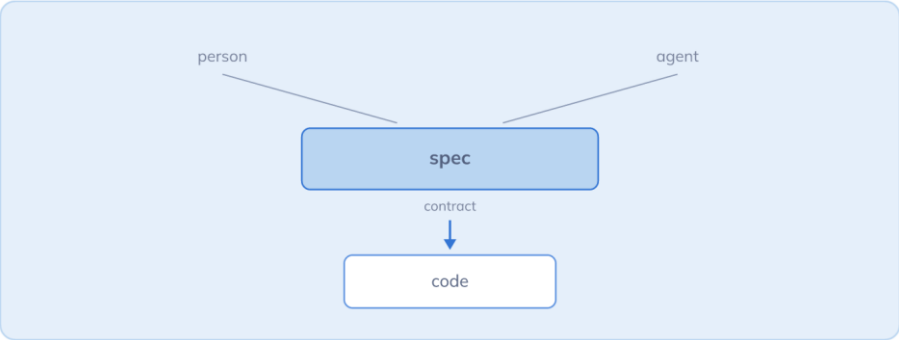
Scrum Manager®
Uncovering better ways

Guide

Spec Driven Development

in agile teams

In-depth development of the State of the art topics



Updated September 2026

Scrum Manager® · Skill Arena

About this document

This document is an in-depth development of the topics in the reference map (State of the art) for steering AI-assisted software development through specifications (Spec Driven Development) in agile teams, which is available at: Skill Arena.

Use it as reference material to keep your practice aligned with, and checked against, current professional knowledge.

It is complemented by the training and assessment platform in Skill Arena. In the Spec Driven Development in agile teams area you can take training tests to check and improve your level of knowledge and, if you wish, you can also obtain a diploma that provides curricular accreditation of professional competence and currency in this area.

State of knowledge

Knowledge about Spec Driven Development evolves extremely fast: method, tools, and practices are renewed within months. The fundamentals are already settled and the method gained academic and institutional backing during 2026, but the toolkit is still taking shape and the evidence on how much AI really speeds up development is under revision. Throughout the document, the labels (ESTABLISHED, CONSOLIDATING, EMERGING) help identify the maturity of each concept:

ESTABLISHED settled consensus; knowledge taken as required.
CONSOLIDATING gaining adoption fast; not yet universal but already relevant.
EMERGING recent frontier; high relevance and high volatility.

Contents

Chapters of the guide, in the order of the State of the art.

BLOCK A · THE PARADIGM: WHY SDD

Chapter 1. From vibe coding to spec-driven development · ESTABLISHED

Vibe coding is productive for prototypes and has a ceiling; SDD answers that ceiling by writing the contract before the code.

Introduction

In February 2025, Andrej Karpathy popularized the expression vibe coding to describe a way of programming with AI that many were already practicing without a name. You describe in natural language what you want, the model generates the code, you try it and, if it fails, you feed the error back until it gets by. There is no prior design or specification. The prompt is the intent, the code is the answer, and the conversation is the method.

In this first chapter we will see why that approach works up to a point, what evidence there is about its limits and how carefully it should be handled, and why spec-driven development emerges as the answer. It is the conceptual base of the rest of the guide, and it affects programmers and non-programmers alike.

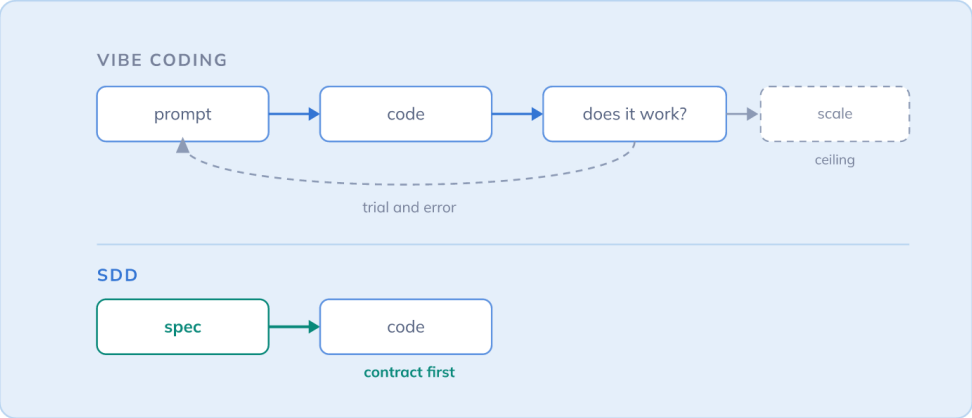


Figure 1. *SDD inverts the order: vibe coding converses with the agent and SDD hands it a contract before generating code.*

1.1 Where vibe coding works

It is worth starting by acknowledging its merit. For prototypes, exploring ideas, single-use internal tools, and everything where getting there early matters more than getting there right, vibe coding is extraordinarily productive. It lets you test a hypothesis in an afternoon, show a customer something tangible before deciding whether it deserves investment, and automate small tasks that would never have justified formal development.

The problem is its ceiling, which appears sooner than most people expect. Beyond a certain scale, the conversation stops holding the design together. Implicit requirements appear that nobody wrote and that the agent resolves its own way, the context gets

contaminated by the pile-up of failed attempts, and the architecture drifts, because each patch is decided on the spot and with no memory of the previous ones.

1.2 The evidence, with care

The most cited figure in the field is still the controlled trial that METR published in July 2025 with experienced developers on open-source projects. With AI they took 19% longer to complete their tasks, while believing they were going 20% faster. The gap between perceived and measured speed became the central argument of those who advocated bringing method to work with agents.

That figure is still published and can be cited, but no longer as a conclusive or current measurement. In February 2026 METR announced it was changing the design of its experiment. Its follow-up study, with 57 developers, 143 repositories, and more than 800 tasks, gave −18% for the ten participants of the original study and −4% for the forty-seven new ones, with confidence intervals that in both cases cross zero. The cause METR declares is serious selection bias, and its conclusion is that its estimate is a lower bound and that today developers are probably more accelerated by AI than what it measured in 2025.

The argument does not depend on a figure. The gap between what is perceived and what is measured is still documented, but it has gone from “it is measured” to “it is under revision”. A professional who cites the 19% without dating or qualifying it is stating something its own author no longer holds.

What does hold, and with more recent evidence, is the diagnosis of the cost. The DORA report on the return on investment in AI-assisted development describes two named phenomena. The J-curve is the temporary drop in productivity that precedes the capture of value, the tuition fee of the transformation. The verification tax is the added effort of checking that the generated code is reliable, secure, and consistent with the architecture. It should be said honestly: DORA backs the problem, and does not name specifications among the return factors. The solution this guide teaches is defended by other means.

1.3 The real cost is cognitive

When someone writes code, their working memory retains the context of what they have just written. When they review code generated by someone else, the comprehension load is different and accumulates with each iteration. The paradox of vibe coding is that a tool designed to save build effort ends up generating verification effort, which is more tiring and worse distributed, because it all arrives at the end and without the mental map that comes from having built it.

The verification tax does not disappear with any methodology. What changes is where it is paid. Paying it in the specification, before generating a line, costs less than paying it by reviewing code whose design nobody decided. That is the intuition that gives rise to SDD.

1.4 The answer: spec-driven development

Spec-driven development (SDD) inverts the order. Instead of jumping to the code, a specification is produced that defines what is built, how, and in which steps, and only after reviewing and approving it is it implemented. Birgitta Böckeler's definition is terse and enough: SDD means writing a spec before writing code with AI. Where vibe coding treats the agent as an interlocutor to converse with, SDD treats it as an executor that receives a clear contract.

SDD is a methodology and not a tool or a framework. It is a way of organizing work that can be implemented with different instruments, as we will see in chapter 18. And three frequent misunderstandings are worth clearing up from the start. SDD does not consist of documenting after building, because the order is precisely what matters. It does not require two-hundred-page specifications, because a good spec is smart rather than long, as chapter 10 develops. And it does not mean abandoning agility, but applying it coherently when part of the team is agents, which is the subject of chapter 3.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain what vibe coding is and in which contexts it is the right tool.
- Recognize the three signs of the vibe coding ceiling at scale: implicit requirements, context contamination, and architectural drift.
- Cite METR's evidence with its date and its later qualification, and distinguish what is measured from what is under revision.
- Explain the J-curve and the verification tax of the DORA report, and what they back and what they do not.
- Define SDD as a methodology and state its premise: the spec precedes the code.
- Distinguish SDD from after-the-fact documentation, from exhaustive specifications, and from a supposed abandonment of agility.

Common mistakes and antipatterns

Treating vibe coding as an enemy to eradicate. It is the right tool for prototypes and exploration. SDD complements it where conversation does not scale and does not replace it everywhere.

Citing METR's 19% as a current figure. The study was redesigned in 2026 and its own author considers it likely that the real speed-up is greater. The argument rests on the perception-measurement gap, not on the figure.

Crediting DORA with an endorsement of SDD. The report describes the verification tax and the J-curve, but does not name specifications among the return factors. It backs the problem, not the solution.

Believing SDD is a tool you install. It is a way of organizing work. Tools support it, they do not define it.

Further reading

· METR — We are changing our developer productivity experiment design (2026)

· METR — Measuring the impact of early-2025 AI on experienced developer productivity

· DORA — ROI of AI-assisted Software Development

· Böckeler — Understanding Spec-Driven Development

BLOCK A · THE PARADIGM: WHY SDD

Chapter 2. The spec as the primary artifact of development · ESTABLISHED

The center of the work shifts from writing code to specifying clearly what is built, why, and under what constraints.

Introduction

Addy Osmani sums it up in a sentence worth keeping at hand: AI is not the bottleneck; your spec is. For decades, the core competence of the software professional has been writing code. SDD proposes that this competence shifts without disappearing, because understanding code is still necessary to review and decide, but it stops being the central act. What makes a team with agents productive is its ability to specify clearly what to build, why, under what constraints, and with what success criteria.

In this chapter we will see what it means to treat the spec as the primary artifact, why it works as a contract and not as an order, and what changes for each role on the team, including those who do not program.

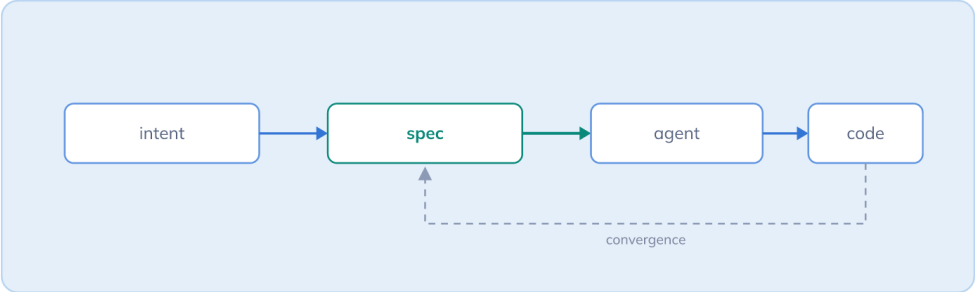


Figure 2. *The spec as a contract: the intent of the people becomes a contract that the agent executes and against which the code is verified.*

2.1 Code as a consequence

The most useful analogy is architecture. An architect does not lay bricks, they design blueprints that others execute, and the quality of the building depends on the quality of the blueprints. The spec is the blueprint and the code is the building. When something fails, the first question stops being what defect the code has and becomes what we did not specify well.

The reason is that the agent is literal. Every decision the spec does not make is a decision the agent makes on its own, and each of them is a potential point of divergence between what was wanted and what is obtained. A user story that a person would interpret with common sense, an agent will implement to the letter or will invent whatever it lacks. Specifying stops being a preliminary formality and becomes the activity that determines the result.

