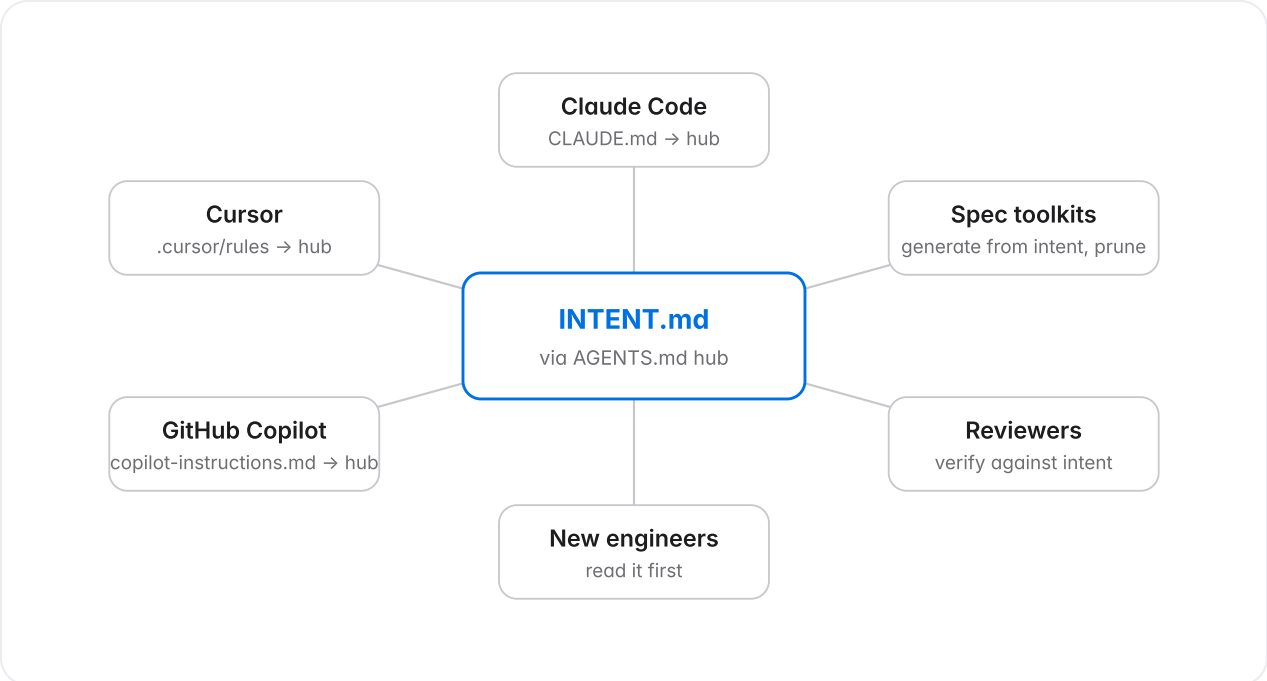


Figure 13.1 — One intent, many tools: tool-specific files point to shared intent

AGENTS.md as the Hub

`AGENTS.md` has become a widely recognized cross-tool convention for agent instructions at the repository root. Use it as the hub:

```
# AGENTS.md

## How work is defined here
Every non-trivial change starts from an INTENT.md (service root or
feature folder). Read the nearest INTENT.md before planning.
Success criteria in the intent are the definition of done.

## Conventions
- Tests for success criteria live in tests/intent/
- Run `make check` before proposing a change
- Do not add dependencies without noting why in the PR

## Constraints that apply to every change
- No secrets in code or logs
- Public API changes require a contract test update
```

Tool-Specific Pointers

Keep each tool's file to the few lines that tool actually needs.

GitHub Copilot (`.github/copilot-instructions.md`):

```
Follow AGENTS.md at the repository root. Before making changes,
read the nearest INTENT.md and treat its success criteria as the
definition of done.
```

Cursor (`.cursor/rules/intent.mdc`):

```
---
description: Intent-driven workflow
alwaysApply: true
---
Follow AGENTS.md. Read the nearest INTENT.md before planning any
change. Do not mark work complete until its success criteria pass.
```

Claude Code (`CLAUDE.md`):

```
@AGENTS.md
Read the nearest INTENT.md before planning. Use the repo's
intent skill to check readiness before starting significant work.
```

The details of these mechanisms will change as the tools evolve. The pattern will not: **one shared source, thin pointers.**

Spec-Driven Toolkits

Spec-driven tools such as GitHub Spec Kit and Kiro generate requirements, design and task documents for each feature. They work well with Intent-Driven Engineering when the intent is upstream of the spec:

1. Write the `INTENT.md` first, with outcome, constraints and success criteria.
2. Let the toolkit generate its spec and plan *from* the intent.
3. **Prune.** Delete anything in the generated spec that is an unnecessary guess, such as file names, library choices or structure you don't care about. Keep what the intent requires.
4. Treat the intent's success criteria as authoritative when the generated spec and the intent disagree.

This keeps the toolkit's planning ability and stops spec bloat. The spec becomes a disposable plan. The intent remains the durable contract.

Skills and Plugins

When the same kind of intent recurs across repositories, package the *how*, not just the *what*, as a skill or plugin. Examples are a readiness check that scores an intent, a review routine that checks a diff against its intent, or a scaffold that creates a new feature folder with an intent template. Claude Code skills and plugins, Cursor rules and Copilot prompt files are all ways to distribute this. Chapter 18 covers when packaging is worth it.

What Not to Do

- **Don't maintain parallel instruction sets.** Three files saying the same thing guarantees two will be wrong.
- **Don't encode tool quirks in the intent.** If the intent says "use Cursor's composer to...," it has stopped being an intent.
- **Don't standardize on one tool to avoid this problem.** You will lose the engineers who work best in another, and the next tool will arrive anyway.

PART III · THE CORE MODEL
CHAPTER 14

Determinism Where It Counts

The first edition's most ambitious claim was that compiled intents made AI execution *deterministic*: the same inputs would produce the same class of outputs, through the same validated path, every time. It placed that determinism in four enforcement points inside the platform: input validation, execution plan adherence, output schema enforcement and success criteria evaluation.

The insight was right. Determinism is a property of the system around the model, not of the model. The location was wrong. You do not need a platform to enforce it. You need checks in the places where your work already passes.

Where Determinism Lives Now

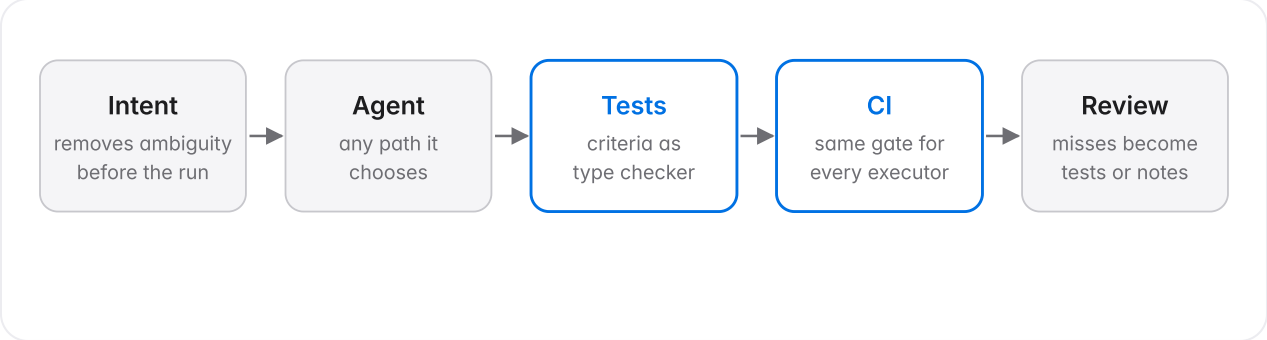


Figure 14.1 — Determinism lives in checks, not in the model

Point 1: The intent. Ambiguity is the first source of variance. A clear outcome, explicit constraints and checkable criteria remove most of the variance before the agent starts.

Point 2: Tests. The success criteria, encoded as tests, are the type checker for AI-assisted work. The agent can take any path it likes, but it cannot get a change accepted without passing them.

Point 3: CI checks. Linters, contract tests, security scans and any intent-specific checks run on every change, whoever or whatever authored it. This is the old "execution plan adherence" made universal. It is the same gate for every executor.

Point 4: Review. A human reviewing against a written intent catches the misses tests did not anticipate, and each miss becomes a new test or a new line in the intent's Notes. The review is the feedback loop.

Point 5: Runtime contracts (when needed). When AI output is consumed at runtime by another system, for example an agent producing structured data for a service, validate the output against a schema at the boundary. This is the one place the first edition's output schema enforcement belongs, and it is a few lines of validation code, not a platform.

Prompt Drift Is Now Change Drift

The first edition warned of *prompt drift*: the slow, invisible degradation of a prompt-based workflow iterated without version control. With agents, the risk becomes *change drift*. Each agent-authored change is reasonable on its own, but the system slowly moves away from its intent because nobody checks the changes against it.

The defense is the same as before, and simpler: **every change is checked against the intent's tests in CI.** Drift that breaks a success criterion fails the build. Drift that doesn't break any criterion was either harmless or a sign that a criterion is missing. Add it.

Determinism Under Pressure

At 2 a.m. during an incident, nobody writes a careful prompt. An engineer who can point an agent at a well-written intent, such as an incident-triage intent with its success criteria and constraints, gets the same quality of first-pass analysis at 2 a.m. as at 2 p.m. The checks run either way.

> Reliability is not a property of the model. It is a property of the checks that every change must pass. Put the checks in the path, and any competent executor becomes reliable enough.

PART III · THE CORE MODEL
CHAPTER 15

Measuring What Actually Matters

Measurement separates engineering from craft. The first edition built a three-tier measurement architecture: execution metrics, quality metrics and business metrics, generated automatically by the platform for every intent. The tiers were the right questions. The automatic generation assumed a platform most teams never built.

This edition keeps the three questions and attaches them to evidence you already collect.

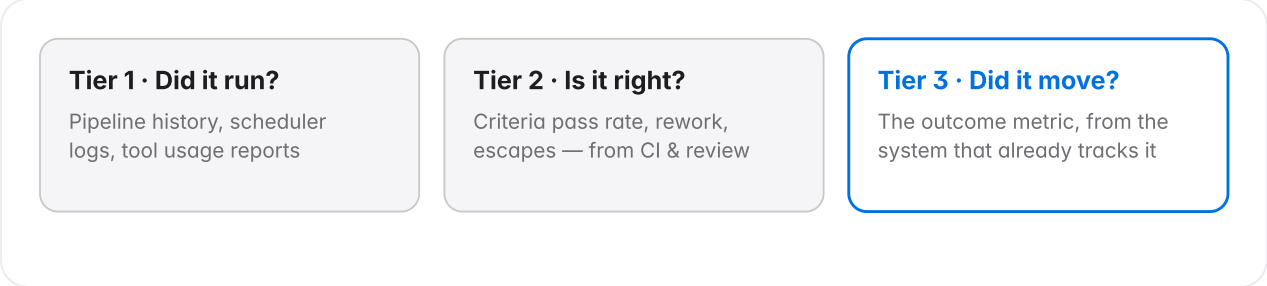


Figure 15.1 — Three questions, three sources of evidence you already have

Tier 1: Did It Run?

