METHODOLOGY PAPER · APPLICATION GUIDE

From Scrum to Spec-Driven Development

A method for teams already working with Scrum: how to apply it, how to reuse its ceremonies and how to measure the maturity of the adoption

Jorge Roberts

Version 1.0 · October 2026

Abstract. When an artificial intelligence agent generates most of the code, the cost of building drops by orders of magnitude, while the cost of deciding what to build and of validating what was built stays the same, because it still depends on people. Scrum was designed on the opposite cost structure: iterating on code was the cheapest way to learn. This paper presents Spec-Driven Development (SDD) as a response to that shift, in operational terms, for a reader who knows Scrum and nothing more. It defines the work cycle, its artifacts and its checkpoints (gates); it shows how every Scrum ceremony and artifact is kept with a new purpose; it proposes a set of metrics that replaces story points and velocity; it discusses why SDD is not a return to waterfall; and it describes a three-level maturity model (spec-first, spec-anchored and spec-as-source), with spec-anchored as the target and the concrete mechanisms to reach it. It includes a complete example in a mobile application, an eight-week transition plan and a catalog of the failure modes of partial adoption.

Keywords: spec-driven development, Scrum, agent-assisted development, design by contract, quality gates, maturity model.

---

1 Introduction

Scrum solved a concrete problem: in contexts of high uncertainty, specifying everything up front was expensive and almost always wrong. The alternative was to build something small, show it, learn and correct in the next cycle. That strategy works as long as building is the cheapest way to discover. The sprint, estimation in story points, velocity and the demo as the moment of discovery are consequences of that assumption.

With AI agents in the development loop, the assumption no longer holds. Generating an implementation takes minutes. What still costs the same is what a person does: understanding the problem, deciding the correct behavior and verifying that what was generated meets it. If a team adds agents and keeps the method intact, it gets more code with the same quality of decisions, and the bottleneck moves to review, where it piles up without any Scrum metric showing it.

The phenomenon has a name: **the AI productivity paradox**. Coding assistants noticeably increased individual speed, but system metrics (lead time, deployment frequency, change failure rate) stayed flat or got worse in many organizations. The explanation is a wrong diagnosis of the original problem: the bottleneck was never typing speed, but ambiguity about what had to be built. Accelerating generation on top of ambiguous requirements does not produce better software: it amplifies the existing dysfunction, with more wrong solutions, more technical debt and more hours of review and rework.

Spec-Driven Development reorganizes work around that new bottleneck. The team iterates on the specification, which is cheap to correct, and generates the code against what was approved. The specification stops being documentation and becomes the contract against which the result is accepted or rejected.

WHAT THE READER GETS

1. A method that can be applied from the first week: the SDD cycle, its artifacts, a specification template and a complete example (sections 3 and 4).
2. The Scrum ceremonies converted: what each one keeps, what changes, which artifact it produces and what its anti-pattern is (section 6).
3. A three-level maturity model (spec-first, spec-anchored and spec-as-source), with spec-anchored as the target, the mechanisms to move from one level to the next and a self-assessment (section 8).

Scope. The paper is tool-agnostic. The concepts apply with any coding agent and any version control system. It assumes a product team of five to ten people who work with Scrum today and already use, or will use, AI agents to generate code.

2 Conceptual framework

2.1 *Definitions*

Specification (spec). A versioned document that describes the expected behavior of a feature in verifiable terms: preconditions, transitions, postconditions, invariants, acceptance criteria and explicit exclusions. It does not describe how to implement it.

Contract. The approved spec. It is the reference against which everything generated is accepted or rejected. The notion comes from design by contract (Meyer, 1992), which in turn formalizes Hoare's precondition, action, postcondition triple (1969).

Technical plan. A document derived from the spec that describes how it will be implemented: affected components, design decisions, dependencies and risks. The agent proposes it and a person validates it.

Gate. A checkpoint that blocks progress until a person with authority (or an automated test) approves the artifact. A gate that does not block is a suggestion.