2.2 Contract, not command document

What distinguishes SDD from mere prior documentation is that the spec works as a contract between the parts of the system. People approve it at the phase gates, agents execute it, and what is delivered is code that fulfills the contract. The underlying bet of the method is that explicit contracts, at the right granularity, allow AI development at team scale to move faster and with fewer surprises.

During 2026 this idea has gained academic formulation. The work of Díaz, Gayoso, Cimmino, and Pérez on spec-driven development for agentic software engineering describes specifications as the contractual substrate between humans and agents, and holds that they are what restores the accountability and verifiability that informal AI assistance erodes. The authors themselves present their proposal as a first step toward a consensus between academia and industry, still pending empirical validation, and it should be cited with that same caution.

Approved, not launched. A spec handed to the agent without anyone having read it with judgment is not a contract, it is an order with the appearance of method. The value lies in deliberate approval, as we will see in chapter 7.

2.3 What changes for each role

The shift affects the whole team and not only the programmer, because specifying precisely is a shared competence.

- **Product Owner and Product Architect.** The demand for precision rises. Acceptance criteria go from being a guide for the conversation to being part of an execution contract.
- **Developer and Product Builder.** The competence shifts from the programming language to designing specs and to orchestrating and reviewing agents. The profile looks more like a technical lead than a programmer.
- **Agile Enabler.** Approval gates appear as inspection points that have to be facilitated without becoming bottlenecks.
- **Stakeholders.** They gain visibility, because a spec is readable by people without technical training to a degree that code never was.

These roles are the ones the Skill Arena skill “Scrum in teams with AI” describes for the team that works with agents. Chapter 15 returns to them with an eye on the ceremonies.

2.4 Readability and accountability

That the spec is readable has a consequence beyond convenience. It reduces the information asymmetry between those who build and those who pay for or use the product. A stakeholder can read a spec and weigh in on priorities or on compliance before it becomes code, something that raw code never put within their reach.

And it restores an accountability that informal assistance blurs. When code comes out of a conversation nobody keeps, there is nothing to refer back to when something goes wrong. When it comes out of an approved spec, the chain of decisions is written down and can be reviewed, discussed, and corrected. Chapter 16 shows that this traceability also has regulatory value when the product enters the high-risk field.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the shift of the core competence from code to specification.
- Apply the blueprint-and-building analogy to reason about the relationship between the spec and the code.
- Distinguish the spec as a contract, approved at gates and executed by agents, from a command document.
- Cite the academic formulation of the spec as a contractual substrate with the caution its authors ask for.
- Anticipate how the work of each role on the team changes, including the non-technical ones.

Common mistakes and antipatterns

Believing the code stops mattering. It has to be reviewed, tested, and validated. What changes is the causal direction, because the spec produces the code and not the other way around.

Specifying with the imprecision of traditional development. What a person would complete with common sense, a literal agent implements to the letter or invents.

Treating the spec as a one-way order. It works as a contract approved with judgment, and its value lies in that approval.

Reserving specification for technical profiles. The precision the spec demands is a competence of the whole team, and stakeholders are the ones who gain most from its readability.

Further reading

· Addy Osmani — How to write a good spec for AI agents

· Díaz, Gayoso, Cimmino, and Pérez — SDD for Agentic Software Engineering (arXiv, 2026)

· Skill Arena — Scrum in teams with AI (in Spanish)

· Martin Fowler — Exploring Gen AI (series)

BLOCK A · THE PARADIGM: WHY SDD

Chapter 3. SDD and agility: not the return of waterfall

CONSOLIDATING

SDD uses phases and gates, but differs from waterfall in scope, feedback, and reversibility, and the critique that denies it has names and deserves to be read.

Introduction

Any agile professional will think, sooner or later, that SDD sounds like waterfall. Sequential phases, approval gates, and documentation before code are the traits agility was born to fight. The objection is reasonable, and resolving it is what allows adopting SDD without betraying the principles. In this chapter we tackle it head-on, and we give the floor to those who hold it, because the critique has names and is still alive.

Figure 3. *Three axes that separate SDD from waterfall: the scope is the increment, feedback arrives at each gate, and going back costs what it costs to edit a document.*

3.1 What it shares with waterfall and what it does not

SDD shares sequentiality with waterfall, because requirements precede design and design precedes tasks. It differs on three axes, which are the ones that matter.

1. **Scope.** Waterfall applied the sequence to the whole project, with months per phase. SDD applies it to each increment, and the full cycle can be measured in hours or days.
2. **Feedback.** In waterfall it arrived at the end. In SDD it arrives at every approval gate, so that each cycle is a complete learning cycle.
3. **Reversibility.** Going back in waterfall was expensive. In SDD the phases before implementation are text documents, and changing them is trivial compared with rewriting code.

Applying the sequence to the whole project would indeed be waterfall. Applying it to a small increment, with review at every step and the possibility of redoing a document in minutes, is something else.

3.2 A new category of documentation

Traditional documentation was informative, because it passed knowledge between people, or administrative, because it satisfied process requirements. The Agile Manifesto

rightly questioned prioritizing it over working software. SDD introduces a third category, operational documentation, which is the spec the agent consumes directly to generate code. It does not inform a person or satisfy a formality, it is the input of the production process.

That is why the Manifesto's objection does not apply to it in the same way. The text says "working software over comprehensive documentation", and "over" does not mean "instead of". Documentation that contributes directly to working software is on the right side of that preference.

3.3 The critique by name

François Zaninotto, of Marmelab, published in November 2025 an article whose title is already a thesis, "Spec-Driven Development: The Waterfall Strikes Back". He does not stop at the resemblance. He lists specific failures he has seen in practice, such as huge documents, loss of the code's context, duplicated review work, and a false sense of security, and holds that the usefulness of the method is almost limited to new projects. As an alternative he proposes natural language development, which consists of breaking down into small, checkable pieces and iterating on failures.

From the agile world, Yuval Yeret has raised the question of whether SDD is a step forward or a step back for product development. His answer is nuanced, and the nuance is what is interesting for the reader of this guide. It depends on where the spec lives and on who writes it, which is exactly what chapter 15 develops.

Read the critique first-hand. The failures Zaninotto lists are the antipatterns of chapter 17 seen from outside. Whoever adopts SDD does well to know them before they appear.

3.4 Where the industry places it

The Thoughtworks Technology Radar keeps SDD in the "Assess" ring, without having promoted it to "Trial". It recommends assessing it when requirements are ambiguous, teams are distributed, or traceability is needed, and warns that it may not pay off for trivial problems. It is the position of an organization that uses it and does not consider it consolidated, and it matches the status of this card.

The ecosystem itself has taken the objection into its discourse. OpenSpec, one of the light tools in the field, states its philosophy in three phrases: fluid, not rigid; iterative, not waterfall; and brownfield-first. When a tool defines itself in opposition to a critique, the critique has done its job.

Properly understood, SDD can be read as the coherent application of agile principles when part of the team is agents. There is fast feedback at the gates, incremental delivery with one spec per increment, collaboration around a readable artifact, and response to change, because changing a spec is cheaper than rewriting code.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Articulate the three differences between SDD and waterfall: scope, feedback, and reversibility.
- Explain the category of operational documentation and why the Agile Manifesto's objection does not apply to it in the same way.
- Summarize Zaninotto's critique and the alternative he proposes, and situate Yeret's question.
- Situate the Thoughtworks Technology Radar's judgment on when SDD adds value and when it does not pay off.
- Argue for SDD as the coherent application of agile principles in teams with agents.

Common mistakes and antipatterns

Rejecting SDD for looking like waterfall. The resemblance is superficial. The differences in scope, feedback, and reversibility are the ones that matter.

Applying the sequence to the whole project. That would indeed be waterfall. SDD applies the sequence to each small increment.

Reading the Manifesto as a ban on documenting. It says "over", not "instead of". Operational documentation contributes directly to working software.

Dismissing the critique as resistance to change. Zaninotto describes real failures that appear when the method is applied badly. Knowing them is the best prevention.

Further reading

· Thoughtworks Technology Radar — Spec-driven development

· Marmelab — Spec-Driven Development: The Waterfall Strikes Back

· Yuval Yeret — Is Spec-Driven Development a step forward or back for product development?

· OpenSpec

· Agile Manifesto (in Spanish)

BLOCK A · THE PARADIGM: WHY SDD

Chapter 4. The principle of proportionality · ESTABLISHED

Adjusting the intensity of the process to the size of the problem; badly calibrated, SDD becomes a specification bureaucracy.

Introduction

SDD is not a form to be filled in mechanically. The principle that governs its sensible application is proportionality, which consists of adjusting the intensity of the process to the size of the problem. A minor defect does not need a four-phase spec, and a complex feature that affects several parts of the system does. In this chapter we will see the risk of over-application, the question that allows calibration, and the recent evidence that reinforces the principle from another angle.

4.1 The risk of over-application

Böckeler herself documented a revealing case. When asking an SDD tool to fix a small defect, the generated requirements document turned the task into four user stories with sixteen acceptance criteria. She described it as using a sledgehammer to crack a nut. Badly calibrated SDD becomes a specification bureaucracy that slows down instead of speeding up, and the solution consists of adjusting the intensity, not rejecting the method.

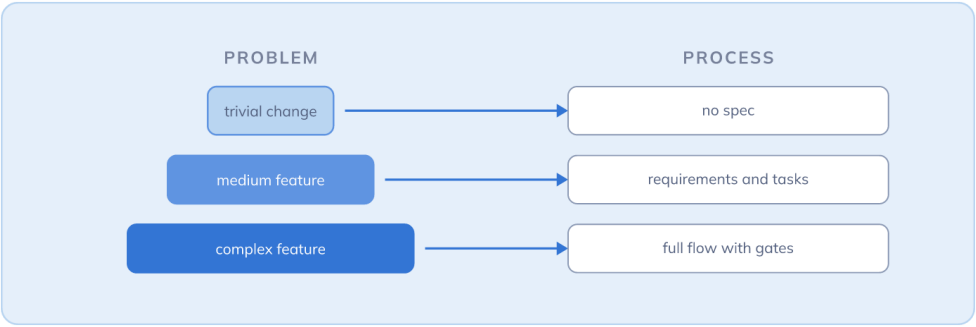


Figure 4. *The principle of proportionality: the intensity of the process is adjusted to the size of the problem.*

Over-application also has a hidden cost that does not show on the clock. When the team learns that any change triggers the full flow, it starts avoiding the flow, and with it the gates that did protect. Excessive rigidity does not produce rigor, it produces shortcuts.

4.2 The calibration question

The practical heuristic is a single question. What happens if the agent makes the wrong decision at this point. If the answer is “I fix it in two minutes”, that point does not need specifying. If the answer is “I lose hours or introduce a subtle defect”, it does.

SDD adds net value when the change is substantial, there is ambiguity, there is risk, more than one person or agent is involved, or the code has to be maintained over

time. When none of those conditions holds, vibe coding is still the right tool. A single-use script, a prototype for a demo, or a one-line change do not deserve the full flow.

4.3 The gradient the evidence shows

The evidence accumulated during 2026 reinforces the principle from another angle. The DORA report describes that the return on AI is high in new, simple work and much lower in legacy, complex code, so the intensity of the process cannot be the same at both extremes. Where the AI gets it right almost alone, heavy specification is a cost with no return. Where the AI stumbles over the system's history, the spec is what keeps the stumble from reaching production.

The principle, therefore, does not only protect from excess. It also points to where to invest. Chapter 9 develops the case of existing code, which is where proportionality leans toward specifying more.

4.4 Knowing when not to use SDD

Knowing when not to apply SDD is part of mastering it. Agile wisdom is contextual, and there is no practice that is always good or always bad. The principle of proportionality reappears in the calibration of the spec, in chapter 10, and in the team antipatterns, in chapter 17, because most failures of the method are calibration failures.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State the principle of proportionality and apply it to decide the intensity of SDD for a given change.
- Use the calibration question, the cost of a wrong decision, to decide what to specify.
- Identify the conditions under which SDD adds net value as opposed to those where it does not.
- Explain the gradient between new work and legacy code that the evidence describes, and what it implies for the intensity of the process.
- Recognize over-application as a failure mode and not as rigor.