Is the work happening reliably? For a one-off feature, this is "did it ship." For a recurring agent task, such as nightly triage, PR review preparation or dependency updates, it is whether the job ran, how long it took and what it cost.

Evidence source: CI and pipeline history, scheduler logs, the agent tool's own usage reporting.

Tier 2: Is It Right?

Did the work meet its success criteria? This is the tier that turns AI output into engineering.

- **Criteria pass rate:** the share of runs or changes that pass the intent's checks the first time.

- **Rework rate:** the share of agent-authored changes needing significant human revision before merge.
- **Escape rate:** problems found after merge that the criteria should have caught.

Every escape is a signal to add a criterion. Every high rework rate is a signal that the intent is missing context. These are Progressive IDE's refinement triggers, made visible.

Evidence source: CI results, PR review data, defect tracking.

Tier 3: Did It Move the Needle?

Did the outcome named in the intent actually change? This is the tier leadership asks about, and it cannot be generated automatically by any tool. It requires that the intent's **Outcome** field names a metric and its **Evidence** field names where that metric lives.

- **Lead time** from ticket to production
- **Review cycle time**
- **MTTR**
- **Time to first merged change** for new engineers
- **Cost per outcome**: what it costs to produce the result, not what it costs to run the agent

Evidence source: the systems that already track these: delivery analytics, incident management, product analytics, finance.

Baselines First

No improvement claim survives without a baseline. Before the first intent runs, capture the current value of its Tier 3 metric. That is one number, recorded once, in the intent's Evidence section. Teams that skip this step spend the ultimatum meeting arguing about estimates.

The Metric That Tells You the Discipline Is Working

The first edition chose *criteria pass rate trend* as the single most important metric. In a Progressive IDE practice, the companion metric is **intent growth versus pass rate**. A healthy capability shows its pass rate rising while its intent grows only slowly. Each addition to the intent fixes a real miss, and the misses become rarer.

If the intent grows quickly while the pass rate stays flat, the team is adding specification without learning. That is spec bloat, and it shows up in the numbers.

> Measurement is not an add-on to Intent-Driven Engineering. It is why the discipline compounds. Without evidence there is no improvement, without improvement there is no trust, and without trust there is no scale.

PART III · THE CORE MODEL
CHAPTER 16

The Full Loop: Intent → Agent → Evidence → Outcome

This is where it all connects. One team, one capability, the full loop, using nothing the team didn't already have.

The Scenario

An app team runs a B2B product. New accounts get a personalized onboarding checklist based on the user's role, organization size and product tier. Today a static checklist is shown to everyone, and activation is weak. The team wants a generated, personalized checklist. *(Illustrative scenario.)*

Stage 1: State the Intent (Rung 1)

The engineer opens the repository, not a chat window, and creates `features/onboarding-checklist/INTENT.md`:

```
# Intent: Personalized onboarding checklist

## Outcome
New users reach first value faster. Baseline median
time-to-first-value: 3.2 days (product analytics, last 90 days).

## Inputs
- User role, org size, product tier (from signup)
- Current feature catalog (features/catalog.json)

## Outputs
- 8–12 checklist items: title, why it matters, est. minutes, category
- Rendered in the existing onboarding panel

## Success criteria
- Items span ≥ 3 categories
- Every item links to a feature that exists in the catalog
- Generation completes in < 2 s at p95
- Median time-to-first-value improves ≥ 15% within 6 weeks

## Constraints
- No PII sent to the model beyond role/size/tier
- Falls back to the static checklist on any failure

## Evidence
- Contract tests in tests/intent/onboarding_checklist.test.ts
- Time-to-first-value dashboard (product analytics)
```

Writing it took twenty-five minutes. Most of that time went to the baseline and the constraint about PII. Those are the two things no agent could have decided for her.

Stage 2: Run

She opens the agent in her editor and points it at the intent. It proposes an approach: a generation function, a catalog lookup, a fallback path and tests for each criterion. She accepts the plan with one change: use the team's existing model-client wrapper rather than a new SDK dependency.

Stage 3: See

The tests pass except one. Two checklist items for the "analyst" role link to a reporting feature that was renamed last quarter. The catalog file is current, but the agent also drew on examples in an old test fixture.

Stage 4: Refine (Rung 2)

She adds one line to the intent's Notes:

```
## Notes
- Catalog is the only source of feature names; ignore fixtures.
  (Added after renamed-feature links in first run.)
```

She runs it again. Every criterion passes.

Stage 5: Review and Lock

The pull request contains the intent, the change and the test results. The reviewer reads the intent first, then checks the diff against it. He raises one issue: the fallback path is tested, but there is no alert when the fallback rate rises. A one-line metric emission is added, and a CI check now asserts it exists. The change merges.

Stage 6: Measure

In week one, generation p95 is 1.3 seconds and the fallback rate is 0.4%. In week three, a pattern appears: enterprise admins get checklists heavy on end-user features and light on admin setup. The category criterion passes, but the checklist is not useful to them. That is a criterion that was too weak.

Stage 7: Evolve

The intent gains one criterion: *"For admin roles, ≥ 3 items are in the 'setup' category."* A test is added. Two weeks later, at week six, median time-to-first-value is 2.5 days, a 22% improvement on the baseline. The number came from the same analytics dashboard that recorded the baseline.

What the Loop Produced

The team ended up with one intent file of about forty lines, two Notes entries, five tests, one metric and one outcome that moved by a measured amount. There was no registry, compiler, platform or new infrastructure.

The intent is now in the repository where every future agent and engineer working on onboarding will read it. Its Notes record exactly what the team learned. Its Evidence section answers the CTO's question before she asks it.

> Every team that runs the full loop once understands why they won't go back. It is not a methodology. It is a habit that makes everything it touches more reliable, more visible and easier to improve.

PART IV

Reuse Without a Registry

What happened to the registry, and how reuse actually travels

17 What Happened to the Registry
18 Skills, Plugins and Starter Repos
19 Turning Knowledge into Systems
20 Intent as Code

PART IV · REUSE WITHOUT A REGISTRY
CHAPTER 17

What Happened to the Registry

The first edition gave the intent registry a starring role. It was organizational memory, onboarding infrastructure, the coordination layer between teams, the executive view for the 60-day ultimatum and the foundation of a "6.2× reuse factor." Whole chapters assumed it.

It never panned out. This chapter explains why, honestly, because the reasons teach more than the original idea did.

What the Registry Promised

The vision was a searchable catalog of every AI capability in the organization, each entry with a name, version, owner, schema, execution history and outcome metric. Engineers would search it before building anything. Teams would consume each other's capabilities through it. Leadership would see the whole AI portfolio in one view.

Why It Didn't Work

Nobody browses catalogs. Engineers look for things where they already work: in the repository, in the editor and, increasingly, by asking the agent. A separate catalog, however searchable, sits outside that path. Usage dropped off after the launch announcement.

The maintenance tax was paid by the wrong people. Every registry entry duplicated information that lived somewhere else: the intent in the repo, the owner in CODEOWNERS, the metrics in the observability stack. Keeping the copy in sync was nobody's real job, so it fell out of sync, and a stale registry is worse than none because people stop trusting it.

Reuse needed more than discovery. Finding another team's intent did not make it reusable. Reuse needed the intent *plus* its context, its tests and a way to install it. A catalog entry provided none of those.

The agents made search cheap. Once an agent could search every repository in the organization in seconds and read the intent files directly, a curated index added little.

It assumed a platform. The registry was designed as the front door to a compiled-intent platform. Where the platform wasn't built, the registry had nothing behind it.

What Survived

Three ideas from the registry were right and are kept in this edition.

1. **Intents should be discoverable.** That is now done by keeping them in predictable places (`INTENT.md` beside the code) so both humans and agents find them by convention.
2. **Ownership should be explicit.** That is now done with the CODEOWNERS file you already have.
3. **Leadership needs a portfolio view.** That is now done with a lightweight index, described next.

The Rethink: The Repo Is the Registry

The replacement is deliberately small. Each repository already contains its intents. For the portfolio view, a single generated file is enough.

```
# INTENTS.md — generated weekly, do not edit by hand

| Intent | Path | Owner | Outcome metric | Baseline → Now |
|---|---|---|---|---|
| PR review prep | platform/pr-review/INTENT.md | @platform | Review cycle (h) | 6.4 → 2.1
|
| Incident triage | sre/triage/INTENT.md | @sre | MTTR (min) | 47 → 31 |
| Onboarding checklist | app/onboarding/INTENT.md | @growth | TTFV (days) | 3.2 → 2.5 |
```

Illustrative example.

A scheduled job, or a scheduled agent run, walks the organization's repositories, finds intent files, reads their Outcome and Evidence sections and regenerates the table. It needs no database or service and does no manual curation. If an intent has no evidence, the empty cell is visible, and that visibility does more for discipline than any registry policy did.

> The registry was never the asset. The intents were. Keep them where the work happens, and generate the index from them, not the other way round.

When a Real Catalog Makes Sense

Some organizations do need more: a regulated environment that requires a formal inventory of AI capabilities, or a marketplace of shared capabilities used by hundreds of teams. Even then, build the catalog *from* the repositories. The source of truth stays where the work happens. The catalog is a view.

PART IV · REUSE WITHOUT A REGISTRY
CHAPTER 18

Skills, Plugins and Starter Repos

If the registry was not how reuse traveled, what was? The answer is in units that carry their own context and install themselves.

The Units of Reuse That Actually Travel

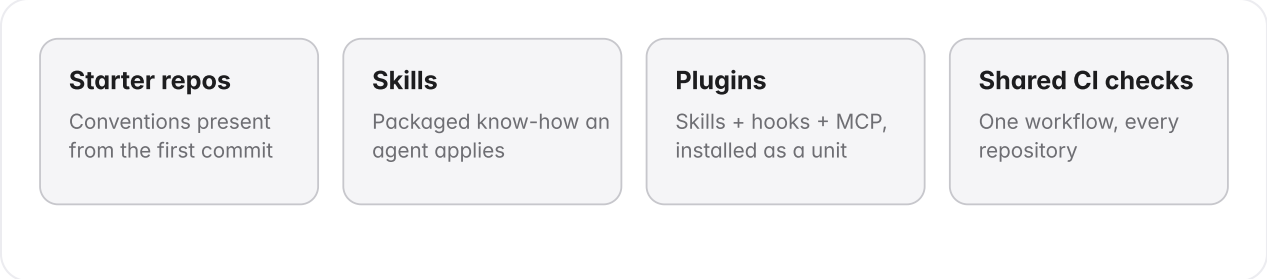


Figure 18.1 — Units of reuse that travel: each carries its own context and installs itself

Starter repositories. A repository that new work begins from. It already contains the intent template, the `AGENTS.md` hub, the tool pointer files, the CI checks and a worked example. Reuse happens at creation time, which is the moment it is cheapest.