Project context (constitution). The minimum set of versioned files the agent reads before every task: non-negotiable principles and rules, approved and forbidden technologies, reference architecture and security requirements. It is the team's institutional memory written for a machine, and it keeps the agent from inventing its own conventions. Agents read it; only people change it.

Dual perspective. SDD requires two specifications that converge before generating: the product one, which defines what is built and why (vision, user journeys, value-oriented criteria), and the engineering one, which defines how the agent must build it (the project context). Each feature's spec lives at the intersection of both.

Executable acceptance criterion. An automated test derived from the spec before the code exists. It replaces the Definition of Done as a verbal agreement.

Drift. Divergence between what the spec says and what the code does. It appears every time someone fixes behavior in the code without updating the spec first.

Calibration. Correcting the project context (not the specific artifact) when the same error appears more than once. It is the mechanism through which the system learns.

Validator. A person with explicit authority to reject an artifact at a gate. Their rejection is not reversed under deadline pressure.

2.2 *The three differences that matter*

Scrum and SDD share the rejection of exhaustive predictive planning, incremental delivery and a focus on value. The differences are three, and the rest of the method follows from them.

First: the substrate of iteration changes. Both models iterate. Scrum iterates on code: it builds a hypothesis, exposes it and corrects it in the next cycle. SDD iterates on the contract: it refines the spec, validates it and generates against what was approved. The reason is economic. Correcting a paragraph of the spec takes minutes. Correcting the same decision after an agent propagated it to dozens of files, schemas and tests takes days, and is usually done badly under delivery pressure. The relationship between when a defect is detected and what it costs to fix has been documented for decades (Boehm, 1981). Agents amplify it, because they spread a wrong decision to more places and faster than a person does.

Second: the enforcement mechanism changes. In Scrum, keeping agreements depends on team discipline, and a broken agreement is discussed in the retrospective. In SDD, it depends on gates that block progress: AI proposes, people decide and the flow enforces it. A team that keeps the specs but not the blocking adopted SDD's vocabulary, not the method.

Third: what is measured changes. Story points measure the effort of building. When building costs minutes of compute, the metric loses its referent and starts rewarding volume. In SDD the effort of validation, the scarce resource, is what gets estimated and measured. The corollary is counterintuitive and worth saying on day one: a large but isolated and well-understood feature can be estimated low, and a small change that touches the user model or a legacy schema can be estimated high. What weighs is the risk of validating, not the size.

Table 1. Structural comparison between Scrum and SDD.

Dimension	Scrum	Spec-Driven Development
Economic assumption	Building is the cheapest way to learn.	Building is almost free; deciding and validating are what is expensive.
Object of iteration	The code increment.	The specification.
Source of truth	The code and the team's conversation.	The approved spec and the project context.
Enforcement	Collective discipline; retrospective.	Blocking gates with a named validator.
"Done"	Agreed Definition of Done.	Executable criteria derived from the spec, passing.
Unit of estimation	Story points (building effort).	Validation effort, weighted by risk.
Unit of delivery	The sprint.	The validated feature, when it passes its gates.
Scarce resource	Development capacity.	Validation capacity of expert people.
System learning	In the people.	In the people and in the versioned context.

2.3 *What is kept and what is retired*

Half of the resistance to the transition comes from assuming that SDD eliminates Scrum. It does not. What is kept: incremental delivery (more granular than in Scrum, because delivery happens per validated feature and not per sprint), adaptation to change (now executed on the spec), the client demo (as communication of results, not as discovery), the retrospective (which gains weight, because it starts auditing the work system) and refinement (which becomes the central ceremony).

Four things are retired: the user story as a direct input for implementation, story points and velocity, the sprint as the unit of delivery and estimation in development hours. All four were calibrated on a cost structure

that no longer exists. Keeping them "just for reporting" is the shortcut that hurts the transition the most (section 10).

The user story does not disappear: it stays as the value statement at the top of the spec. What is retired is its use as a work order. An agent that receives a user story fills in, by probability, everything the story does not say, and does it without warning.