Common mistakes and antipatterns

Applying SDD with the same intensity to everything. The trivial defect and the complex feature do not deserve the same process. Equating them generates bureaucracy and, afterwards, shortcuts.

Confusing exhaustiveness with quality. More phases and more criteria do not make better SDD. They are often the sledgehammer to crack a nut.

Forgetting that vibe coding still has its place. For scripts, prototypes, and trivial changes, specifying is a cost with no return.

Further reading

· Böckeler — Understanding Spec-Driven Development

· DORA — ROI of AI-assisted Software Development

· Scrum Manager — Scrum in the age of AI (in Spanish)

BLOCK B · THE METHOD: PHASES, WRAPPERS, AND GATES

Chapter 5. The flow: the four-phase core and its two wrappers · ESTABLISHED

Requirements, design, tasks, and implementation are still the core; the constitution precedes it and convergence closes it.

Introduction

Regardless of the tool, the method has a stable backbone, a sequence of four phases with one document per phase. Kiro calls them requirements, design, and tasks; Spec Kit, specify, plan, and break down into tasks. The underlying structure is the same because it reflects the minimum logical sequence for going from an intention to working software when the builder is an agent.

What has changed since the first edition of this guide is what surrounds that core. The reference implementations have settled two wrappers, one before the requirements and another after the implementation, and it is no longer accurate to say that all of them converge on four phases. In this chapter we will see the core, the wrappers, and the principle that governs the whole.

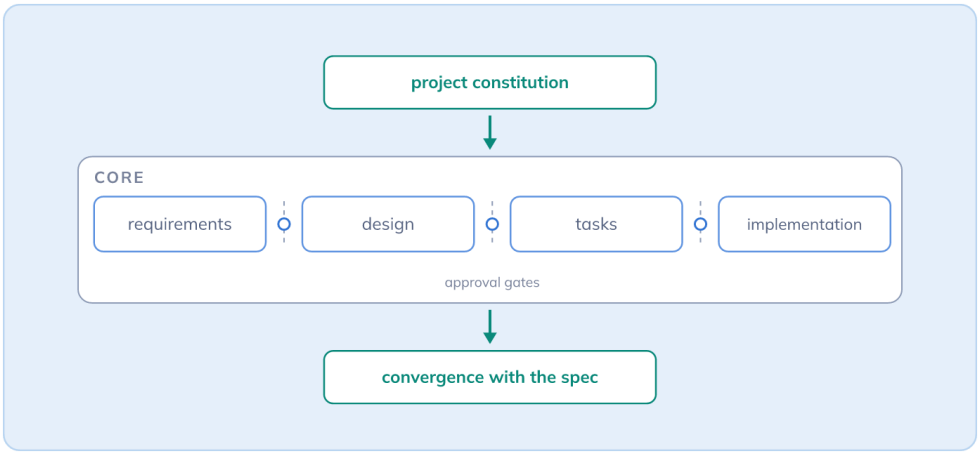


Figure 5. *The four-phase core and its two wrappers: the constitution precedes the requirements, convergence closes the implementation, and review is concentrated at the gates.*

5.1 The four phases and their deliverables

1. **Requirements.** What is built, from the perspective of the user and the business. It is not technical. It describes the expected behavior, the conditions, and how it will be verified.

2. **Design.** How it is built. Here there are technical decisions, such as patterns, affected components, and data structure, consistent with the code that already exists.
3. **Tasks.** The design is broken down into atomic implementation units, each with its structured prompt, as chapter 8 develops.
4. **Implementation.** The agents execute the tasks and produce code, tests, and commits. It is the phase in which the person intervenes least, if the previous ones were done well.

Each phase produces a document, and each document is reviewed before moving on to the next. That review is what chapter 7 calls an approval gate.

5.2 The two wrappers

GitHub Spec Kit, the most adopted implementation, published its version 1.0 in August 2026 and documents a six-step process. The first establishes the project principles in a constitution. Then come the four of the core. The sixth checks that the implementation converges with what was specified. To these are added quality commands to clarify requirements, analyze the consistency between documents, and generate checklists.

The first wrapper, the constitution, records what is decided once for the whole project, and has enough substance to deserve chapter 6. The second, convergence, is the final review that compares the result with the spec, and answers the question no intermediate document could answer, whether what was built is what was agreed.

Kiro arrives at the same place by another path. It offers requirements-first or design-first flows, depending on whether you want to start from the expected behavior or from an already mature technical idea, and an optional deep analysis of the requirements before moving on to design. The convergence of the two reference tools on the same shape, a wrapped core, is what makes it possible to describe the method without tying it to either of them.

5.3 The guiding principle

The principle that governs the flow comes down to reviewing at the phase gates, not during implementation. In conversational work, the programmer approves micro-decisions one by one, which produces approval fatigue. SDD concentrates human review where the information is worth most and the cost of correction is lowest, between phases. Once the tasks are approved, the implementation runs with minimal intervention, because the contract is already clear.

It is not all or nothing. The four phases and the two wrappers represent the full flow, and not every problem needs them at the same depth. A medium-sized feature can carry requirements and tasks with a light design. Calibrating the intensity is the principle of proportionality of chapter 4 applied to the flow.

What you should be able to do

Having mastered this chapter, a professional is able to:

- List the four phases of the core and the deliverable of each one.
- Explain the two wrappers, the constitution and convergence, and why they force the description of the flow to be reformulated.
- Describe how Spec Kit and Kiro arrive at the same shape by different paths.
- Apply the guiding principle, review at the gates and not during implementation, and reason why it reduces approval fatigue.
- Calibrate the depth of each phase according to the size of the problem.

Common mistakes and antipatterns

Mixing the what and the how. Technical decisions in the requirements phase are one of the fastest paths to over-specification.

Reviewing during implementation instead of at the gates. It reintroduces the approval fatigue SDD seeks to remove.

Considering the increment finished when implementation ends. Without the convergence step nobody has checked that what was built matches what was specified, and drift starts there.

Applying the phases at equal depth every time. The flow is calibrated. It is not a mandatory form.

Further reading

· GitHub Spec Kit

· Kiro — specs

· Kiro — spec best practices

· Böckeler — Understanding Spec-Driven Development

BLOCK B · THE METHOD: PHASES, WRAPPERS, AND GATES

Chapter 6. The project constitution · CONSOLIDATING

What is decided once for the whole project, as opposed to what is decided in each increment, and the artifact that preserves it between agent sessions.

Introduction

In the first edition of this guide, the constitution was half a sentence inside the boundaries chapter. Today it is the first step of the method in the reference implementation, it has documented content, and it is taught as the first competence in training in the field. In this chapter we will see what it is, what it contains, where it lives, and why it preserves judgment between different sessions of an agent that remembers nothing from one to the next.

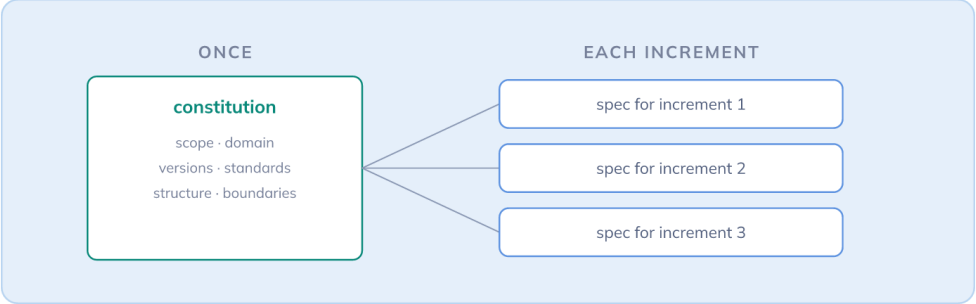


Figure 6. *Once versus each increment: the constitution fixes what holds for the whole project and the specs decide what belongs to each increment.*

6.1 What it is and what problem it solves

The constitution records what is decided once for the whole project, as opposed to what is decided in each increment. It is the founding rulebook that aligns the development life cycle and persists above individual specs. It solves a problem different from the spec's, which says what is built, and from the boundaries', which say what the agent may do. The constitution says which project we are in and by what rules we work in it.

The problem it solves is the agent's lack of memory. Each session starts from zero, and without a document that fixes the context, structural decisions are made anew in every conversation, with different results. The constitution is what gives continuity to judgment and reduces what training in the field calls cognitive debt, the effort of explaining the same thing again every time.

6.2 What it contains

Thoughtworks teams working with SDD on existing codebases have documented what a constitution that really works contains.

- **Project scope.** What the system is and is not, so the agent does not expand it on its own.

- **Domain context.** The business vocabulary and the rules a newcomer would need to know.
- **Technology versions.** Languages, frameworks, and libraries with their versions, so that others are not introduced.
- **Code standards.** Naming, testing, and style conventions.
- **Repository structure.** The chosen architecture, for example hexagonal or layered, and where everything goes.

To that list are added the project boundaries, the general rules about what the agent always does, asks about first, or never does, which chapter 12 develops. And it should be short. A constitution that tries to contain everything falls into the curse of instructions of chapter 13.

6.3 Where it lives and how it is maintained

In Spec Kit the constitution is created with the first command of the flow and lives in the repository, versioned with the code. In the rest of the ecosystem the usual support is an agent instructions file at the root of the project, with AGENTS.md as the most widespread format, as we will see in chapter 12. What matters is not the name of the file, but that it is where the agent reads it when starting and where the team reviews it when a structural decision changes.

It is maintained like any architecture decision. It changes little, it changes deliberately, and every change is reviewed, because it affects everything built afterwards. An increment that needs to bypass a rule of the constitution is a signal that the rule, or the increment, deserves a conversation.

A constitution is not a long spec. What holds for a single increment goes in its spec. What holds for all of them goes in the constitution. Mixing them fattens the constitution and leaves the spec out of date.

6.4 Why it is taught first

The DeepLearning.AI course on spec-driven development with coding agents teaches writing the constitution as the first competence, before feature specs, for three reasons. It preserves context between agent sessions, improves fidelity to intent, and reduces cognitive debt. It is also the natural order of the work. When the constitution exists, each spec can be shorter, because it does not repeat what is already decided.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Distinguish what is decided once for the whole project from what is decided in each increment.
- List the contents of a useful constitution and justify each element.
- Place the constitution in the flow of the method and in the project repository.

- Maintain the constitution as an architecture decision, with deliberate and reviewed changes.
- Explain why writing the constitution before the specs shortens them and preserves judgment between sessions.

Common mistakes and antipatterns

Putting into the constitution what belongs to an increment. The constitution fattens, the spec goes out of date, and the agent receives rules that do not apply to its task.

Having no constitution and trusting the memory of the conversation. Each session decides the architecture again, and structural drift appears within weeks.

Writing it once and never looking at it again. A constitution that contradicts the team's current decisions is worse than none, because the agent obeys it.

Further reading

· Thoughtworks Technology Radar — GitHub Spec Kit

· GitHub Spec Kit

· DeepLearning.AI — Spec-Driven Development with Coding Agents

BLOCK B · THE METHOD: PHASES, WRAPPERS, AND GATES

Chapter 7. Approval gates · ESTABLISHED

The points where the person exercises judgment between phases, and the recent counterpoint: a badly sized gate stops protecting and starts slowing down.

Introduction

If the phases are the body of SDD, the approval gates are its nervous system. They are the points where the person exercises judgment, problems are detected before they become expensive, and the team aligns. There are three main gates, one between each pair of phases of the core, and the convergence review at closure. In this chapter we will see what is reviewed at each one, the economic logic that justifies them, and the counterpoint the evidence of 2026 adds.