Skills. A packaged routine that an agent can apply, such as checking an intent's readiness, reviewing a diff against its intent or turning a story into an intent. Skills carry their own instructions, so the receiving team does not need to understand how they were built.

Plugins. A bundle of skills, commands, hooks and MCP connections installed as a unit. Plugins are how a governed capability, with its checks and guardrails, is distributed across many repositories at once.

Shared CI checks. A reusable workflow that any repository can include, such as an intent-readiness lint or a check that every changed feature folder contains an `INTENT.md`.

Why These Work When the Registry Didn't

	Registry entry	Starter / skill / plugin
Found by	Searching a separate catalog	Being the default, or being installed
Carries context	Metadata only	Instructions, tests, examples
Installs	No	Yes
Stays current	Manual sync	Versioned with its source
Read by agents	Rarely	Always, because it is in the repo

The intent-driven-starter Story

LocationMaster was built as an intent-driven application with a governed architecture of skills, tools, MCP connections and hooks. It worked. The question was which parts of it were reusable.

Within three days, the reusable parts were distilled into `intent-driven-starter`, a starter repository and Claude Code plugin. New projects begin from it and inherit its intent template, its governance hooks, its readiness checks and its conventions on the first commit.

That is the pattern to copy:

1. **Build one real thing.** Not a framework, but a working system with an outcome.
2. **Identify what was reusable.** Usually it is the conventions, checks and routines, not the business logic.
3. **Package only that.** Into a starter, a skill or a plugin.
4. **Make it the default.** New work starts from it, so nobody has to remember to search.

Versioning Shared Capabilities

This is where the first edition's versioning discipline belongs. A skill or plugin used by many teams needs semantic versions, a changelog and a clear statement of what a major version break means. Consumers pin a version and upgrade deliberately. It is ordinary package management, applied to the units that are actually shared.

When to Package

Package when the same intent pattern has succeeded at least twice in different places and a third team is about to need it. Packaging earlier means guessing what is reusable. Packaging later means three diverging copies to reconcile.

> Reuse does not happen because people search. It happens because the right thing is already there when they start.

PART IV · REUSE WITHOUT A REGISTRY
CHAPTER 19

Turning Knowledge into Systems

Every organization holds knowledge that is not in its systems. It sits in the senior engineer who knows which deployments are risky, the support lead who recognizes an error pattern on sight, and the architect who knows why the service boundaries are where they are. That knowledge is valuable and fragile. It walks out the door, it doesn't transfer in handoffs and no agent can use it.

Intent-Driven Engineering turns that knowledge into systems by writing it where it will be *executed*, not just consulted.

Documentation Is Read. Intent Is Run.

Documentation captures what someone knew when they wrote it, and it waits to be found. An intent, a skill or a check puts knowledge in the path of the work. When the senior engineer's judgment about risky deployments becomes a deployment-risk intent with success criteria and a CI check, it applies to every deployment, including the ones she never sees.

The First Edition's Intent API, Reconsidered

The first edition proposed that every compiled intent automatically become a REST endpoint on the agentic platform, so any system could call any intent. For a small number of capabilities that are genuinely shared services, such as a fraud prescreen or a document classifier, an API is right. Build it as you would any service, with the intent as its contract.

For most knowledge, an API is the wrong shape. Knowledge about how to review, triage, scaffold or assess is best delivered *to the agent doing the work*, as a skill or an intent it reads, not as an endpoint someone has to call.

The Deployment-Risk Example, Revised

A senior reliability engineer has watched four years of incidents. She knows which kinds of changes go wrong. Here is her knowledge as an intent:

```
# Intent: Pre-deploy risk check

## Outcome
High-risk deployments are caught before production, not in the
post-incident review.

## Inputs
- The PR diff and deployment manifest
- Changes merged to the same services in the last 24 h
- Current error rate and latency for touched services (via MCP)

## Outputs
- Risk level (low / elevated / high) with the top 3 reasons
- Suggested watch metrics for the rollout

## Success criteria
- Every "high" names a specific, verifiable reason
- Over a quarter, ≥ 80% of deploy-related incidents were flagged
  elevated or high beforehand

## Constraints
- Advisory only; never blocks an emergency deploy
- No production data leaves the approved MCP connection

## Evidence
- Incident review tags (flagged / not flagged), reviewed monthly

## Notes
- Schema migrations + traffic peak = high, always. (Her rule.)
- Config-only changes to the edge proxy have caused 3 of the
  last 10 incidents; treat as elevated. (Added after Q2 review.)
```

The Notes section is the knowledge. Her rules are written down, attributed and placed where an agent reads them on every deployment. Each quarter's incident review either confirms them or adds a new one. The intent records not just what she knew but what the system has learned since.

TeamBrain as a Knowledge System

TeamBrain applies the same pattern to a different kind of knowledge: what was promised in sales conversations. Its intent turns transcripts, decks and emails into an engineering-ready brain of requirements, risks, assumptions and open questions, each traced to its source.

Packaged as a team-brain plugin, it travels to every new engagement.

> Documentation captures what someone knew at the moment they wrote it. An intent captures what the system has learned from every run since.

PART IV · REUSE WITHOUT A REGISTRY
CHAPTER 20

Intent as Code

Infrastructure as Code changed how the industry thought about servers. Before it, servers were configured by hand and drifted apart. After it, configuration was version-controlled, reviewed, tested and reproducible.

Intent as Code is the same shift for AI-assisted work. Before it, what the agent was supposed to achieve lived in chat sessions and drifted from person to person. After it, intent is version-controlled, reviewed, tested and reproducible.

The Five Properties, Revisited

1. Version control. Intent files live in Git. Every change is attributed, explained and reversible. When an outcome degrades, `git log` on the intent shows what changed.

2. Review. Intent changes go through pull requests. Reviewers ask whether the outcome is right, whether the criteria are checkable and whether a constraint is missing. A change to an intent is a change to the definition of done, and it deserves the same scrutiny as a change to code.

3. Automated testing. Success criteria are tests. They run in CI on every change. A change that breaks a criterion does not merge, whoever wrote it.

4. Dependency management. Where intents depend on shared skills or plugins, those dependencies are declared and pinned like any package. The first edition's executor registry is replaced by the package manager you already use.

5. Evidence as a code property. The intent's Evidence section names where proof lives. A scheduled job can verify that every intent's evidence source exists and has recent data, which turns "do we have evidence?" into a CI check.

The GitOps Connection, Kept Simple

GitOps declares desired state in Git and lets automation reconcile reality to it. Intent as Code applies the same idea at human scale. The desired outcome is declared in Git. CI and review reconcile each change against it. Evidence shows whether production matches.

There is no reconciliation controller and no platform. It is the same pull-request workflow your team already runs, with one more file that defines what done means.

> Infrastructure as Code transformed how we manage servers. Intent as Code transforms how we manage AI-assisted work. The discipline is the same, and the tooling is already on your laptop.

PART V

Just Enough Platform

When infrastructure earns its place, and how little of it you need

21 When You Actually Need a Platform
22 Agents, Skills, Tools and Systems
23 Orchestration: Start Sequential
24 MCP: Connect, Don't Build
25 Governance Without Bureaucracy

PART V · JUST ENOUGH PLATFORM
CHAPTER 21

When You Actually Need a Platform

The first edition devoted Part V to WEGA, a five-layer agentic platform with an intent API, orchestration, an executor pool and shared services. Wipro's WEGA work is real, and for some enterprise contexts a platform of that kind is the right answer. But the first edition presented it as the destination for every organization, and that was the error.

This chapter reverses the default. **You do not need a platform until you can prove you do.** Here is how to tell.

Signals That You Might

These are real reasons, each tied to a running intent that cannot be sustained without more infrastructure:

- **Unattended execution at volume.** An intent must run thousands of times a day, triggered by events, with no human in the loop, and your existing pipelines cannot carry it.
- **Cross-team runtime sharing.** Several teams call the *same* capability at runtime, not just reuse the same pattern, and need a shared, versioned service.
- **Imposed governance.** An auditor, regulator or contract requires a central inventory, central policy enforcement or an immutable execution record that your current tools cannot produce.
- **Cost or model routing at scale.** Spend has become significant enough that routing between models, caching and centralized budgets pay for themselves.

Signals That You Don't (Yet)

- "We'll need it eventually."
- "Other companies have one."
- "It will make adoption easier."

- "The vendor demo looked great."
- You cannot name three intents in production that would run on it on day one.

The Extraction Path

When a platform is justified, build it by extraction, not installation:

1. Start from the intents that are already running and the pain they are already causing.
2. Build the smallest shared service that removes that pain for those intents.
3. Move one intent onto it. Measure. Then the next.
4. Only add capabilities, such as orchestration, routing or policy engines, when a running intent needs them.

A platform built this way is smaller than the one you would have designed up front, it runs real work from its first day, and every component has a user.

Where WEGA-Class Platforms Fit

Enterprise platforms like WEGA earn their place in large, regulated, multi-team organizations where the signals above are present and the first rungs of the infrastructure ladder have been proven insufficient. In those settings, a platform that provides central governance, audit, cost tracking and a governed runtime can be the right investment. The discipline is unchanged. The intents, success criteria and evidence come first. The platform runs them.

> A platform is a consequence of proven intents, not a prerequisite for them.

PART V · JUST ENOUGH PLATFORM
CHAPTER 22

Agents, Skills, Tools and Systems

When everything is called an agent, the word loses meaning and architectural decisions get muddled. The first edition offered a three-layer taxonomy of tools, agents and systems. It still holds. This edition adds two layers that became standard in the meantime: skills and hooks.

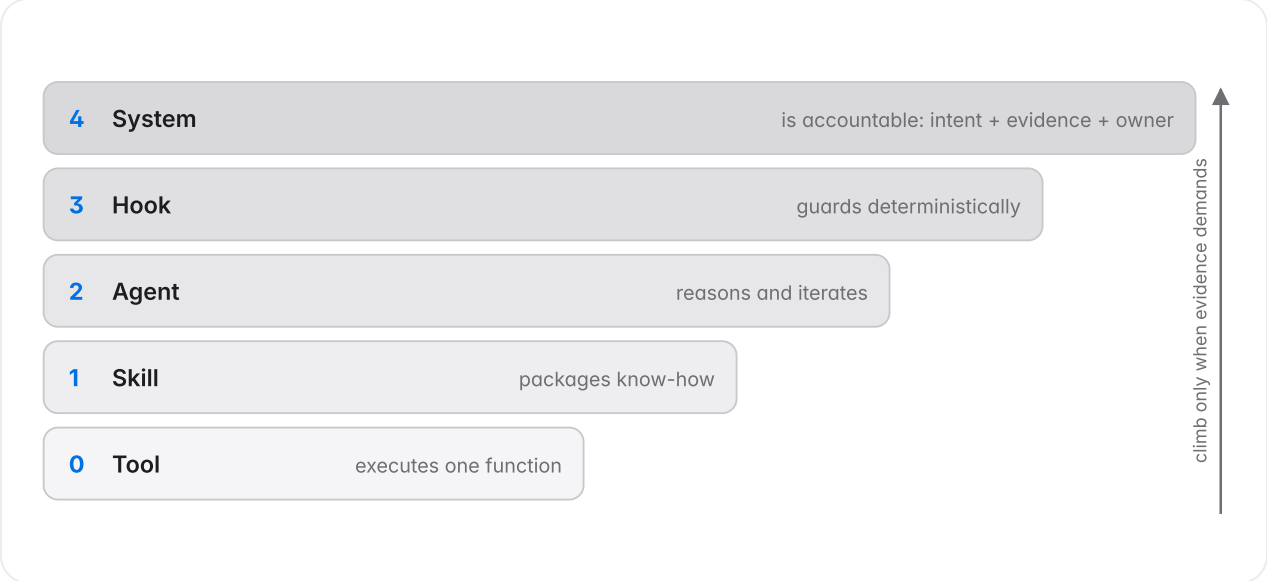


Figure 22.1 — The taxonomy: tools execute, skills package know-how, agents reason, hooks guard, systems are accountable

Tools

A tool is a single-function capability with a defined input and output: run the tests, query the database, fetch the ticket, search the code. It makes no decisions. In practice most tools now arrive through MCP servers or are built into the agent.