2.4 *SDD is not waterfall*

The most common objection from an agile team is that SDD is the waterfall of the seventies under another name: dense documentation before code and analysis paralysis. The equivalence is superficial, because it confuses writing the specification early with freezing it. In waterfall, documentation is a governance artifact, designed to obtain approvals and resist change. In SDD, the spec is a feedback mechanism, meant to be iterated and versioned as the implementation reveals new realities (Brooker, 2026).

Table 2. Structural properties of waterfall and SDD.

Property	Waterfall	Spec-Driven Development
Nature of the document	Rigid governance artifact, to obtain formal approvals per phase.	Active technical feedback mechanism; a living artifact.
Format	Static documents (word processor, PDF, isolated wikis).	Machine-readable structured text (Markdown, YAML), versioned with the code.
When validation happens	At the end, in a separate phase, months after design.	Continuously, inside each feature's cycle, in the pipeline.
Response to change	Resistant: going back requires bureaucracy that discourages adapting.	Adaptive: revising the spec is the expected response to what is learned.
Document-code drift	Manual, exhausting reconciliation; the document becomes obsolete almost immediately.	Automatic detection through tests derived from the contract.
Roles	Analysts who write in isolation and developers who build in silence.	Collaboration: the same people write the intent and validate, in a loop with the agent.
Scope and cycle time	Macro: the whole system, for months, before the first line.	Micro: one feature at a time, in cycles of hours or days.

The decisive difference is how drift is handled. In waterfall, keeping the document in sync with code that changes all the time costs too much human effort, so the document is abandoned and the code takes over as the source of truth. In SDD, spec, code and tests live in the same repository and under the same version control: updating the spec is an act of semantic refactoring, not paperwork.

2.5 *Where it comes from*

SDD is not an isolated invention but the instrumented convergence of earlier practices. From **design by contract** (Hoare, 1969; Meyer, 1992) it takes the precondition, transition and postcondition triple, and the notion of invariant. From **Behavior-Driven Development** (North, 2006) it takes describing behavior through concrete examples in structured language, of the given, when, then type. BDD's historical limitation was that the scenarios stayed as consultative documentation or fed tests that were expensive to maintain. With agents, those scenarios become executable and, in addition, constrain what the agent can generate.

It also converges with requirements management. The classic distinction between *requirement* (the business need that motivates the work) and *specification* (the precise description of how it is satisfied) is exactly the one SDD makes operational: intent lives in the backlog, the specification in the repository, and the bridge between

them is traceable. In regulated industries such as aviation (DO-178C), medical devices (IEC 62304) or automotive (ISO 26262), that traceability stops being a manual effort at the end of the project and becomes a by-product of the daily cycle.

3 The Spec-Driven cycle

The cycle has six phases and three human gates, plus a calibration loop. Figure 1 summarizes it. The return arrows are as important as the forward ones: they define where work goes back when something fails.