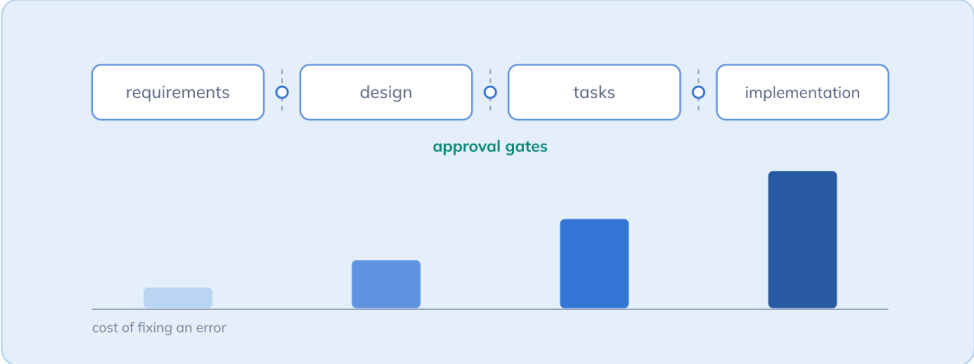


Figure 7. *Three gates and the cost of an error: the later an error is detected, the more it costs to fix, and that is why review is concentrated between phases.*

7.1 What is reviewed at each gate

- **Gate 1, from requirements to design.** Whether we are solving the right problem. Stories that reflect real needs, verifiable criteria, manageable scope, and detected implicit requirements, such as security, performance, or accessibility.
- **Gate 2, from design to tasks.** Whether the design is viable and coherent. It respects the patterns of the existing code, its decisions are justified, and its dependencies and risks are identified.
- **Gate 3, from tasks to implementation.** Whether these tasks, in this order, produce the design. Coverage without gaps, correct dependencies, precise prompts, and assessable success criteria.
- **Convergence.** Whether what was built matches what was specified. It is the final wrapper of chapter 5, and the only review that looks at code.

7.2 The cost of an error by phase

There is an economic reason behind the gates. The cost of fixing an error grows steeply with each phase it passes through. An ambiguous acceptance criterion is fixed in requirements by rewriting a sentence. If it reaches design, the solution has to be

rethought. If it reaches tasks, the decomposition has to be redone. If it reaches implementation, code has to be discarded and regenerated. Every minute spent reviewing a requirement saves hours of reimplementation.

7.3 Reviewing a spec versus reviewing code

The first three gates review text documents, and that has three implications. Accessibility, because more people on the team, including those who do not program, can take part. Speed, because reading a spec is faster than reading code. And focus, because the question is whether this is what we want, separated from whether it is well implemented, which is reserved for convergence. Separating the two questions makes each review more effective.

Approval is an act, not a formality. A gate crossed without reading detects nothing and gives a false guarantee. Chapter 17 calls it specification theater.

7.4 The counterpoint: when the gate slows down

The evidence of 2026 adds something the first edition of this guide did not have. The DORA report identifies manual review as the bottleneck that saturates when the speed of code generation rises. Gates concentrate review, and for that very reason they are the point where the system jams if review capacity does not grow at the pace of generation capacity. A badly sized gate stops protecting and starts slowing down.

Kief Morris has explored the same question from another angle, that of how people and agents are distributed within the loops of software engineering. His question is at which points of each loop the person contributes a judgment the agent does not have, and at which their intervention only adds latency. It is the question to ask when designing the gates of a specific team, and its answer changes with accumulated trust, as we will see with the boundaries in chapter 12.

The practical answers to the jam do not involve removing the gate. They involve delegating approval to whoever has judgment, accepting asynchronous approval, sizing the specs so the review is short, and automating the part of the check that requires no judgment, such as the consistency analysis step that Spec Kit adds between documents.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Describe what is reviewed at each of the three gates and at the convergence review.
- Explain the economic logic: the cost of fixing grows steeply by phase.
- Justify why reviewing specs is more accessible, faster, and more focused than reviewing code.
- Recognize a saturated gate and apply responses that relieve it without removing it.
- Ensure that approval is a deliberate and explicit act.

Common mistakes and antipatterns

Skipping gates under schedule pressure. It is where the error is cheapest to fix. Skipping them moves the cost to the expensive phases.

Approving without reading. A gate that is not really exercised detects nothing, and it is worse than not having it, because it gives a false guarantee.

Mixing "is this what we want?" with "is it well implemented?". Separating the two questions is what makes each review effective.

Responding to the gate jam by removing it. The bottleneck is resolved with delegation, asynchrony, shorter specs, and automated checks, not by giving up judgment.

Further reading

· Kief Morris — Humans and Agents in Software Engineering Loops

· DORA — ROI of AI-assisted Software Development

· GitHub Spec Kit

BLOCK B · THE METHOD: PHASES, WRAPPERS, AND GATES

Chapter 8. Atomic tasks, waves, and atomic commits ·

CONSOLIDATING

Breaking the design down into small, verifiable units, running them in waves, and recording one commit per task.

Introduction

The tasks phase transforms the complexity of a large change into a series of small, manageable changes. The complexity does not disappear, but it becomes a sequence of pieces with clear boundaries. In this chapter we will see the anatomy of an atomic task, its organization into waves, the practice of the atomic commit, and the limit beyond which decomposing stops helping.

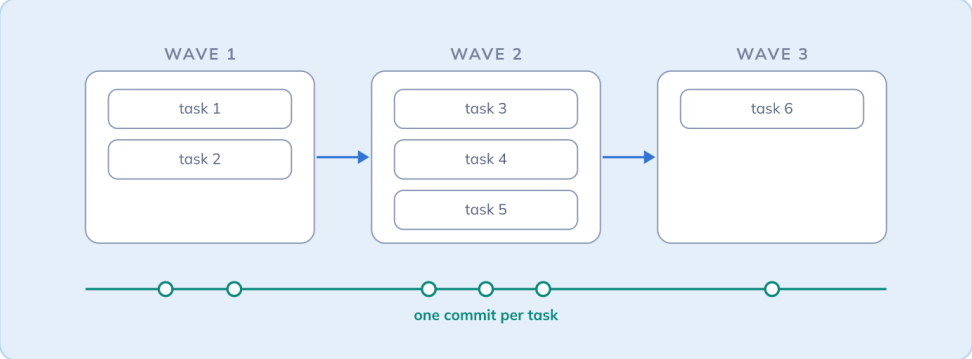


Figure 8. *From task to commit: independent tasks run in parallel within a wave, waves are chained, and each task closes with its own commit.*

8.1 Anatomy of an atomic task

A well-defined task normally affects one to three files. If it needs more, it can probably be broken down. It includes a structured prompt with four fields: the role the agent assumes, the specific task, the constraints it must respect, and the success criteria with which it will be verified. And it can be verified in isolation, because its result can be assessed without having completed the others.

The four fields are not bureaucracy. The role fixes the point of view, the task fixes the scope, the constraints are the task boundaries of chapter 12, and the success criteria are what allows the convergence gate to check the task without rereading all the code.

8.2 Dependencies and waves

Tasks are not necessarily sequential. Those that do not depend on each other are grouped into the same wave and run in parallel, each by a subagent with a clean context, and the waves run in sequence. This model takes advantage of the agents' ability to work in parallel without the context of one task contaminating another's, and it is the orchestrator-and-workers pattern described in Anthropic's guidance on effective agents.

The order of the waves is dictated by the dependencies of the design. First what creates the structures the others depend on, then what uses them, finally what integrates. A task waiting for another in its own wave is a signal that the decomposition did not respect the dependencies.

8.3 Atomic commits

Each task generates an independent commit, and that has three consequences. Granular reversibility, because a defective task is reverted without affecting the others. Traceability, because each commit is linked to a task, a design, and some requirements, which is the chain chapter 16 shows to be useful before a regulator. And effective bisection, because it is possible to identify precisely which task introduced a defect.

The risk of excessive decomposition. If the prompt that describes a task is longer than the code it will produce, the decomposition has gone too far. The right size is the one that lets the agent work with full context and the person review the result at a glance, in minutes and not in hours.

Granularity is still a matter of professional judgment more than of closed consensus, and that is why the card keeps its status. What is settled is the direction. Small pieces, verifiable separately, and with their own record in the history.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Define an atomic task by its limited scope, its four-field prompt, and its isolated verifiability.
- Organize tasks into waves according to their dependencies, taking advantage of the parallelism of subagents with a clean context.
- Justify the atomic commit by its granular reversibility, its traceability, and its usefulness for bisection.
- Recognize the right size of a task and avoid both overload and excessive decomposition.

Common mistakes and antipatterns

Tasks that touch too many files. They reduce the precision of the implementation. It is worth decomposing further.

Excessive decomposition. If the prompt is longer than the code it produces, the cost exceeds the benefit.

Commits that mix several tasks. They lose granular reversibility and traceability, and make bisection useless.

Further reading

· Anthropic — Building effective agents

· GitHub Spec Kit

BLOCK B · THE METHOD: PHASES, WRAPPERS, AND GATES

Chapter 9. Specifying over an existing codebase (brownfield) · CONSOLIDATING

In a project with history the spec describes the delta, not the system, and fixes what exists so the agent does not overwrite it.

Introduction

The literature on SDD was born thinking of projects that start from scratch. The reality of most teams is another, work on existing codebases, often large and inherited. In those environments, which the industry calls brownfield, the first phase begins by looking at the code that is already there, before writing a single requirement. And it is where SDD adds most value, because an agent that ignores the existing context causes more damage than one that works on a blank canvas.

In this chapter we will see what is analyzed before specifying, how what exists is fixed, why the new case stops being new right away, and a note on the name this guide used before for this analysis.

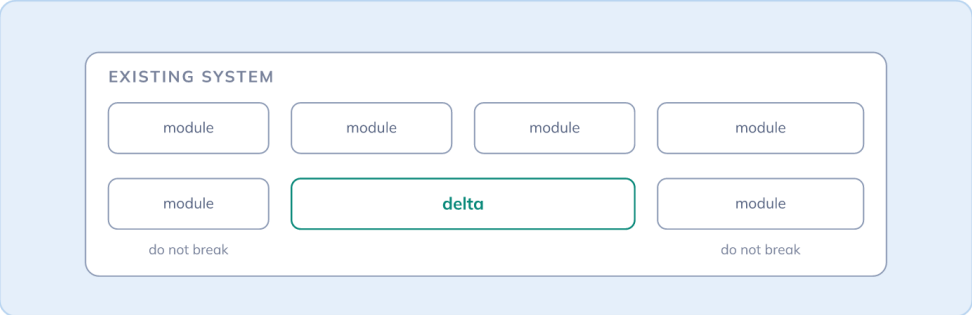


Figure 9. *The delta over what exists: the spec describes the change, fixes what must not be broken, and leaves the whole system outside the contract.*

9.1 The delta, not the system

In a project with history, the spec does not describe the whole system, it describes the delta. Before writing a requirement it is worth answering three questions. Which files will be affected by the change. Which patterns, conventions, and dependencies already exist and must be respected. And what must not be broken, that is, which side effects the change could have on other parts of the system. The answers come from a search of the code, structural and semantic, which the agent can do and the team reviews.

That prior analysis has a preventive function. It stops requirements from asking for something that duplicates what exists, contradicts established patterns, or ignores

dependencies. If the system already has an email-sending mechanism, the requirement should not ask to “build an email system”, but to “extend the existing mechanism for the new channel”. It is the difference between designing on known ground and designing on imagined ground.

9.2 Fixing what exists

The second move consists of putting in writing what exists so the agent does not overwrite it. A complete specification of the system is impracticable at scale, and unnecessary besides. What is needed is to fix the pieces the change touches and those it must not touch. Kiro offers to generate specs of what is already built, so that legacy code is described in the same language as new code. OpenSpec goes further and declares itself outright brownfield-first, with the separation between current truth and proposed changes that chapter 14 describes.