Skills

A skill is packaged know-how: instructions, and sometimes scripts and references, that tell an agent how to do a particular kind of task well. Examples are "check this intent for readiness," "review this diff against its intent" and "scaffold a new feature folder." Skills are the main unit of reuse for *how* work is done.

Agents

An agent is a reasoning loop that plans, calls tools, applies skills, observes results and iterates toward a goal. Copilot's agent mode and coding agent, Cursor's agent and Claude Code are agents. The agent is the executor, and in second-edition terms, the compiler.

The first edition insisted agents have *bounded* state, with no memory carried between executions. That principle still matters for automated, unattended runs, where each run should start from the same baseline. For interactive work, memory and project context are useful, as long as anything durable is written into the repository, where it can be reviewed, rather than left in the agent's private memory.

Hooks

A hook is a deterministic guard that runs at a defined point in the agent's loop, for example before a tool call, after a file edit or before a commit. Hooks enforce constraints the intent marks as non-negotiable: block writes outside the allowed paths, require tests to pass before a commit, redact secrets. They are the lightweight successor to the first edition's platform-level policy gates.

Systems

A system is accountable. It has an intent, success criteria, evidence and an owner. A system may be a single agent run guided by an `INTENT.md`, or a scheduled pipeline that calls several skills. What makes it a system is not its complexity but its accountability.

> Call it what it is. A tool executes. A skill teaches. An agent reasons. A hook guards. A system is accountable. Governance follows the taxonomy.

Why the Distinction Matters

Each layer is tested and governed differently.

Layer	How it's tested	Who owns it
Tool	Functional tests, uptime	The system it exposes
Skill	Example runs against known inputs	The team that packaged it
Agent	Vendor's responsibility, plus your evals for your use	Vendor / platform
Hook	Unit tests: does it block what it should?	Platform or security
System	The intent's success criteria and evidence	The intent's owner

The common mistake is building a sophisticated agent workflow and calling it a system without the intent, criteria and evidence. It has the capability without the accountability.

PART V · JUST ENOUGH PLATFORM
CHAPTER 23

Orchestration: Start Sequential

The first edition presented three orchestration patterns, sequential, parallel and conditional, as compilation-time declarations in the intent file. The patterns are still useful. What changed is who does the orchestrating: **in most work, the agent does it.**

The Default Is One Agent, One Intent

Most intent-driven work is one agent working toward one intent: it plans, edits, tests and iterates. That is sequential orchestration, handled by the agent's own loop. You don't need to declare it, design it or build it.

When to Orchestrate Explicitly

Explicit orchestration, where you define the steps instead of letting the agent choose, earns its place in three situations.

Sequential, for unattended pipelines. A scheduled or event-triggered run that must follow the same steps every time for auditability, such as nightly dependency review: gather the updates, assess each one, open pull requests for the safe ones and report the rest. Define the steps in your pipeline tool and let an agent perform each one.

Parallel, for independent analyses. Several independent reviews of the same input, such as security, performance and accessibility on one pull request, can run as separate agent tasks or subagents and be merged. Use it when the latency saving or the separation of concerns is measurable.

Conditional, for routing by complexity. A classifier step routes simple cases to a lightweight path and complex ones to a thorough path. Support triage is the classic example. Use it when the cost or latency difference between the paths matters.

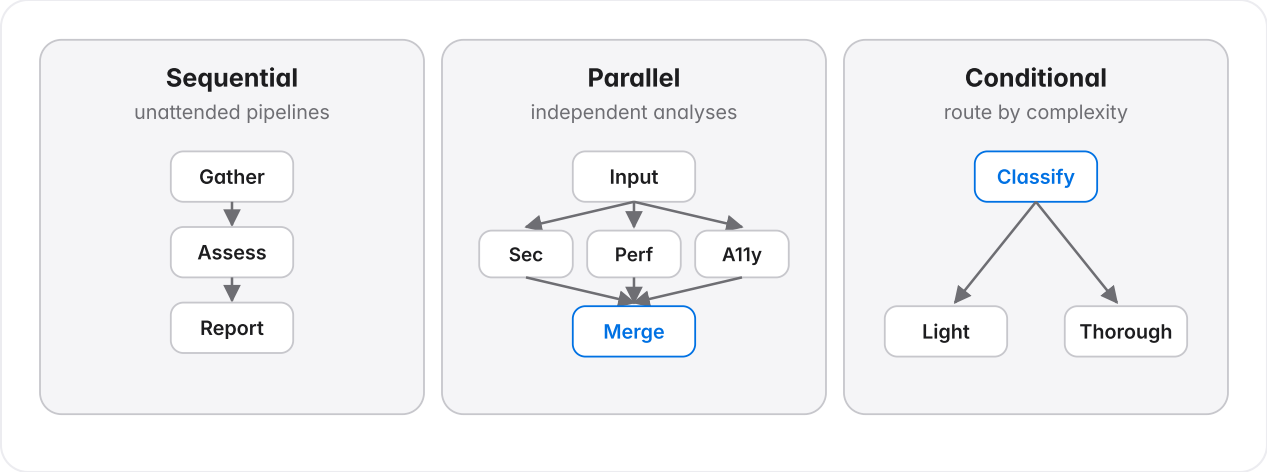


Figure 23.1 — Three orchestration patterns, used only when one agent's own loop isn't enough

Choosing

1. Can one agent, guided by the intent, do it in its own loop? **Then don't orchestrate.**
2. Must it run unattended, the same way every time? **Sequential pipeline.**
3. Are there independent analyses whose separation is measurably better? **Parallel.**
4. Does the right path depend on something learned in a first step? **Conditional.**

Start with the simplest pattern that meets the intent. Add structure only when the evidence shows the simpler pattern falling short.

> Orchestration is not architecture for its own sake. It is the shape work takes when one agent's loop is not enough. Most of the time, it is enough.

PART V · JUST ENOUGH PLATFORM
CHAPTER 24

MCP: Connect, Don't Build

The first edition introduced the Model Context Protocol as the "control plane" connecting agents to enterprise systems. MCP has since become the ordinary way agents reach tools and data. The idea is the same. The posture is different: **connect only what an intent needs, and prefer existing servers to new ones.**

What MCP Gives You

MCP standardizes how an agent discovers tools, calls them and receives results. One MCP server for your ticketing system serves every MCP-capable agent: Claude Code, Cursor, Copilot and others. That turns the integration problem from *agents × systems* into *agents + systems*.

The Least-Infrastructure Posture

Start from the intent. List the systems the intent's inputs and outputs actually touch. Connect those. A pre-deploy risk check needs deployment history and service metrics. It does not need the HR system.

Prefer existing servers. Many enterprise systems now have maintained MCP servers. Use them before building your own.

Scope tightly. Give each connection the narrowest permissions the intent requires: read-only where reading is enough, specific projects rather than all projects.

Centralize credentials, not logic. The first edition was right that agents should not hold raw credentials. Use your organization's standard secret management and identity, and keep business logic out of the MCP layer.

When to Build an MCP Server

Build one when an intent needs a system that has no server, and at least two intents or teams will use it. Build it as a small, owned service with tests. It is a tool in the taxonomy, so it needs functional tests and uptime, not intent-level governance.

> MCP is not a platform to build. It is a plug to fit. Connect what the intent needs, and nothing more.

PART V · JUST ENOUGH PLATFORM
CHAPTER 25

Governance Without Bureaucracy

Every enterprise AI conversation reaches the same three questions. Who controls what the AI does? How do we know it is doing it correctly? What happens when it goes wrong?

The first edition answered them with platform machinery: a policy engine in the execution path, an executor registry with approval workflows and a write-once audit store. Those are real options for heavily regulated settings. For most organizations, the same answers come from controls that already exist.

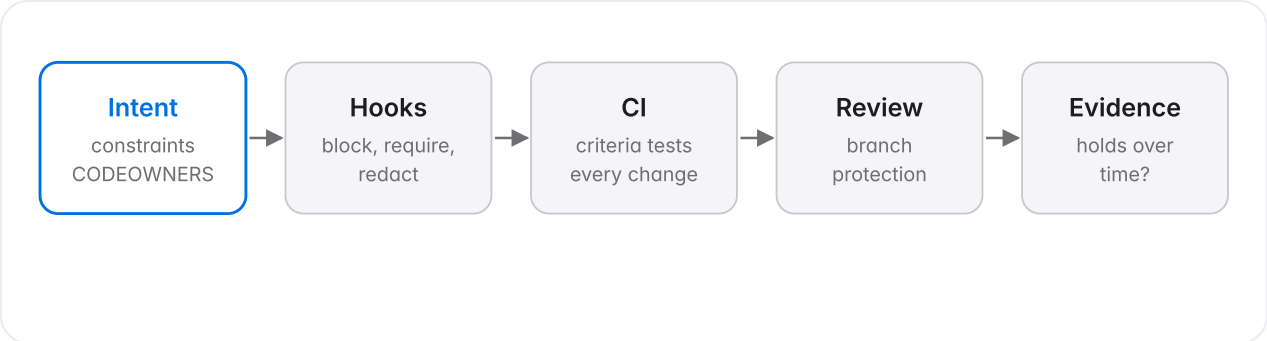


Figure 25.1 — Governance in the path you already have: intent, hooks, CI, review and evidence

Question 1: Who Controls What the AI Does?

The intent. Its constraints section declares what must not be violated. It is reviewed like code and owned through CODEOWNERS.

Permissions and hooks. Agent tools let you restrict which commands, paths and tools an agent may use. Hooks enforce the non-negotiables deterministically, for example blocking edits to protected paths, requiring tests before commit or stripping secrets.

Branch protection. No change reaches the main branch without passing CI and review, whoever or whatever wrote it. This one control does more for AI governance than any policy engine, because it applies uniformly to every tool.