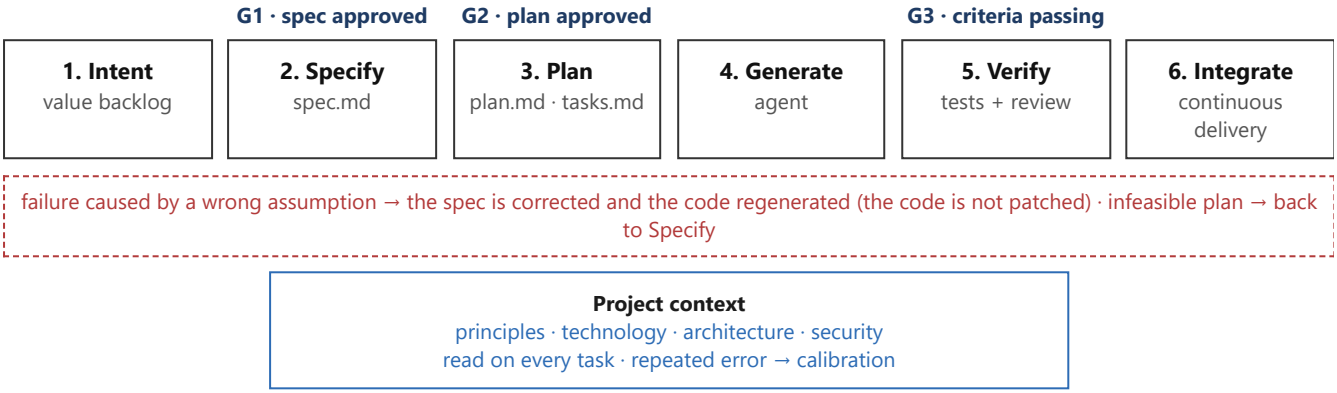


Figure 1. The Spec-Driven cycle: six phases, three gates and two return loops. The dashed red line shows where work goes back after a failure; the blue box, the learning that stays in the project context.

3.1 *The phases, one by one*

Table 3. Input, activity, output and control of each phase.

Phase	Input	Activity	Output	Control
1. Intent	Business need, feedback, metric.	The Product Owner states the outcome sought and why it matters, with no solution.	Item in the intent backlog.	None; it is an input.
2. Specify	Prioritized intent and project context.	The spec is written (with or without help from the agent), ambiguities are resolved and unknowns are declared. Acceptance criteria are derived.	spec.md	G1: the business and technical validators approve.
3. Plan	Approved spec.	The agent proposes the technical plan and breaks it into small tasks, each verifiable on its own (micro-increments integrated separately, like stacked pull requests).	plan.md, tasks.md	G2: the technical validator approves design and risks.
4. Generate	Approved plan and tasks.	The agent implements task by task, with the tests derived from the spec written first. When facing an unforeseen design fork, it stops and presents closed options instead of guessing.	Code and tests on a branch.	Local tests; no person writes code except as a documented exception.
5. Verify	Branch with the implementation.	Automated tests run. Human review focuses on what the tests do not cover: security, design decisions, unforeseen cases.	Verification report.	G3: criteria passing and review approved.
6. Integrate	Approved verification.	It is merged into the main branch and deployed at the cadence the product allows.	Delivered feature.	Continuous integration pipeline.

3.2 *What to do when something fails*

The most important rule of the cycle is that **the correction goes to the level where the error originated**. If the error is in the spec, the spec is corrected and the code regenerated. If the error has already happened before, the project context is corrected. Patching the code directly is the fastest way to slide back to Scrum without noticing: the code starts contradicting the spec and, within two weeks, nobody reads it.

Table 4. Decision when facing an incorrect result.

Situation	Decision	Rationale
The result does not meet the spec, but the spec is correct.	**Regenerate**	The agent deviated. It is rejected and regenerated with the same contract; if it happens again, calibrate.
The result meets the spec, but the spec was wrong.	**Correct spec and regenerate**	The error is a decision error, not an execution error. Fixing the code would leave the spec out of date.
The same type of error appears a second time.	**Calibrate**	It is a context problem, not an artifact problem. The corresponding rule is added or fixed so it does not come back.
The result is wrong at its root.	**Discard**	Discarding is cheap and returns to a clean context. Iterating on a bad result drags its wrong assumptions along.
Nobody knows how the affected part of the system works.	**Extract knowledge or spike**	First assisted code reading and interviews; if that is not enough, a bounded spike (section 3.3).

3.3 *Where iteration still happens: governed spikes*

SDD does not eliminate experimentation. It moves it: iteration is free in the specification, in interface prototypes shown to the user and in technical spikes. What is not done is discovering production behavior by building it. A spike is legitimate when it meets five conditions:

1. It answers a question written in advance, which the spec cannot resolve without evidence.
2. It has a declared time limit, usually hours, not days.
3. Its deliverable is a decision recorded in the spec, not code.
4. The spike's code is discarded. It is not promoted to production even if it works.
5. A validator reviews the resulting decision with the same criteria as any spec.

Condition four is the one most often broken. A spike promoted to production introduces a component that no spec governs and no criterion validated, and turns the exception into the method.