Field experience confirms the emphasis. Thoughtworks teams report using SDD mostly in brownfield environments, precisely where unclear intent and hidden assumptions generate rework. The method performs best where the agent has most to respect.

Inverted proportionality. Chapter 4 showed that the return on AI is high in new, simple work and low in legacy, complex code. Legacy code is exactly where the spec pays off most, because it is where the agent errs most.

9.3 When a new project stops being new

In a new project, the analysis of existing code does not apply in the first iteration, but it does from the second feature. As soon as there is code, there is context to respect, and the greenfield lasts as long as the first increment. The analysis also feeds the design phase, which lists which files will be created and which modified, and the constitution, which fixes the patterns the following specs will take for granted.

9.4 A note on terminology

The first edition of this guide called the prior analysis this chapter describes an “Impact Report”. On reviewing the topic we found that the name is not used in the field or in the tools’ documentation, and this edition withdraws it. The practice is the same, and we describe it by what it does, specifying the delta and fixing what exists, instead of by a name that existed only in our material.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Answer the three questions of the prior analysis on existing code: what is affected, what must be respected, and what must not be broken.
- Specify the delta of a change without trying to describe the complete system.

- Fix in writing what exists so the agent does not overwrite it, with the tools that make it easier.
- Recognize when a new project starts behaving like an existing one.
- Explain why legacy code is where the method pays off most.

Common mistakes and antipatterns

Writing requirements without looking at the existing code. It leads to asking for things that duplicate or contradict what is already there.

Trying to specify the whole system. It is impracticable at scale and not needed. The spec describes the delta and fixes the pieces the change touches.

Assuming a new project when there is already code. Most real work is on legacy code, and ignoring it is the main source of side effects.

Treating the prior analysis as a formality. If it is not really reviewed, it does not fulfill its preventive function.

Further reading

· OpenSpec

· Kiro — spec best practices

· Thoughtworks Technology Radar — GitHub Spec Kit

BLOCK C · WRITING GOOD SPECS

Chapter 10. The smart spec, not the long one · ESTABLISHED

A good spec covers just enough to guide the agent without overwhelming it; minimal does not mean short, and the prompt is treated as an artifact that is maintained.

Introduction

If one sentence sums up what separates a good spec from a bad one, it is that minimal does not necessarily mean short. A good spec does not skimp on detail when it matters and does not add information that does not contribute to the result. In this chapter we will see the principles that have settled for writing it, the areas an effective spec covers, the further turn that structured-prompt-driven development brought in 2026, and how the detail is calibrated.

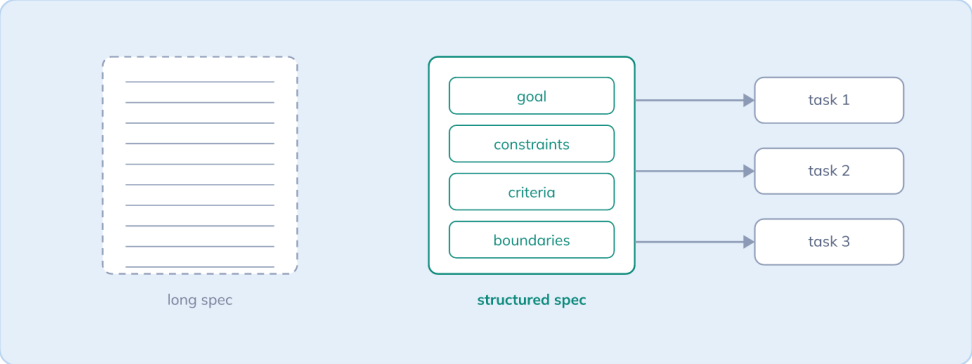


Figure 10. *The smart spec, not the long one: each task receives the portion of the spec that belongs to it, not the whole document.*

10.1 Osmani's five principles

Addy Osmani distilled five principles that are still the practical frame of reference, and he has kept developing them in his work on agentic engineering and in his book *Beyond Vibe Coding*.

1. **Start with the high-level vision and let the AI elaborate the details.** The person provides the what and the why, the agent proposes the how in detail, and the person validates. The spec is created between the two, not handed over finished.
2. **Structure the spec as a professional document.** Agents process structured information better, with sections, lists, and explicit criteria.
3. **Split the tasks into modular prompts.** Feed the agent the portion of the spec relevant to its task and not the whole spec, as chapter 13 develops.
4. **Build in self-verification, constraints, and expert knowledge.** Instruct the agent to compare its result with the spec, anticipate where it may go wrong, and pour in the domain knowledge that only the professional contributes.
5. **Treat it as a living document.** Update it when decisions are made or new information is discovered, because an outdated source of truth is worse than none, as we will see in chapter 14.

10.2 The areas an effective spec covers

GitHub's analysis of more than two thousand five hundred agent configuration files, which Osmani takes as backing, identified six areas present in effective specs: the project commands, the testing strategy, the project structure, the code style, the git workflow, and the boundaries. A good part of those areas belong to the constitution of chapter 6, and that is why a well-made constitution makes all the specs that come after it shorter.

10.3 The prompt as an artifact: SPDD and the REASONS canvas

In April 2026, Wei Zhang and Jessie Jie Xia, of Thoughtworks, published in Martin Fowler's series their proposal of structured-prompt-driven development (SPDD). Their thesis is that the prompt is a maintained, versioned, and governed artifact, and not something generated once and discarded. It is the same idea that SDD applies to the spec, carried to the piece the agent receives in each task.

Their practical contribution is a seven-part template, the REASONS canvas, whose initials name the requirements, the entities, the approach, the structure, the operations, the norms, and the safeguards. It dialogues well with Osmani's principles. The five principles say how to think about the spec, and the canvas offers a concrete template so that none of the parts the agent needs is missing. It is the proposal of a consultancy, from this very year and with no measured adoption, and that is why we present it as chapter material and not as an established practice.

10.4 Proportionality applied to the spec

The principle of chapter 4 returns here as a heuristic for detail. The detail of the spec is adjusted to the complexity of the task, with the same calibration question. What happens if the agent makes the wrong decision at this point. If it is fixed in two minutes, it does not need to be in the spec. If it costs hours or introduces a subtle defect, it does.

Curate, do not accumulate. Adding detail to cover more cases has diminishing and then negative returns, because of the curse of instructions of chapter 13. The smart spec selects what the agent needs to know for that task.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State and apply Osmani's five principles for writing a spec.
- Cover the six areas of an effective spec, distinguishing those that belong to the constitution from those specific to each increment.
- Explain the thesis of the prompt as a maintained artifact and use the REASONS canvas as a template for a task spec.
- Calibrate the detail of the spec with the question about the cost of a wrong decision.

- Distinguish a smart spec from a merely long one.

Common mistakes and antipatterns

Adding detail to cover more cases. It has diminishing and then negative returns. The smart spec curates, it does not accumulate.

Handing the complete spec to every task. It overloads the context. Only the relevant portion should be fed.

Generating the prompt and discarding it. The prompt of a task is an artifact that is reviewed and reused, and discarding it forces the team to rediscover in every session what had already been decided.

Writing the spec once and forgetting it. An outdated spec misleads whoever consults it.

Further reading

· Addy Osmani — How to write a good spec for AI agents

· Addy Osmani — Agentic engineering

· Zhang and Xia — Structured-Prompt-Driven Development

· AGENTS.md

BLOCK C · WRITING GOOD SPECS

Chapter 11. The EARS notation for acceptance criteria ·

CONSOLIDATING

Patterns that structure natural language just enough to remove ambiguity without losing readability.

Introduction

Several SDD tools use the EARS notation (Easy Approach to Requirements Syntax) to structure acceptance criteria. Alistair Mavin developed it at Rolls-Royce and presented it in 2009, and organizations in the aerospace and systems engineering sectors have adopted it. It is not complex. It is a set of keyword-based patterns that reduce the ambiguity of natural language without sacrificing readability, and that makes it useful for agents.

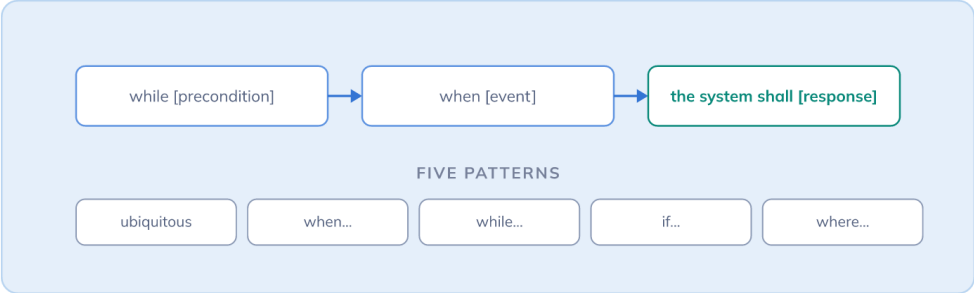


Figure 11. *The EARS template: precondition, event, and system response, with five patterns that cover most requirements.*

11.1 The template and the five patterns

The basic template fits three pieces together: “While [precondition], when [event], the system shall [response]”. On it, EARS defines five patterns that cover most requirements.

- **Ubiquitous.** A permanent property of the system. “The system shall encrypt all passwords with a configurable-cost algorithm”.
- **Event-driven** (when...). It triggers when something happens. “When a version of a document is published, the system shall queue a notification in under five seconds”.
- **State-driven** (while...). Active while a condition holds. “While the user has do-not-disturb mode enabled, the system shall hold notifications”.
- **Unwanted behavior** (if...). It handles errors and abnormal situations. “If the password is entered wrongly three times, the system shall lock the account for fifteen minutes”.
- **Optional** (where...). Functionality conditioned on configuration. “Where the organization has enabled the second factor, the system shall request a code after the password”.

The patterns combine. A requirement can be state-driven and event-driven at the same time, and the template allows it without losing clarity.

11.2 Why it works with agents

Agents respond well to structured requirements with consistent patterns, because the fixed form reduces the room for interpretation. A requirement in EARS is less likely to be misread because the structure removes the most common ambiguities of natural language, such as the implicit subject, the condition that may be prior or simultaneous, or the response with no deadline.

The notation is not mandatory. It is especially useful for making acceptance criteria verifiable, that is, so that looking at the result it can be determined whether they are met. “The notification is fast” is not verifiable. “The notification reaches the channel in under thirty seconds in 99% of cases” is. EARS pushes toward the second form. Its integration into Kiro, which uses it to generate the criteria of its requirements documents, has brought back to the foreground a notation that had spent more than a decade in a very specific sector.

Use it where it reduces ambiguity. EARS is applied where the criterion is important enough to deserve the fixed form. Forcing it onto every line of the spec turns a help into an obligation, and its adoption status reflects that it is used that way, with judgment.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Recognize and apply the EARS template and its five patterns to real acceptance criteria.
- Choose the appropriate pattern according to the nature of the requirement.
- Explain why structured requirements reduce the agent’s misinterpretation.
- Write verifiable criteria instead of vague ones.

Common mistakes and antipatterns

Vague acceptance criteria. “Fast”, “intuitive”, or “secure” without a threshold are not verifiable for a person or for an agent.

Forcing EARS where it adds nothing. It is a tool and not an obligation. It is used where it genuinely reduces ambiguity.

Confusing the pattern. Using “when” for a permanent property or “while” for a one-off event distorts the criterion.

Further reading

· Alistair Mavin — EARS

· Kiro — spec best practices

BLOCK C · WRITING GOOD SPECS

Chapter 12. The boundaries system: Always / Ask First / Never

CONSOLIDATING

A three-level delegation framework, in two scopes, with an agent instructions format that has become almost a standard.