Approved tools list. Which agents and model providers are permitted is an organizational decision, recorded once and enforced through procurement and identity. It does not need a runtime registry.

Question 2: How Do We Know It's Correct?

Success criteria as tests. Every intent's criteria run in CI. Correctness is evidence, not trust.

Review against intent. Reviewers check the change against a written definition of done.

Evidence over time. Tier 2 and Tier 3 metrics from Chapter 15 show whether correctness holds across hundreds of changes, not just one.

Human-in-the-loop where stakes require it. For high-stakes outputs, such as contract analysis that informs a deal or a customer-facing decision, the intent's constraints require human approval before the output is used. The approval is recorded wherever your workflow records approvals.

Question 3: What Happens When It Goes Wrong?

Detection. A failing criterion breaks the build. A degrading outcome metric shows in the evidence. An escaped defect is traced to the intent it should have been caught by.

Containment. Intents state their fallback in their constraints, for example *"falls back to the static checklist on any failure."* The fallback is tested like any other path.

Recovery. Revert the change. Git is the rollback mechanism, and it works for intent files exactly as it does for code.

Learning. Every escape adds a criterion or a note. Governance improves through the same Progressive IDE loop as everything else.

When You Need More

Regulated industries may need an immutable execution record, central policy enforcement or formal model risk management. When those requirements are imposed, they are exactly the platform signals described in Chapter 21. Build the minimum that satisfies them, and extract it from the controls that already work.

Addressing the Autonomy Concern

If agents act autonomously, how do we make sure human judgment is not bypassed where it matters? The answer is the same in every edition. Autonomy is bounded by the intent and its constraints. The agent does the work that should not need human judgment, and it routes to humans the decisions that should. The difference now is that the routing is a line in an intent and a required review, not a configuration in a platform.

> Governance is not the price of using AI. It is what makes AI trustworthy enough to use at scale. Put it in the path you already have, and it costs almost nothing extra.

PART VI

Architecture in Practice

Intent in the systems you already run

26 AI in the Architecture You Already Have

27 Observability and the Evidence Loop

28 How Teams Actually Work with This

PART VI · ARCHITECTURE IN PRACTICE
CHAPTER 26

AI in the Architecture You Already Have

Most enterprise engineering happens in a microservices landscape, often with an event backbone. The first edition gave that landscape two chapters: a three-lane architecture separating microservices, agents and shared infrastructure, and a Kafka-based intent trigger bus with consumers, input mappers and idempotency keys.

The principles still hold. The machinery is mostly unnecessary. This chapter keeps the principles and drops the machinery.

Two Kinds of AI in a Service Landscape

It helps to separate two very different uses.

AI that builds the system. Agents writing, reviewing, testing and migrating code. This is where most AI value is today. It needs no runtime architecture at all, only intents, tests and CI in each service's repository.

AI that runs in the system. Model calls made at runtime as part of a business process, such as a fraud prescreen on payment, a classification on ticket creation or a summary on document upload. This does need architectural care.

Keep them distinct. Many architecture conversations go wrong because the second kind's concerns are applied to the first.

Principles for AI That Runs in the System

1. Services own business logic; AI capabilities own reasoning. A payment service owns the transaction. It does not embed fraud reasoning in its own code. The fraud prescreen is a separate capability with its own intent.

2. The boundary is a contract. Runtime AI capabilities expose a typed input and output. The service validates AI output at the boundary before acting on it. This is where the first edition's output schema enforcement belongs: a few lines of validation, not a platform.

3. Events decouple, where you already use events. If your landscape is event-driven, a domain event such as `payment.created` can trigger the AI capability, which publishes its result as a new event. Use your existing consumers, schema registry and idempotency practices. AI capabilities are ordinary event consumers and should follow the same rules as every other consumer.

4. Every runtime AI call has a fallback. The intent's constraints state it, for example "approve and queue for manual review if the prescreen is unavailable." The fallback is tested.

5. Version the contract, not the model. The capability's input/output contract changes deliberately and is versioned. The model behind it can be upgraded freely, as long as the contract tests still pass.

What You Do Not Need

You do not need a separate agent lane, a dedicated intent trigger bus, an input-mapping service or a platform-level idempotency store. Your event platform, your service framework and your existing practices already provide these. A runtime AI capability is a service with a probabilistic dependency. Treat it like a service and protect it like a dependency.

> The microservice boundary and the AI capability boundary are both contracts. Honor both with the tools you already use, and the architecture stays coherent.

PART VI · ARCHITECTURE IN PRACTICE
CHAPTER 27

Observability and the Evidence Loop

An AI capability that cannot be observed is not a production system. It is a hope.

The first edition extended the three pillars of observability (logs, metrics and traces) with a fourth, *outcome tracking*, and built a five-stage automated feedback loop: execution capture, rolling trend analysis, alert routing, automated recompilation and A/B promotion. The fourth pillar was the right addition. Most of the loop's automation was premature.

The Four Pillars, Applied Simply

Logs. Runtime AI calls emit a structured log line: capability name, contract version, input fingerprint, latency, outcome (success, fallback or error) and criteria results where they can be checked. Use your existing logging stack.

Metrics. The three tiers from Chapter 15, sourced from tools you already have: pipeline history for Tier 1, CI and review data for Tier 2, business systems for Tier 3.

Traces. If you already use OpenTelemetry, instrument AI calls like any other dependency. A trace that shows the model call taking 9 of a request's 10 seconds answers most performance questions immediately.

Outcomes. Each intent's Evidence section names its outcome metric and its source. A scheduled job checks that the source has recent data and that the metric is moving in the stated direction.

The Evidence Loop

The first edition's feedback loop tried to close itself automatically. This edition closes it with a person on a schedule, assisted by an agent.

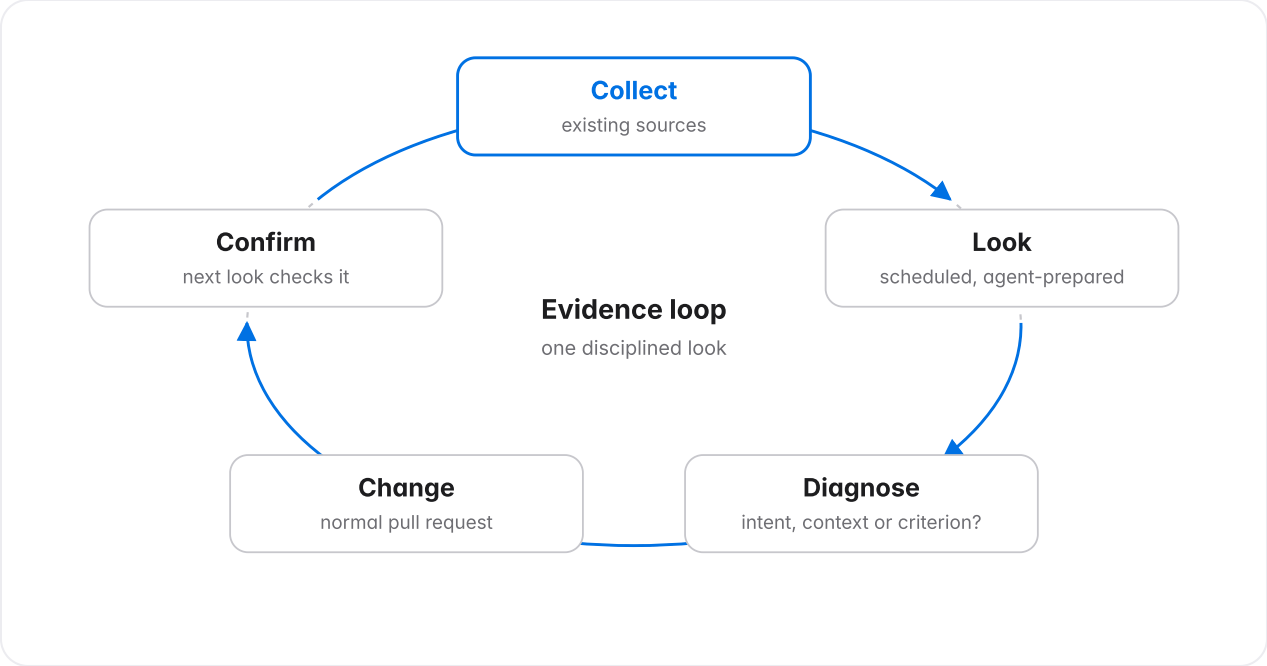


Figure 27.1 — The evidence loop: a scheduled look, an agent-assisted diagnosis, a reviewed change

1. **Collect.** Evidence accumulates in its existing sources.
2. **Look.** On a regular cadence, weekly for active capabilities and monthly for stable ones, the owner reviews the evidence. An agent can prepare the summary: what passed, what failed, what escaped and where the outcome metric is trending.
3. **Diagnose.** For each miss, ask the Progressive IDE question: was the intent unclear, was context missing or was a criterion too weak?
4. **Change.** Update the intent, add a test or add a note, through a normal pull request.
5. **Confirm.** The next look checks that the change helped.

That is the whole loop. It runs on your existing tools, and its cost is one regular look per capability. Teams that do that look consistently outperform teams with elaborate automated pipelines that nobody reads.

Fault Tolerance, Briefly

The first edition described six failure modes, each with a platform-level response. For runtime AI capabilities, the essentials fit in a short list:

Failure	Response
Invalid input	Reject at the boundary with a typed error. Never relax validation to hide it.
Transient provider error	Retry with backoff using your existing HTTP client policies.
Malformed output	Validate against the contract; on failure, use the fallback and log it.
Output fails criteria	Use the fallback or route to review, as the intent's constraints specify.
Provider outage	Circuit-break to the fallback. Alert the owner.
Rising fallback rate	Treat as an evidence signal; diagnose in the next look, or sooner if it crosses a threshold.

Every row uses a resilience pattern your services already have. None needs a dedicated platform.

> Observability tells you what is happening. The evidence loop tells you what to do about it. It needs one disciplined look, not a self-driving pipeline.

PART VI · ARCHITECTURE IN PRACTICE
CHAPTER 28

How Teams Actually Work with This

What does a Tuesday morning look like for a team practicing Intent-Driven Engineering?

When New Work Arrives

A ticket comes in. Before opening the agent, the engineer asks three questions. Is there already an intent nearby that covers this? Does the ticket state an outcome, or only an activity? How will we know it worked?

If an intent exists, the work extends it. If not, she writes one, usually in five to fifteen minutes. If she can't answer the success question, she goes back to the ticket's author. That conversation is the most valuable thing that happens all morning.

Then she runs the agent in whichever tool she prefers. Her colleague uses a different one. Both tools read the same `AGENTS.md` and the same intent.

When a Pull Request Opens

The pull request description links the intent and shows the evidence: which criteria tests were added or passed, and what the agent was told that it needed telling. The reviewer reads the intent first. Review is verification against a stated standard, and it is faster for it.

If the reviewer finds a miss that no criterion caught, the fix includes a new test or a new note in the intent. The miss becomes part of the system's memory.

When Something Breaks

The on-call engineer points an agent at the incident-triage intent with the alert and logs. The intent's criteria, such as "every hypothesis cites a log line or metric" and "no remediation that touches production data without confirmation," shape a first-pass analysis she can trust

enough to act on. After the incident, the review asks one extra question: *which intent should have caught this, and what does it need?*

Addressing the Real Concerns

"This feels like overhead. We just need to ship." Writing an intent takes minutes. Most of that time is spent deciding what success means, which you would otherwise decide during review, rework or an incident. The overhead is not added. It is moved to the cheapest point.

"Our work is too exploratory for this." Exploration doesn't need an intent. Once someone will depend on the result, it does. The gate is "will people depend on this?", not "is this AI?"

"We don't have a platform team." You don't need one. You need a template, an `AGENTS.md`, and a CI check. One engineer can set those up in an afternoon, or start from a starter repository and do it in minutes.

"Do we have to rewrite everything?" No. Add intents to new work. Add them to existing work when it causes an incident, needs extending to a new team or comes up in an audit. Formalize where the cost of informality has become visible.

"Our engineers use different tools." Good. Chapter 13's write-once, point-everywhere pattern means the intent works with all of them.

Team Growth

New engineers, and new agents, read the intents in the repositories they touch. They learn what each part of the system is for, how success is judged and what the team learned along the way, without a single briefing. The intents are the onboarding.

> Intent-Driven Engineering does not eliminate coordination. It removes the coordination that shouldn't need a meeting, and frees people for the decisions that do.

PART VII

Enterprise Execution

Taking the practice across the organization

29 The Pipeline You Already Have

30 Scaling Across Many Repositories

31 Organizational Adoption

PART VII · ENTERPRISE EXECUTION
CHAPTER 29

The Pipeline You Already Have

DevOps automated the path from commit to production, put quality gates at every stage and measured everything. Intent-Driven Engineering adds a few gates to that pipeline. It does not add a second pipeline.

The first edition specified an intent-extended CI/CD flow: intent lint, intent tests, compile, deploy to staging, a criteria gate on shadow traffic, promotion and automated rollback through the registry. The stages that remain are the ones that need nothing beyond your current pipeline.

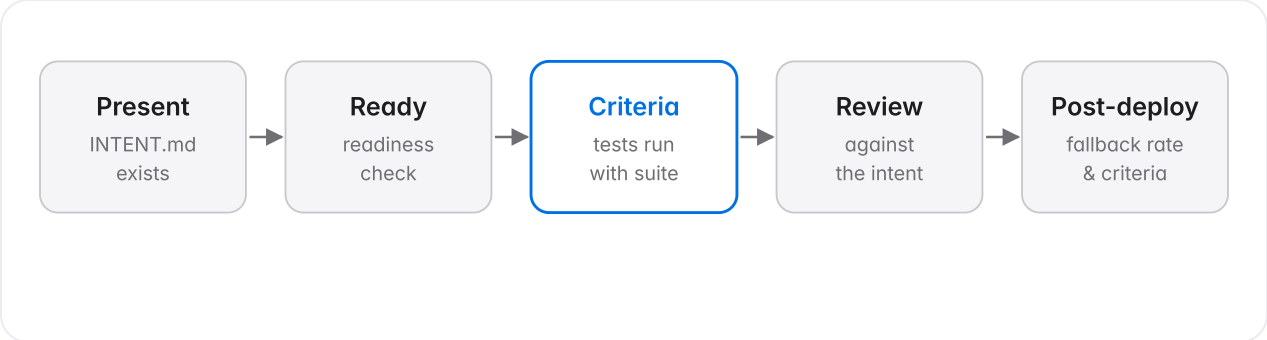


Figure 29.1 — Intent gates added to the pipeline you already run

Gate 1: Intent Present

For changes to feature folders, check that an `INTENT.md` exists and contains the six headings. It takes seconds and catches work that skipped the definition of done.

Gate 2: Intent Ready

Run the readiness check from Chapter 8: an outcome stated as a change, at least one checkable criterion, constraints present and an evidence source named. This can be a simple script or an agent-run skill that comments on the pull request. Treat it as advisory at first.

Make it blocking once teams are used to it.

Gate 3: Criteria Tests

The intent's success criteria, encoded as tests, run with the rest of the suite. A failing criterion blocks the merge exactly as a failing unit test does.

Gate 4: Review Against Intent

Branch protection requires review. The pull request template asks the reviewer to confirm that the change satisfies the linked intent and that the evidence is attached.

Gate 5: Post-Deploy Evidence (for Runtime AI)

For capabilities that run in production, the deployment's health checks include the capability's fallback rate and criteria results. A spike triggers the same rollback mechanism as any other bad deploy.

A Minimal Workflow

```
# .github/workflows/intent.yml
name: intent
on: pull_request
jobs:
  intent:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Intent present and ready
        run: ./scripts/intent-check.sh
      - name: Criteria tests
        run: make test-intent
```

That is the whole addition. Everything else, including build, deploy, rollback and alerting, stays as it is.

> The intent pipeline is not a separate pipeline. It is your pipeline, with a definition of done it can check.

PART VII · ENTERPRISE EXECUTION
CHAPTER 30

Scaling Across Many Repositories

Fifty repositories means fifty teams, fifty cadences and fifty places where the same capability might be built twice. The first edition answered this with a federated registry: domain registries and a platform registry, with a unified search catalog and compile-time cross-domain dependency checks.

The second-edition answer is **conventions over catalogs**.

The Four Conventions

1. Same file, same place. Every repository uses `INTENT.md` at the service root and in feature folders, with the same six headings. Agents and humans find intents by convention, not by search.

2. Same hub. Every repository has an `AGENTS.md` that points to intents and states the cross-cutting constraints. Tool-specific files are thin pointers to it.

3. Same starting point. New repositories are created from a starter, such as `intent-driven-starter` or your organization's equivalent, so the conventions are present from the first commit.

4. Same shared capabilities. Cross-cutting routines, including readiness checks, intent review and scaffolding, are distributed as versioned skills or plugins that every repository installs.

Portfolio Visibility

The generated `INTENTS.md` index from Chapter 17 gives leadership and architecture the portfolio view across all repositories. It is rebuilt on a schedule, with no curation. Gaps show up as empty evidence cells.

Preventing Duplication

Duplication is caught where it starts. When an engineer writes a new intent, an agent can search the organization's repositories for similar intents in seconds and suggest reuse. That is the check-before-build habit the first edition wanted from a registry, delivered through the tool the engineer is already using.

Cross-Team Dependencies

When one team's runtime capability is consumed by another, treat it like any service dependency: a versioned contract, contract tests on both sides and a deprecation window for breaking changes. This is ordinary service management and needs no intent-specific machinery.

The Platform Team's Role at Scale

The platform team's job is stewardship of the conventions and the shared capabilities: maintain the starter, version the shared skills and plugins, run the index and watch for patterns where three teams are solving the same problem. Its health metric is not how many things it built. It is **how much of the organization starts from what it maintains.**

> At fifty repositories, consistency comes from conventions every repository shares, not from a catalog every engineer is supposed to check.

PART VII · ENTERPRISE EXECUTION
CHAPTER 31

Organizational Adoption

The most complete discipline in the world delivers nothing if the organization does not adopt it. The first edition mapped adoption as a five-phase S-curve over twenty-four months, gated on platform construction. Without the platform dependency, the curve is faster and the first phase is measured in weeks.

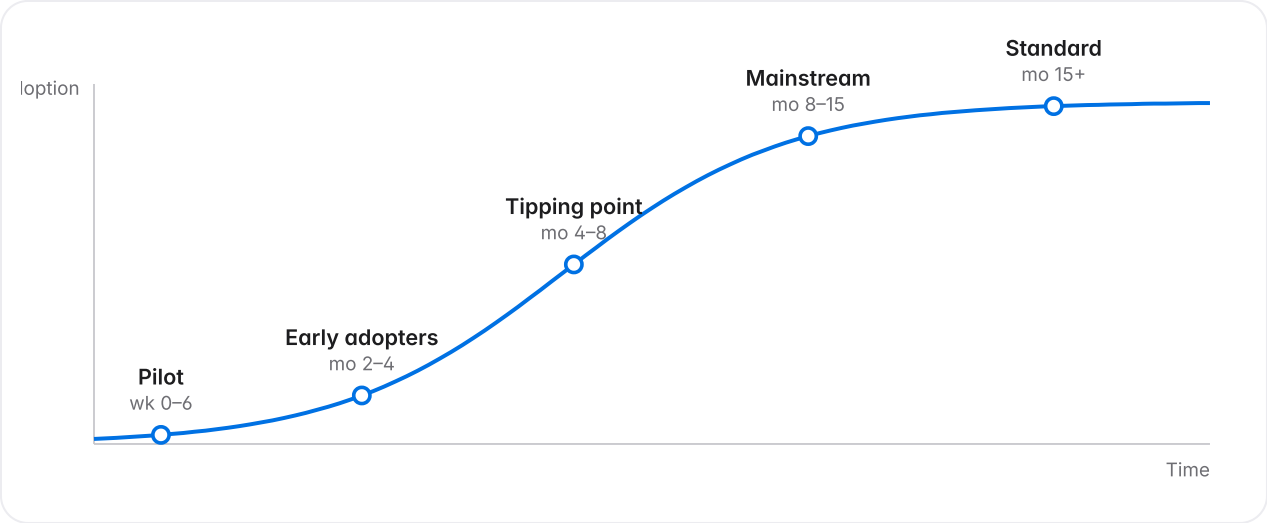


Figure 31.1 — The adoption curve, second edition: weeks to a pilot win, not quarters

Phase 1: The Pilot (Weeks 0–6)

Choose one team with three properties: they already use agents daily, they have a metric they genuinely want to move, and their lead is respected enough that a win will be believed. Avoid the team that is simply most excited about AI.

The pilot team picks the two or three pieces of recurring work that most affect their metric, writes intents for them, captures baselines and runs the Progressive IDE loop. There is no platform to wait for, because the starter repository, `AGENTS.md` and a CI check are set up in the first day.

By week six they have a story: *here is the intent, here is the baseline, here is the number now.*

Phase 2: Early Adopters (Months 2–4)

Share the pilot's result as the actual intent files and the actual evidence, not as a deck. Three to five teams adopt. The platform team, if there is one, packages what the pilot found reusable into the starter and shared skills.

The leading indicator of health is that new teams are **starting from** the starter and shared skills, not rebuilding them.

Phase 3: The Tipping Point (Months 4–8)

IDE moves from "something some teams do" to "how work is defined here." The signal is that new work arrives with an intent attached, or prompts the question "where's the intent?" from the reviewer.

At this point, make it a standard. Work that others will depend on has an `INTENT.md` with checkable criteria and named evidence, and the intent-present gate becomes blocking.

Phase 4: Mainstream (Months 8–15)

Intent is the default. The generated index is a standing item in engineering leadership reviews. Onboarding includes reading the intents in your first repository.