Introduction

The most effective specs do not use a flat list of rules, but a three-level system that gives the agent a clear decision framework about when to act, when to ask, and when to stop. It works like delegation to a competent professional, with the difference that with an agent the rules must be explicit, because it lacks the common sense that would let it infer them. In this chapter we will see the three levels, why they work and how they evolve, the two scopes they operate in, and the format the ecosystem has converged on for writing them.

Figure 12. *Three levels of delegation: what the agent always does, what it asks about first, and what it never does, with the middle level moving as trust grows.*

12.1 The three levels

- **Always.** Actions the agent performs without asking. Running the tests before a commit, following the naming conventions, logging errors, including error handling at every entry point.
- **Ask First.** Actions that could be right but require approval because of their impact. Modifying the database schema, adding new dependencies, changing the continuous integration configuration, modifying the public API.
- **Never.** Absolute red lines. Committing secrets, editing dependency directories, deleting a failing test without approval, modifying the production configuration.

12.2 Why it works and how it evolves

The system grants autonomy where it is safe, because Always frees the agent from consulting about micro-decisions. It creates checkpoints where they are needed, because Ask First concentrates human intervention on high-impact matters. And it sets unequivocal red lines, because Never eliminates whole categories of error. The three levels make the agent’s autonomy safe in the implementation phase and allow that autonomy to be raised without raising the risk.

Moreover, the system is not static. As the team gains trust, something that is Ask First today can move to Always, and something believed harmless can rise to Ask First after an incident. It is gradual autonomy, analogous to how a person delegates more and more to a professional who shows judgment, and it is the practical answer to Morris's question of chapter 7 about where the person should come into each loop.

12.3 Two scopes: project and task

Boundaries operate at two levels. In the project constitution, of chapter 6, they set the general rules that hold for everything that gets built. In each task they add the specific constraints of that implementation, which is the third field of the four-field prompt of chapter 8. The combination gives global consistency and local precision, and the distinction avoids the most common mistake, which is writing into the constitution constraints that hold only for one task.

12.4 AGENTS.md, the format that has become standard

The ecosystem's support is no longer a matter for each tool. AGENTS.md, an instructions file at the root of the repository, has become the closest thing to a universal standard for agent instructions. Some twenty coding tools read it natively, more than sixty thousand projects have adopted it, and its specification has been stewarded since December 2025 by the Agentic AI Foundation, within the Linux Foundation.

For the team, the practical consequence is that the constitution and the project boundaries have a home that any agent understands, regardless of the vendor. Writing them once in that format avoids maintaining different versions for each tool, and it is the second signal the first edition of this guide left under watch and which has resolved in the positive.

The format does not replace judgment. That the file exists and every tool reads it says nothing about whether its rules are the right ones. The content is decided by the team, and it is reviewed with the same deliberation as the constitution.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Classify actions into Always, Ask First, and Never with judgment.
- Explain why the three-level system beats a flat list of rules.
- Manage the evolution of the boundaries as gradual autonomy, in both directions.
- Distinguish project boundaries, in the constitution, from task boundaries, in the prompt.
- Situate AGENTS.md as the agent instructions format and explain what it resolves for the team.

Common mistakes and antipatterns

A flat list of prohibitions. Without levels, the agent does not know what it may do alone, what to ask about, and what to avoid.

Boundaries that never evolve. Keeping everything in Ask First indefinitely wastes the trust earned and saturates with queries.

Confusing the project scope with the task scope. “Never commit secrets” is project-wide. “Do not touch the existing entry points” belongs to the specific task.

Keeping one instructions file per tool. With a common format, the versions fall out of sync needlessly and the agent receives different rules depending on who invokes it.

Further reading

· AGENTS.md

· Agentic AI Foundation

· Addy Osmani — How to write a good spec for AI agents

BLOCK C · WRITING GOOD SPECS

Chapter 13. The curse of instructions (instruction overload) ·

ESTABLISHED

More context degrades the ability to retrieve what matters; the countermeasure is to give each task its portion and to systematize reusable context.

Introduction

One of the most relevant findings for SDD comes from research on the models themselves. Adding instructions to an agent does not improve its behavior indefinitely. Past a certain point, more context degrades the ability to retrieve what matters, and what sits in the middle of a long window is what is remembered worst. In this chapter we will see the finding and its body of evidence, the named discipline that has been born from it, its implications for spec design, and the practice with which teams resolve it.

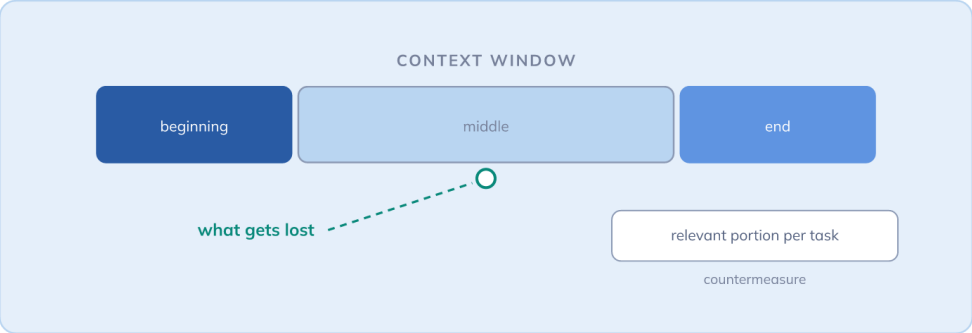


Figure 13. *The curse of instructions: what sits in the middle of a long context window is what is remembered worst, and the countermeasure is to give each task its relevant portion.*

13.1 The finding and its evidence

The work known as the curse of instructions showed a regular result. As instructions pile up in a prompt, the model's performance in following each of them drops predictably. Models follow the first ones reliably and start breaking the last ones as the list grows. With ten detailed rules, the agent may follow the first and overlook the last. With fifty, it will read the spec only partially.

The phenomenon also has a body of research of its own. The work on lost in the middle showed that models retrieve information at the beginning and end of the context better than information in the center. And research on context rot has found that retrieval ability decays as the number of tokens grows in all large models, regardless of the nominal size of their window. That double settling, of the name and of the evidence, is what makes this edition of the map consider established what the previous one gave as consolidating.

13.2 From prompt engineering to context engineering

The industry's response has been to give a discipline a name. Context engineering is now distinguished from prompt engineering, because the question changes. It no longer asks

how to word the instruction. It asks what should occupy the window at each moment and what should be evicted, compressed, or delegated to another agent. Anthropic's guidance on context engineering for agents frames it as the management of a scarce resource, the model's attention, which is administered as any budget is administered.

13.3 Implications for spec design

The consequences for whoever writes specs are four.

- **A smarter spec, not a longer one.** Adding an instruction can worsen compliance with the earlier ones.
- **Decompose instead of accumulating.** Five specs of ten instructions, assigned to the tasks where they are relevant, perform better than one of fifty. It is what the tasks phase of chapter 8 does.
- **Prioritize.** If it cannot be split further, the most important instructions, such as security ones, go first, where the model gives them more weight.
- **Separate levels.** The boundaries system of chapter 12 creates categories of different urgency instead of a flat list of equal weight.

There are signs that betray an overloaded spec. The agent ignores written constraints, follows what is at the beginning and not what is at the end, improves when the spec is reduced, or gets isolated tasks right and fails with several together. When they appear, the problem is not that the agent is bad, but that the spec asks too much of it at once.

13.4 Systematizing reusable context

The operational countermeasure teams have settled has two parts. The first is to give each task the portion of spec it needs, instead of loading the complete document into every interaction. The second is to systematize reusable context in dedicated files the agent loads when it needs them, instead of fattening a single document. Thoughtworks documents it as the practice with which its teams keep coherence and prevent instruction inflation.

That institutional knowledge, such as house conventions, deployment procedures, or in-house architectural patterns, is what the model does not come with. The ecosystem today packages it in named pieces, agent skills, with a file format that describes them and that the agent invokes when the task calls for it. Training in the field closes its syllabus precisely there, automating the flow with the team's own skills. It is the answer to the curse of instructions the chapter describes, and for now it lives inside this chapter and not in a card of its own.

Few rules always, the rest when it is time. The constitution contains the few rules that always apply. The prompt of each task, the specific ones. The reusable pieces are loaded when they are needed. And the complete spec exists as a reference, without being loaded in full into every interaction.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the curse of instructions and its effect on compliance, with the evidence that supports it.
- Distinguish context engineering from prompt engineering.
- Respond to overload by decomposing, prioritizing, and separating levels, instead of adding detail.
- Recognize the signs of an overloaded spec.
- Systematize reusable institutional knowledge in pieces the agent loads when it needs them.

Common mistakes and antipatterns

Responding to a failure by adding more instructions. It makes the problem worse. The answer is to split better, not to write more.

Long lists of rules of equal weight. The model does not weigh them. The critical ones go first and separated by level.

Loading the complete spec into every interaction. It saturates the context. Only the relevant portion is fed.

Repeating the house knowledge in every spec. Conventions and procedures are written once in a reusable piece, and the spec refers to it.

Further reading

· Anthropic — Effective context engineering for AI agents

· Curse of Instructions (OpenReview)

· Liu et al. — Lost in the Middle (arXiv)

· Chroma — Context Rot

· Thoughtworks Technology Radar — GitHub Spec Kit

BLOCK C · WRITING GOOD SPECS

Chapter 14. Living specs, drift, and the Clarity Gate · CONSOLIDATING

Keeping the spec aligned with the code as it evolves; drift is the main medium-term risk and there is a simple pattern for managing it.

Introduction

Writing a good spec is half the challenge. The other half is keeping it aligned with the code as the project moves forward. Drift occurs when the spec says one thing and the code has evolved toward another, and it happens fast. A minor agent decision the spec did not cover, or an adjustment during implementation that nobody carried to the document, is enough. A drifted spec is an active risk, because whoever consults it decides on incorrect information. In this chapter we will see the three levels of life of a spec, the test that detects implicit assumptions, and the strategies for keeping it alive.

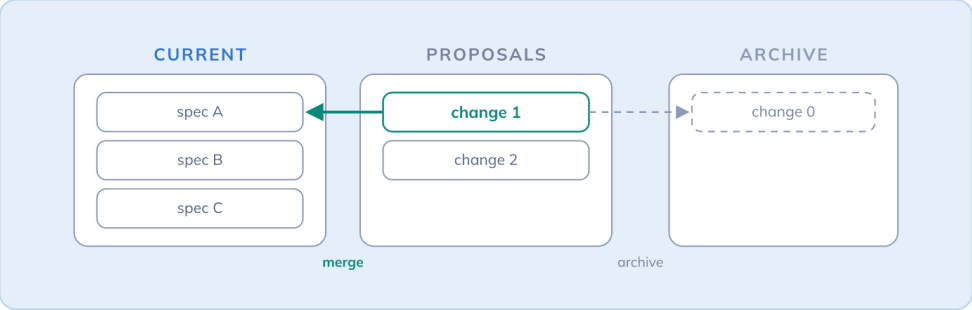


Figure 14. *Current truth and proposed changes: current specs live apart from change proposals, and what is completed is archived instead of being mixed with what is alive.*

14.1 The three levels of life of the spec

Böckeler's taxonomy describes how different teams manage the life cycle.

- **Spec-first.** It is written before coding, guides the current task, and is discarded. Drift is not a problem because there is no spec to maintain, but the documentation is lost as an asset.
- **Spec-anchored.** It is maintained as living documentation of the system. Drift is a real risk that demands discipline, because when the code changes, the spec is updated.
- **Spec-as-source.** The spec is the main artifact, only the spec is edited, and the code is regenerated. Drift disappears by design, but regeneration poses performance and determinism challenges.

The first edition of this guide left the third level as an open frontier under watch. The signal has resolved in the negative. Tessl's engine, the reference for that level, is still in closed beta, and the company itself has shifted the weight of its product toward the specification registry and governance. Spec-as-source is still experimental, and most teams work today at the second level.

14.2 The Clarity Gate

The Clarity Gate is a quality test with a single question. Can a different agent, or a new session of the same agent, generate functionally equivalent code by reading only the spec. If yes, the spec is clear and complete. If not, it has implicit assumptions, knowledge that lived in the context of the original conversation and never got written down, and those assumptions are the main cause of drift.

It is a test any team can apply. The spec is given to another agent with no context and the result is compared. What the second agent does differently is what the spec did not say. And it is also a cleaner diagnostic than traditional debugging. If the spec is correct and the code does not meet it, the problem is one of implementation and the task is regenerated. If the spec is incorrect, the spec is fixed and regenerated.

14.3 Strategies for keeping specs alive

For teams working with spec-anchored specs, the practices that have settled are four. Update the spec in the same commit as the change, because “later” does not exist. Version it, with a history of what changed and why. Review the state of the specs in the Sprint Retrospective, to decide which are still alive. And, for critical specs, create conformance tests that verify the code meets the spec.

Meanwhile a simple, practical pattern has appeared, OpenSpec’s, which separates what is current truth from what is only a change proposal. The system specs live in one directory, the change proposals in another, and when a change is completed, its spec is merged into the current one and the proposal is archived. The pattern does not depend on the tool. Any team can adopt it with two folders and one rule, and it resolves the most frequent confusion, that of not knowing whether what you are reading describes what there is or what was wanted.

What is completed is archived. A spec that describes an already made change is not living documentation of the system, it is history. Merging what is approved into the current truth and archiving the proposal keeps what matters readable.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain drift and why an outdated spec is an active risk.
- Distinguish the three levels of life of the spec, situate the real status of the third, and choose the one appropriate to the context.
- Apply the Clarity Gate to detect implicit assumptions in a spec and use it as a diagnostic when something fails.
- Use the maintenance strategies: update in the same commit, versioning, review in the retrospective, and conformance tests.

- Organize the specs of a project by separating current truth from change proposals and archiving what is completed.

Common mistakes and antipatterns

“I’ll update it later”. Later does not exist. The spec is updated in the same commit as the change or it drifts.

Keeping every spec alive by default. Maintenance has a cost, and a spec nobody consults or updates is zombie documentation, as we will see in chapter 17.

Assuming the spec is clear without testing it. The Clarity Gate reveals implicit assumptions that cannot be seen from inside the original conversation.

Mixing what is current and what is proposed in the same place. Whoever reads does not know whether it describes what there is or what was wanted, and neither does the agent.

Further reading

· Böckeler — Understanding Spec-Driven Development

· OpenSpec

· Tessl

BLOCK D · SDD IN THE TEAM AND ITS ECOSYSTEM

Chapter 15. Fit with agile roles and ceremonies · CONSOLIDATING

How SDD redistributes the team's work, where the spec lives in relation to the backlog and the sprint, and the tension between individual efficiency and team capacity.

Introduction

SDD is not a layer added without changing the existing work. It changes the nature of each role's work and the shape of the ceremonies. The Skill Arena skill "Scrum in teams with AI" describes the structure of the team that works with agents, with its roles and its artifacts. SDD describes how that team builds software day to day. They are two sides of the same adaptation, and in this chapter we will see the second: what each role does, where the spec lives, what changes in the ceremonies, and what new tension the team has to manage.

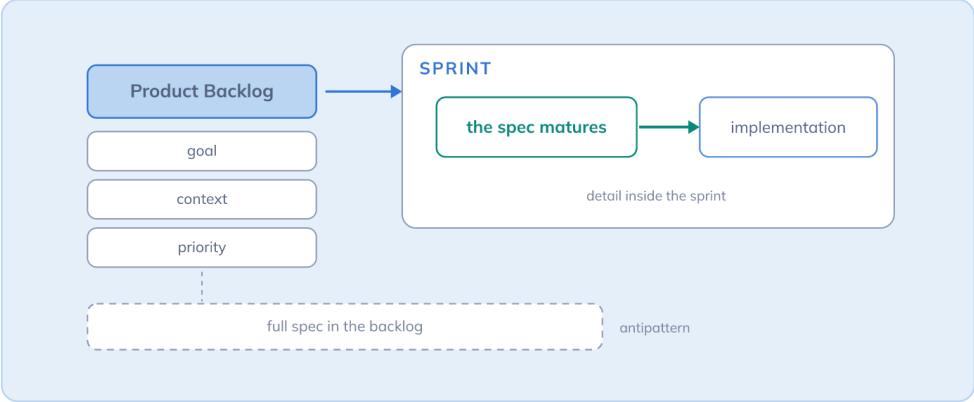


Figure 15. *Where the spec lives: the Product Backlog agrees on goal, context, and priority, and the contract matures inside the sprint.*

15.1 The roles

- **Product Owner and Product Architect.** The demand for precision rises, because acceptance criteria go from guide to contract. In return, specs are artifacts any stakeholder can read, which shortens the feedback cycle with the business.
- **Developer and Product Builder.** The competence shifts from code to analyzing, specifying, orchestrating, and reviewing. They gain the ability to produce at a speed previously impossible, but only if the earlier phases were done well, because the investment shifts from execution to planning.
- **Agile Enabler.** Facilitates the approval gates as inspection points and watches the new risks, which chapter 17 catalogs, so the team keeps its iterative pace.
- **Stakeholders.** They gain visibility. They can read a spec and weigh in on priorities or compliance before it becomes code, and the information asymmetry between those who build and those who pay or use is reduced.

15.2 Where the spec lives

During 2026 practical doctrine has appeared with a clear thesis. The spec does not replace the backlog or refinement. In the Product Backlog and in refinement the team agrees on goals, context, and priority, and the detailed contract matures inside the sprint, as part of doing the work. That is where the detail is decided, with the work already started and with the information that only appears when starting.

The most common integration mistake consists of taking the complete spec to the Product Backlog. It turns the backlog into a store of documents that expire before being executed, gives the team back the waterfall feeling that chapter 3 dismantled, and shifts the refinement conversation, which is where the why is agreed, toward technical detail that is not yet due. Yeret's answer to his own question goes along these lines. SDD is a step forward when the spec is born inside the team that builds, and a step back when it becomes a document someone hands over from outside.

The backlog agrees, the sprint specifies. Goal, context, and priority in the backlog. Requirements, design, and tasks inside the sprint. The approved spec is the Definition of Ready of that feature, said in the language of SDD.

15.3 The ceremonies with SDD

The sprint comes to contain two kinds of items, specs to be created and approved specs ready to implement, and Sprint Planning decides how many of each kind get in. The estimate splits into the specification effort, which is human and the one that consumes the team's real capacity, and the implementation effort, which falls mostly on the agent and is more predictable.

The Definition of Done is enriched with four conditions: the code meets the success criteria of the tasks, it respects the design, the commits are atomic, and the spec has been updated if the implementation differed. The Daily Scrum focuses on the state of the specs and on deviations, instead of the mechanical run-through of tasks. And code review changes its question. It is reviewed against the spec, that is, whether it meets what it says, and not only against the reviewer's taste.

15.4 The paradox of the individual and the team

The academic work in the field describes a tension the team has to manage. Individual efficiency rises with agents, while the team's review capacity degrades and the stability of the system drops. Each person produces more, and the team as a whole absorbs what is produced worse. It is the same bottleneck DORA describes from the delivery side, seen from the organization of the work.

The proposal of Díaz, Gayoso, Cimmino, and Pérez is to use specifications as the governance of agent behavior at team scale, with a double harness: technical controls around the agent and methodological governance around the team. The term links to another Skill Arena skill, "Harness Engineering", which deals with the first half of that

pair, and the paper also identifies five interaction patterns between people and agents that redefine the human roles. For the team, the practical lesson is that the gates, the constitution, and the Definition of Done are the instruments with which the organization recovers the review capacity that individual speed erodes.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Describe how the work of each role on the team changes with SDD, including the non-technical ones.
- Situate the spec in relation to the Product Backlog, refinement, and the sprint, and recognize the antipattern of taking the complete spec to the backlog.
- Explain the two kinds of sprint items, the split of the estimate, and the conditions SDD adds to the Definition of Done.
- Review code against the spec and not only against the reviewer's judgment.
- Explain the paradox between individual efficiency and team capacity, and which instruments of the method offset it.

Common mistakes and antipatterns

Treating SDD as a matter for developers only. Specification is shared work, and the non-technical roles gain voice and visibility.

Taking the complete spec to the Product Backlog. The backlog agrees on goal, context, and priority. The contract matures inside the sprint, with the work already started.

Estimating only the implementation. The effort that consumes the team's capacity is specification, not the agent's.

Measuring success by individual output. If the team cannot review what each person produces, speed turns into instability.

Further reading

· Díaz, Gayoso, Cimmino, and Pérez — SDD for Agentic Software Engineering (arXiv, 2026)

· Yuval Yeret — Is Spec-Driven Development a step forward or back for product development?

· Skill Arena — Scrum in teams with AI (in Spanish)

· Skill Arena — Harness Engineering

· Scrum Manager — Scrum in the age of AI (in Spanish)

BLOCK D · SDD IN THE TEAM AND ITS ECOSYSTEM

Chapter 16. Traceability and human oversight as an obligation · EMERGING

Since August 2026 the high-risk framework of the EU AI Act requires logging, human oversight, and transparency; SDD is not mandatory, but what it produces serves the conformity file.

Introduction

Since August 2, 2026, the high-risk framework of the European Union's Artificial Intelligence Act has applied. It is a recent fact, with professional practice yet to be built, and that is why this card enters the map as emerging. In this chapter we will see what the regulation requires, where the boundary lies between what it governs and what it does not, what SDD brings to the conformity file, and what should not be claimed.

Figure 16. *What the regulation governs and what SDD provides: the AI Act governs the high-risk system that is built, and the requirement-to-commit chain and the documented gates are material for its file.*

16.1 What the high-risk framework requires

For AI systems classified as high-risk, the AI Act requires three things of interest to this topic. Automatic logging of the system's operations, so that its functioning can be reconstructed (Article 12). Human oversight assigned to people with the necessary competence, training, and authority, which falls on the deployer (Article 26). And transparency about AI-generated content (Article 50). They are obligations for whoever builds and deploys the system, with deadlines that have already passed for the main categories.

16.2 The boundary: the system, not the tool

The scope should be made precise with care, because it is easy to state something false. What the AI Act governs is the high-risk AI system the team builds, for example a system that decides on access to credit or to a job. It does not govern the use of AI to program that system. That a team uses coding agents does not make it subject to these obligations, and not using them does not exempt it if its product is high-risk.

Usefulness, not obligation. SDD is not mandatory under the AI Act. What happens is that, when the product enters the high-risk field, the artifacts SDD produces anyway turn out to be usable to demonstrate conformity. The relationship is one of convenience, and that is how it should be stated.

16.3 What SDD brings to the file

When the product is indeed regulated, two pieces the method already had change in value. The first is the chain that runs from requirement to code. Each atomic commit is linked to a task, each task to a design, and each design to some requirements, as we saw in chapter 8, and that chain is exactly the traceability a conformity file needs to explain why the system does what it does.

The second is the documented approval gates. Each gate records that a person with judgment reviewed and approved a document at a specific moment, and that is evidence of human oversight over the build process. To them is added the project constitution, which fixes in writing the constraints the system must respect and works as a statement of technical policy. None of the three pieces is created for the regulator, and all three serve it.

16.4 What it does not bring and why the card is emerging

SDD does not by itself cover the obligations of the AI Act. Automatic logging of operations is a property of the system in production, not of its build process, and transparency about generated content is resolved in the product, not in the spec. A team that confuses the traceability of its method with the compliance of its system will be making a serious mistake.