Phase 5: The Standard (Month 15+)

The question "should we write an intent for this?" stops being asked. The only questions are "what's the outcome?" and "how will we know?"

The Change-Resistance Map

The overloaded lead. The best engineer has no spare capacity. Make the first intents the ones that currently require their presence, such as triage, risky reviews or the scaffold only they know. Their load drops, and that is the argument.

The legacy AI workflow. A team already has a working agent workflow. Don't rewrite it. Write the intent it was implicitly serving, add its criteria as tests and attach its evidence. It joins the practice without changing.

The platform enthusiast. Someone wants to build the platform first. Redirect that energy into the starter, the shared skills and the index, which are the platform pieces the evidence actually justifies. Revisit the full platform when Chapter 21's signals appear.

The ROI skeptic. Do not argue for the discipline on principle. Show the pilot's Tier 3 number, its baseline and its source.

> Adoption is not a communications problem. It is a value-delivery problem. The organization adopts Intent-Driven Engineering when it sees outcomes the previous approach could not produce.

PART VIII

Anti-Patterns

What will pull teams backward, and how to recognize it in time

32 Prompt Addiction and Spec Bloat

33 Platform Before Proof

PART VIII · ANTI-PATTERNS
CHAPTER 32

Prompt Addiction and Spec Bloat

Part VIII exists because the most common reason organizations fail to reach the outcomes in this book is not a lack of capability or budget. It is the pull of patterns that feel productive while quietly preventing progress. This chapter names two of them, one old and one new.

Prompt Addiction

Prompt addiction looks like productivity. A developer asks the agent, gets a useful answer and ships. That is real. The addiction is in *repetition without accumulation*: the next developer faces the same class of problem, asks a similar question, gets a similar answer and ships something slightly different. At the end of the quarter, twenty developers have produced twenty isolated improvements instead of one improving system.

The first edition named five symptoms, and all five still appear:

1. **The hero prompt.** The one that works lives in someone's chat history and disappears when they do.
2. **The invisible regression.** A model update shifts behavior subtly and nobody notices for weeks.
3. **The quality negotiation.** Every output is judged by whoever is looking, against a standard nobody wrote down.
4. **The undocumented dependency.** Something starts depending on the shape of an AI output that nobody declared.
5. **The confidence mismatch.** Leadership asks how good it is, and the answer is "pretty good."

The remedy is the same in every case: **write the intent, encode the criteria as tests, keep the evidence.** The hero prompt becomes an intent in the repository. The regression fails a criterion. The quality standard is written down. The dependency becomes a contract. The

confidence question gets a number.

Spec Bloat

Spec bloat is prompt addiction's mirror image, and it is the anti-pattern this edition was most needed to name. Where prompt addiction says too little, spec bloat says far too much.

Symptom 1: The generated spec nobody read. A spec-driven tool turns a one-line request into twelve pages. The pages are accepted wholesale. Three weeks later, nobody can say which requirements were decisions and which were the tool's guesses.

Symptom 2: The spec that outlived the system. The spec was written for version one. The code is on version four. The agent still reads the spec as truth and keeps "fixing" the code back toward a design that was deliberately abandoned.

Symptom 3: The over-constrained agent. The spec prescribes library choices, file structures and function names the author did not care about. The agent complies, produces something worse than it would have chosen and no one can tell which constraints matter.

Symptom 4: Instructions as sediment. Each miss is fixed by adding a paragraph to the rules file. Nothing is ever removed. After six months the file is three thousand words long and contradicts itself, and the agent's behavior has become *less* predictable.

Symptom 5: Spec time exceeds build time. Writing and reviewing the specification takes longer than the agent takes to build the feature and the human takes to verify it. The ratio is upside down.

The Remedy for Spec Bloat

Climb the ladder, don't jump to the top. Start at the lowest rung the risk allows. Add specification only to explain a miss.

Prune on every pass. When refining an intent, ask what to *remove* as well as what to add. Anything that has never prevented a miss and isn't a real constraint is a candidate.

Date and attribute every note. Each line in an intent's Notes says what miss it fixed. Lines that cannot justify themselves get deleted.

Treat generated specs as drafts. A spec-driven tool's output is a proposal. Keep the intent upstream and prune the spec down to what the intent needs.

Watch the ratio. Track intent length against criteria pass rate (Chapter 15). Rising length with a flat pass rate means bloat.

> Prompt addiction and spec bloat are opposite errors with the same root: treating the instruction as the work. The work is the outcome. The instruction should be exactly as long as the outcome requires.

PART VIII · ANTI-PATTERNS
CHAPTER 33

Platform Before Proof

The first edition's second anti-pattern was **tool hoarding and architecture drift**: organizations that acquired every AI tool, kept them all and ended up with systems that could not share context or be governed together. That anti-pattern is still real. This edition adds its larger sibling.

Tool Hoarding, Revisited

Tool hoarding still feels strategic: evaluate everything, commit to nothing, keep every license. The first edition's remedy was right and is now easier. **Commit to the intent, not to the tool.** When intents live in the repository and every tool reads them through `AGENTS.md`, teams can use different agents and switch freely without the capability library changing. Tool choice becomes a preference rather than an architecture decision.

Platform Before Proof

Platform before proof is the anti-pattern in which an organization builds AI infrastructure before it has proven intents that need it. It is the second edition's most important warning, because this book's first edition encouraged it.

What it looks like. A platform team spends two quarters building an agent orchestration layer, an intent API, a capability catalog and a governance engine. The launch demo is impressive. Then the platform waits for work. Product teams, meanwhile, have moved on with Copilot, Cursor and Claude Code, which already do most of what the platform does, inside the tools they use every day.

Why it happens. Platforms are visible, fundable and interesting. They look like strategy on a slide. They let a team show progress without having to move an outcome metric.

What it costs. Every layer needs ongoing work: upgrades, security review, on-call, documentation. Without proven intents running on it, nothing offsets that cost. Worse, the platform becomes the initiative's story, so the 60-day ultimatum arrives with a beautiful platform and no outcomes.

The Drift Trajectory, Updated

Period	Tool hoarding path	Platform-before-proof path
Months 1–3	Many tools adopted independently	Platform design and build begin
Months 3–6	Instruction files diverge	Platform launches; few workloads
Months 6–9	Behavior inconsistent across teams	Teams bypass the platform with their editors' agents
Months 9–12	Consolidation request	Platform ROI questioned
Months 12+	Expensive consolidation	Platform rescoped or retired

The Coherence Test, Revised

Ask these five questions and count the honest yes answers:

1. Can anyone find the intent governing a given piece of AI-assisted work in under a minute, by convention?
2. Can you show leadership, today, each important capability's baseline and current outcome?
3. If your best AI engineer left today, would their capabilities keep working and be improvable from what is in the repository?
4. Can a team switch agent tools without rewriting any intents?
5. Can you name, for every piece of AI infrastructure you run, the proven intents that required it?

Five yes answers means you have coherence. Four or fewer means drift is under way. Each no points to a specific chapter of this book, and none of the fixes requires building something large.

> Architecture drift is cheaper to prevent than to correct, and so is a platform nobody uses. The prevention for both is the same: intent first, evidence second, infrastructure only when the evidence demands it.

PART IX

Real Systems

The full loop, nothing held back

34 From Idea to Running System

PART IX · REAL SYSTEMS
CHAPTER 34

From Idea to Running System

Every chapter has built toward this one. Here is the full journey for a real class of problem: identifying the capability, writing the intent, running the agent, gathering the evidence and seeing the outcome. Every step uses only what a typical team already has. *(The scenario and its figures are illustrative. Run it on your own problem and record your own numbers.)*

The Problem

A product engineering team's review cycle averages 6.4 hours from pull request opened to approved. It is the single biggest contributor to their lead time. Junior reviewers miss security issues. Senior reviewers are a bottleneck. Agent-authored pull requests have made the queue longer, not shorter.

The capability: **prepare every pull request for human review**, with a structured first pass that flags real issues, assesses test coverage on changed files and lets the human reviewer spend their time on judgment.

Day 1 — Look Before You Build

The engineer asks her agent to search the organization's repositories for existing intents about pull request review. It finds one in another team's repository: a Python-focused review-prep intent with six months of notes. Her team works in TypeScript.

Decision: reuse the intent's structure and notes, adapt them for TypeScript, and tell the other team. Good citizenship means both teams may later merge into a single shared skill.

Day 1 — State the Intent (Rung 1)

```
# Intent: PR review preparation

## Outcome
Review cycle time drops from 6.4 h (baseline, last 60 days)
to under 3 h, with no increase in escaped defects.

## Inputs
- The PR diff, description and linked ticket
- Team conventions in AGENTS.md

## Outputs
- One review comment with: blockers, majors, minors
- Test-coverage assessment per changed file
- A suggested reviewer focus ("look hardest at X")

## Success criteria
- Every blocker cites file + line and includes a suggested fix
- Coverage assessed for every changed source file
- Zero known security issues missed on the seeded test set
- Human reviewers rate ≥ 80% of blockers as valid (sampled weekly)

## Constraints
- Advisory only; never approves or requests changes itself
- Never comments on generated or vendored files

## Evidence
- Seeded test set in tests/intent/pr-review/ (20 PRs, known issues)
- Review cycle time from PR analytics
- Weekly validity sample in the team's review log
```

Ninety minutes. Most of it was spent building the seeded test set: twenty historical pull requests with known issues, including four security problems that reviewers had missed. That test set is the most valuable artifact of the week.

Day 2 — Run, See, Refine

She implements the capability as a CI job that runs the team's agent with the intent and a review skill, then posts the comment.

First run against the seeded set. It catches three of four security issues and assesses coverage on 17 of 20 pull requests. It misses an authorization check on a new endpoint, and it skips coverage on files in a new directory.

Refine (rung 2). Two notes are added:

```
## Notes
- New route handlers: always check for the authz middleware
  (see src/http/authz.ts). Missed on seeded PR #14.
- "Changed source files" includes new directories under src/.
  Missed coverage on seeded PRs #3, #9, #17.
```

Second run. Four of four security issues are caught and coverage is assessed on all twenty. The criteria pass.

Day 3 — Review and Lock

A colleague reviews the intent, the job and the seeded results. He adds one constraint: *"Skip PRs labeled `docs-only`."* The seeded set becomes a CI test for the capability itself, so any future change to the intent or the skill must still catch all four security issues. The change merges, and the job runs on every pull request.

Week 1 — Evidence

The job runs on 310 pull requests. Human reviewers rate 84% of blockers as valid. One skeptical senior engineer, who said "I don't need a machine to review my code," posts three examples of issues the first pass caught that he would have missed on a busy afternoon.

Review cycle time on pull requests the job ran on: 2.9 hours, against a 6.4-hour baseline.

Week 4 — First Evidence Loop

The weekly look surfaces a pattern. On pull requests with more than forty changed files, coverage assessment is shallow, because the agent summarizes instead of checking each file.

Diagnose. It is a context-window problem, not an intent problem. **Refine.** The job splits large diffs into file groups and merges the results, and a note records why. A test with a large seeded pull request is added.