The card is emerging because the intersection between SDD and the AI Act is yet to be explored in practice. There are no published conformity files yet that draw on the traceability of a spec, nor doctrine from the authorities on what evidence of the development process they consider sufficient. What is certain is that the team that works with approved specs and traceable commits comes to that conversation with more material than the one that works from conversations nobody keeps.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State the three obligations of the high-risk framework that affect development: logging, human oversight, and transparency.
- Situate the boundary between what the AI Act governs, the high-risk system that is built, and what it does not, the use of AI to program.
- Explain which SDD artifacts turn out to be usable for a conformity file and why.
- Distinguish the traceability of the build process from the compliance of the system in production.

- Communicate the relationship between SDD and the AI Act as usefulness and not as obligation.

Common mistakes and antipatterns

Claiming that SDD is mandatory under the AI Act. The AI Act governs the high-risk system, not the method used to program it. The relationship is one of usefulness.

Believing that using agents makes the team a regulated subject. What is governed is what the product does, regardless of how it was built.

Confusing process traceability with system compliance. Logging of operations and transparency are resolved in production. The spec provides evidence of the process, not of the system.

Ignoring the matter for being recent. The deadlines have passed, and the team that builds high-risk products needs the material SDD produces anyway.

Further reading

· EU AI Act — Article 12, record-keeping

· EU AI Act — Article 26, obligations of deployers

· EU AI Act — Article 50, transparency

BLOCK D · SDD IN THE TEAM AND ITS ECOSYSTEM

Chapter 17. Antipatterns: specification theater and zombie documentation · CONSOLIDATING

The characteristic failure modes of badly applied SDD, the external critique that describes them, and the Agile Enabler's role in watching for them.

Introduction

Recognizing the antipatterns is as much a part of mastering the method as knowing the flow. They are the points where SDD stops adding value and starts getting in the way, and they appear easily if there is no calibration with the principle of proportionality. In this chapter we will see the four characteristic failure modes, the external critique that describes them without mercy, a related question about the agent loop, and the role of whoever watches for them.

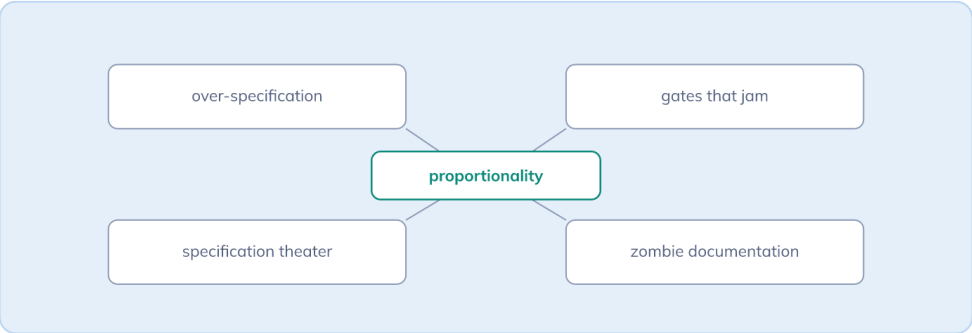


Figure 17. *Four failure modes: over-specification, gates that jam, review that is not exercised, and specs nobody consults, all four with proportionality as the remedy.*

17.1 The four antipatterns

- **Over-specification.** The team invests more time in perfecting the spec than in producing working software. It is the sledgehammer to crack a nut of chapter 4 carried to the team's process, and its extreme form is rigidity, applying the same intensity to a trivial defect and to a complex feature.
- **Gates as bottlenecks.** An approval gets blocked because whoever must approve is unavailable or cannot keep up. The answer is that of chapter 7, delegate, accept asynchronous approval, and shorten the specs, and not remove the gate.
- **Specification theater.** Specs are written that nobody really reviews, and the process runs mechanically without providing the value of the review. A gate that is not exercised is worse than its absence, because it gives a false guarantee.
- **Zombie documentation.** Specs nobody consults or updates. A spec that serves no purpose is not living documentation, it is dead weight, and deciding which specs to keep and which to let die is part of the management, as we saw in chapter 14.

17.2 The external critique

Zaninotto's critique that we saw in chapter 3 deserves to be read first-hand, because it describes these failures with the sharpness of someone who has suffered them. Huge documents nobody reads in full. Loss of the code's context, because the spec talks about an imagined system and not the one that exists. Duplicated review work, first of the spec and then of the code that does not meet it. And a false sense of security, that of believing the process protects when it merely runs.

Each of those failures has its remedy in a chapter of this guide, and that correspondence is the best proof that the method, well applied, already has them provided for. The critique describes what happens when it is applied badly, and that is why it is so useful.

17.3 Theater or value: the same question for the agent loop

Böckeler has asked the same question this chapter asks of specs, but of the agent loop. On examining whether test-driven development inside that loop provides real value or only theater, she finds that an agent can write tests that pass without the tests checking anything relevant, in the same way that a team can write specs that get approved without anyone reading them. The parallel is instructive. The form of a practice does not guarantee its substance, and the question to ask before any ritual of the method is whether someone exercises judgment in it or it merely runs.

The sign of theater. When a gate never rejects anything, or a spec never changes after its review, or tests never fail, the most likely thing is not that everything is fine. It is that nobody is looking.

17.4 The role of the Agile Enabler

These risks do not fix themselves. The Agile Enabler ensures that real time is devoted to review without the gates becoming a formality, moderates disagreements about the specs, detects when the process generates cost without return and adjusts the intensity, and watches that the specs do not turn into small waterfall projects. It is the function that keeps SDD agile in practice, and the reason the method needs someone to look at the process in addition to those who execute it.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Identify the four antipatterns of badly applied SDD and the chapter of the guide that remedies each one.
- Distinguish specification theater and zombie documentation from practice that adds value.
- Summarize the external critique of SDD and use it as a preventive checklist.

- Apply the question “theater or value” to any ritual of the method, including the agent’s tests.
- Describe the Agile Enabler’s role in watching for and correcting these risks.

Common mistakes and antipatterns

Confusing rigor with the amount of documentation. More specs do not make better SDD. Over-specification is a failure, not a virtue.

Keeping gates nobody exercises. Specification theater gives a false guarantee. It is worth approving with judgment or rethinking the gate.

Piling up zombie specs. A spec nobody consults or updates is dead weight. Deciding what to let die is part of the management.

Trusting that the tests pass. Tests that never fail may be checking nothing. The question is what they verify, not how many there are.

Further reading

· Marmelab — Spec-Driven Development: The Waterfall Strikes Back

· Böckeler — TDD inside the agent loop: theater or actual value?

· Böckeler — Understanding Spec-Driven Development

BLOCK D · SDD IN THE TEAM AND ITS ECOSYSTEM

Chapter 18. The tool ecosystem · EMERGING

The toolkit grows fast and the specific list expires before the method does; what is worth learning is the selection criteria.

Introduction

The tool ecosystem around SDD is the most volatile area of the topic. The first edition of this guide devoted this chapter to a snapshot of the landscape as of June 2026, with versions and figures, and warned that it would expire before the rest. Three months later it had expired point by point, which shows that the warning worked and that the snapshot was not the right format. In this edition we do something else. We give the criteria with which a tool is chosen, which do not expire, and the signs that indicate the method is becoming institutionalized.

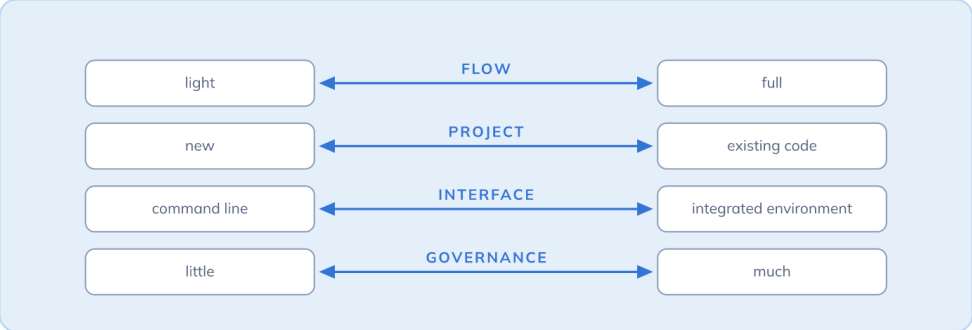


Figure 18. *Four selection criteria: light or full flow, new project or existing code, command line or integrated environment, and how much governance is needed.*

18.1 Why we do not give a list

A list of tools with their versions inside a document expires in weeks. The reference implementation releases versions frequently, alternatives appear and disappear, and adoption figures change daily. A professional who learns the list learns something that will stop being true before they need it. A professional who learns the criteria can evaluate any tool that appears, including those that do not exist yet.

18.2 The four criteria

The criteria that separate tools today are four.

1. **Light or full flow.** Some tools implement the whole flow, with constitution, phases, gates, and convergence. Others are deliberately light and limit themselves to organizing the specs and their changes. The principle of proportionality of chapter 4 also applies to the choice of instrument.
2. **New project or existing code.** There are tools designed to start from scratch and tools that declare themselves brownfield-first, with the ability to describe what already exists. Chapter 9 explains why the second family matters more than the literature suggested.

3. **Command line or integrated environment.** A command-line tool is added to the team's existing flow and works with the agent it already uses. An integrated environment imposes its own flow and its own agent, in exchange for a more guided experience.
4. **Governance and traceability.** How much the team needs the tool to log, version, and allow auditing. It is the criterion that gains weight when the product enters the field of chapter 16.

The underlying warning stands. SDD is the method and not the tool, and mastering a tool is not mastering SDD. The fundamentals of blocks A, B, and C are stable, and the tools that implement them are the layer that moves most.

18.3 Signs of institutionalization

Without giving figures that expire, there are signs from 2026 that indicate the method is settling as a practice and not a fad. The reference implementation published its version 1.0 in August 2026, on turning one year old, with a revealing rationale. It declines to promise the stability a 1.0 used to mean, because refactoring has become cheap, and claims adaptability instead. The method already has its own entry in the Thoughtworks Technology Radar, formal training in a DeepLearning.AI course, and academic literature, as we saw in chapters 2 and 15.

And the ecosystem has widened instead of concentrating. Around the reference implementation, options with different profiles have taken shape, from integrated environments to light tools aimed at existing code, and the comparisons being published already have half a dozen stable names. Proliferation is not dispersion. It is the sign that there is enough demand for several answers to coexist.

Check any snapshot, including this one. The specific landscape is consulted in the comparisons that are kept up to date, not in a dated document. What this chapter offers is the criteria for reading them.

18.4 Why the card is still emerging

The card is emerging because the toolkit is still taking shape, and at the same time it is the one that gives the clearest sign that the method is becoming institutionalized. The two things are compatible. A method settles when its tools start competing, and tools compete by changing fast. Investing the learning in the method, and in the criteria for choosing the instrument, is what protects the professional from the obsolescence of any list.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain why a dated list of tools is not the right way to learn the ecosystem.

- Apply the four selection criteria to a specific tool and justify the choice for a given team.
- Distinguish the SDD method from the instrument that implements it.
- Recognize the signs of institutionalization of the method and what each one indicates.
- Consult the up-to-date comparisons instead of taking a snapshot as a permanent reference.

Common mistakes and antipatterns

Confusing the method with the tool. Mastering a tool is not mastering SDD. The tool changes and the method remains.

Choosing the tool without understanding the method. It leads to applying the flow mechanically, with no judgment about when and how intensely.

Choosing the most complete tool by default. A full flow for a team that needs to organize light specs is the sledgehammer to crack a nut in instrument form.

Taking any landscape as permanent. It is the most volatile layer. It is checked against the current edition of the map and against the up-to-date comparisons.

Further reading

· GitHub Spec Kit — releases

· Spec Kit turns one and ships 1.0.0

· Thoughtworks Technology Radar — GitHub Spec Kit

· Böckeler — Understanding Spec-Driven Development

· DeepLearning.AI — Spec-Driven Development with Coding Agents