Coverage criteria on large pull requests now pass consistently. Review cycle time is 2.1 hours, a 67% reduction from baseline. Escaped defects over the same period are flat.

What This Walkthrough Shows

1. **It takes the time of a feature, not a platform.** Three days to running, four weeks to a measured outcome.
2. **The seeded test set is the intent's backbone.** Success criteria you can run are worth more than any amount of prose.
3. **Progressive IDE kept the intent small.** It gained four notes and one constraint, and each one traces to a specific miss.
4. **Human judgment is preserved.** The reviewer still decides. The machine prepares.
5. **The Tier 3 number ends the conversation.** When leadership asks, the answer is 6.4 hours to 2.1, measured from the same source as the baseline.

> Every capability your organization builds can follow this path. Start with the problem that matters most, state the smallest intent that could solve it, and run the loop until the number moves. Then do it again.

PART X

The Future

What has been built, and what it opens

35 Owning the Practice

PART X · THE FUTURE
CHAPTER 35

Owning the Practice

The first edition closed with a chapter called *Owning the Category*. It claimed that Intent-Driven Engineering was "the standard": complete, proven and specified with enough precision that teams could implement it without further instruction.

The second edition makes a smaller and stronger claim. Intent-Driven Engineering is a **practice** that any team can start today with what it already has. It produces measurable outcomes, and it improves with every pass of its own loop. This book has improved the same way. The first edition was its rung-4 specification. This edition is what remained after the evidence came in.

What This Book Has Established

The problem. Organizations generate AI activity without AI outcomes. Agents have made activity easier to produce and easier to mistake for progress.

The root causes. These are no shared intent, activity metrics, no evidence loop, infrastructure before proof and specification as a substitute for judgment.

The practice. State the outcome, constraints and checkable success criteria before the work begins, in the smallest form that can succeed. Run any competent executor. Accept on evidence. Refine only where the evidence shows a miss. That is Progressive IDE.

The principle. Use the least infrastructure that reliably produces the outcome. Extract platforms from success.

The mechanics. `INTENT.md` beside the code, `AGENTS.md` as the hub, criteria as tests, CI as the gate, Git as the registry, skills and starter repositories as the units of reuse, and evidence from the systems you already run.

The Vocabulary

A shared vocabulary makes conversations between teams and organizations productive.

Term	Meaning
Intent	What must be true when the work is done, what must not be violated and how we'll know
INTENT.md	The file that holds an intent, beside the code it governs
Progressive IDE	The execution model: state, run, see, refine, lock
Specification ladder	Sentence → intent → context → contract → governed capability; climb only as needed
Least-Infrastructure Principle	The least machinery that reliably produces the outcome; each rung earned
Success criteria	Checkable conditions, ideally tests, defined before the work begins
Evidence	The durable record that the outcome happened and keeps happening
Readiness check	Outcome stated, criteria checkable, constraints present, evidence named
Spec bloat	Specification growing faster than the learning it represents
Platform before proof	Infrastructure built before a proven intent needs it

When two engineers from different organizations say *"our review-prep intent took review cycle time from 6.4 to 2.1 hours on four notes and one constraint,"* they understand each other immediately.

What Gets Built Next

The tooling ecosystem is already here, and it is the agents themselves. What remains is the practice layer: readiness checks packaged as skills, intent-review routines, starter repositories for common stacks, and evidence collectors that read the systems teams already run. Much of it is open on LearnTeachMaster, intent-driven-engineering.com and repogenic.com.

Domain extensions will follow: intent templates and seeded test sets for regulated industries, healthcare protocols, financial controls and platform operations. Each will be a small, reusable unit rather than a platform.

A community of practice will compare not tools but evidence: *what was your baseline, what is your number now, and how short is your intent?*

The Invitation

The practice is not complete. It will only be complete when the teams practicing it have extended it beyond what any one book can describe.

Write your first intent. Run it with the agent you already use. Look at the evidence. Add only what explains the miss. Lock it with a test. Then write the second.

When someone asks why your AI initiative is producing measurable results while others are still producing activity, you will have a precise answer. *We stopped treating AI as a tool to roll out and started treating outcomes as something to engineer. We said what we wanted before we started. We did the least that worked. We kept the evidence.*

That answer is Intent-Driven Engineering.

SUMMARY

What This Edition Changed, and Why

First edition	Second edition	Why
YAML intent files with typed schemas	`INTENT.md` in markdown, six headings	Readable by every human and every agent; schema only where a consumer depends on shape
An IDE compiler	The agent you already use	Agents now plan, execute and iterate
Central intent registry	The repo is the registry; generated `INTENTS.md` index	Engineers don't browse catalogs; intents belong where work happens
WEGA platform as the destination	Least-Infrastructure Principle; platform by extraction	Infrastructure before proof stalled initiatives
Four questions	Six fields: outcome, inputs, outputs, success, constraints, evidence	Outcome anchors the work; evidence makes success durable
Full specification up front	Progressive IDE: climb the ladder only as needed	Over-specification produced drift, not quality
Automated feedback and recompilation	Scheduled evidence loop, agent-assisted	A disciplined look outperforms an unread pipeline
40 chapters	35 chapters	The book follows its own principle

The Monday Morning Commitment

Five actions stand between reading this book and running the practice it describes.

1. Write one outcome. Pick the recurring piece of work that most affects a metric your team cares about. Write one sentence: what will be different when it's done right? Record today's baseline.

2. Write the intent. Add the six headings beside the code. Keep it under a page. Pass the readiness check. If it takes more than an hour, the scope is too broad, so narrow it.

3. Run it with the agent you already use. Point `AGENTS.md` and your tool's instruction file at it. Let the agent propose. Look at the evidence. Add only what explains the miss.

4. Lock it in the pipeline. Turn the success criteria into tests. Add the intent-present and criteria-test gates to the CI you already have.

5. Share the number. When the outcome moves, show the intent file, the baseline, the current value and where both came from. That is how the practice spreads: through evidence that makes it obvious to anyone who sees it.

One team. One intent. One measured outcome. That is how every lasting engineering practice has spread.

A Final Word

The AI era will not be defined by which organizations had access to the most capable agents. Every organization will have access to the most capable agents.

It will be defined by which organizations had the discipline to say what they wanted, do the least that worked and prove that it worked.

That discipline is not expensive. It is not a platform. It is an intent, a test and the willingness to look at the evidence.

Intent-Driven Engineering is that discipline: smaller than it was, stronger for it, and ready for Monday.

APPENDIX A

The INTENT.md Template

```
# Intent: <short name>

## Outcome
<What is different in the world when this is done?
Name the metric and its baseline if one exists.>

## Inputs
<What the work starts from. Only what the agent can't discover.>

## Outputs
<What is produced. Describe shape only if someone depends on it.>

## Success criteria
- <Checkable: a test, a measurement, or a yes/no review question>
- <Checkable>

## Constraints
- <Non-negotiable: security, data, compatibility, scope>

## Evidence
- <Where the proof lives: test path, CI check, dashboard, export>

## Notes
<Empty at first. Each line added after a miss, with the reason.>
```

APPENDIX B

Progressive IDE Quick Reference

The loop: State → Run → See → Refine → Lock

Step	Do	Don't
State	Smallest intent that could succeed; pass readiness	Pre-plan the implementation
Run	Use the agent you already have; let it propose	Rewrite the intent for each tool
See	Compare the result to the intent's criteria	Compare it to what you'd have written
Refine	Add only what explains the miss; prune what's unused	Add paragraphs "just in case"
Lock	Criteria → tests; commit intent with evidence	Accept on "looks right"

The specification ladder: 0 Sentence → 1 Intent → 2 Context → 3 Contract → 4 Governed capability

The infrastructure ladder: 0 Repo + INTENT.md → 1 Tests → 2 CI check → 3 Shared skill / plugin / starter → 4 MCP connections → 5 Scheduled / event runs → 6 Dedicated platform

The rule for both ladders: climb a rung only when evidence from the rung below demands it.

APPENDIX C

Readiness Checklist

Before the first run:

- [ ] **Outcome** is a change in the world, not an activity
- [ ] At least one **success criterion** is checkable without asking the author
- [ ] **Constraints** list every non-negotiable you'd block a PR over
- [ ] **Evidence** source is named and exists today
- [ ] A **baseline** is recorded, if the outcome is a metric
- [ ] The intent fits on **one page**

Before adding infrastructure:

- [ ] I can name the **running intent** that needs it
- [ ] I have **evidence** that the current rung is insufficient
- [ ] This is the **smallest addition** that closes the gap
- [ ] It has an **owner**

APPENDIX D

Tool Pointer Snippets

AGENTS.md (hub)

```
Every non-trivial change starts from the nearest INTENT.md.
Its success criteria are the definition of done.
Criteria tests live in tests/intent/. Run `make check` before proposing.
```

GitHub Copilot — `.github/copilot-instructions.md`

```
Follow AGENTS.md. Read the nearest INTENT.md before changing code.
```

Cursor — `.cursor/rules/intent.mdc`

```
---
alwaysApply: true
---
Follow AGENTS.md. Read the nearest INTENT.md before planning.
```

Claude Code — `CLAUDE.md`

```
@AGENTS.md
Read the nearest INTENT.md before planning significant work.
```

Tool mechanisms evolve. Keep the pattern: one shared source, thin pointers.

Resources

- **LearnTeachMaster** — learnteachmaster.org — free, open platform for Intent-Driven Engineering
- **Intent-Driven Engineering** — intent-driven-engineering.com
- **Repogenic** — repogenic.com — case studies in AI-native engineering
- **intent-driven-starter** — github.com/kendallmark3/intent-driven-starter — starter repository and plugin

SOFTWARE ENGINEERING · ARTIFICIAL INTELLIGENCE

Every developer has an agent. So why hasn't anything measurably changed?

Copilot, Cursor, Claude Code and spec-driven toolkits are now standard in enterprise engineering, and the adoption dashboards are green. The question from the steering committee hasn't changed: *what measurably changed because of any of this?*

This second edition of *The Intent-Driven Enterprise* rebuilds Intent-Driven Engineering around what a year in the field showed. You get to outcomes through the least specification and the least infrastructure that reliably produce them, not through bigger specs or bigger platforms.

- **Progressive IDE**: state, run, see, refine, lock. A practical execution model for building better software with less specification.
- **The Least-Infrastructure Principle**: why your repo, your tests and your CI are already the platform, and how to tell when they aren't.
- **INTENT.md**: one six-field intent, written once and read by every agent and every reviewer.
- **Reuse without a registry**: starter repos, skills and plugins that actually travel.
- **New anti-patterns**: spec bloat, and platform before proof.

Mark Kendall is an enterprise AI architect and the founder of Intent-Driven Engineering and LearnTeachMaster, a free, open platform for the discipline. He leads large-scale AI-assisted engineering adoption across enterprise engineering organizations.

learnteachmaster.org
intent-driven-engineering.com
Published by LearnTeachMaster

ISBN [TO BE ASSIGNED]