3.4 *Oversight proportional to risk*

Not every artifact deserves the same scrutiny. If validators review everything with the same intensity, they become the bottleneck; if they review too little, they approve out of fatigue. The alternative is to grade oversight by the risk of the artifact and the confidence left by the automated validations.

Table 5. Oversight levels.

Risk	What it includes	Treatment
Low	Repetitive code, mechanical transformations, changes covered by passing tests.	It moves forward, is recorded and reviewed by sampling.
Medium	Implementations with bounded business logic.	Contextual judgement. Specs and plans are always reviewed; the implementation, according to its criticality.

High	Complex domain, security, sensitive data, regulated work, artifacts from the early phases of the cycle or cases where the agent declared uncertainty.	Mandatory human review. Nothing moves downstream until it is approved.

There is an empirical reason not to lower the guard in the high tier: **automation complacency bias**. A result that looks coherent lowers the reviewer's attention, and models are designed to sound confident even when they are not. Coherence is not evidence of correctness. That is why the G3 rule is explicit: nothing is ever approved because it "looks fine".

4 The artifacts

4.1 *Minimum repository structure*

All artifacts live in the same repository as the code and are versioned with it. This is a technical condition, not a preference: if the spec lives in a wiki and the code in a repository, drift is inevitable.

```
repository/
├── context/                    # read by the agent before every task
│   ├── principles.md           # non-negotiable rules of the product and the team
│   ├── tech-policy.md          # approved / forbidden technologies, versions
│   ├── architecture.md         # layers, modules, reference patterns
│   └── security.md             # what is never logged, how auth works, sensitive data
├── specs/
│   └── 042-checkout-coupon/
│       ├── spec.md             # the contract (G1)
│       ├── plan.md             # the technical design (G2)
│       └── tasks.md            # small, verifiable tasks
├── criteria/                   # review rules: what a person validates against
│   └── ui-review.md
├── lib/ … test/                # generated code and tests
└── CHANGELOG.md
```

The project context starts with four short files and grows only when a concrete failure justifies it. A forty-page context written before the first run is as useless as having none: the agent reads all of it and prioritizes badly. The `criteria/` folder solves a different problem: that two validators do not approve different things when facing the same artifact.

4.2 *Anatomy of a spec*

A well-written spec answers six questions. The first four follow the logic of design by contract; the last two exist because the reader is an agent, which does not ask when information is missing.

Table 6. Mandatory sections of a spec.

Section	Question it answers	Typical error if missing
Preconditions	What must be true before this happens?	The agent validates halfway or not at all, and impossible states appear.
Transitions	What changes, in what order and with which calculation rules?	Inferred business rules: rounding, priorities and arbitrary combinations.
Postconditions	What must be true afterwards?	The result "works" on screen but leaves the system in an inconsistent state.
Invariants	What can never stop being true, whatever happens?	Side effects on other parts of the system that nobody asked to review.

Acceptance criteria	Which concrete examples show that it is right?	Verification by visual impression: "it looks fine".
Out of scope and unknowns	What is not done and what is not known yet?	The agent fills in what was not said with the statistically most likely option, not with the correct one.

> **THE TWO-ENGINEER TEST**
>
> A spec is ready when two engineers who read it separately would build the same thing. If they would need to agree on something, that something is missing from the spec. It is the SDD version of the Definition of Ready, and the main criterion of gate G1.

4.3 *Complete example: discount coupon at the checkout of a mobile app*

The example uses a Flutter application, but the structure is identical in any stack. On the left, the user story as it would enter a sprint. On the right, the spec that enters the SDD cycle.

USER STORY (SCRUM INPUT)

```
As a buyer
I want to apply a discount coupon
so that I pay less.

Criteria:
- I can enter the code at checkout.
- The discount is applied.
- I see the new total.
```

In Scrum, this story worked because the developer asked during the sprint. An agent does not ask: it decides at least ten unwritten things. For example, whether coupons stack, whether the discount covers shipping or taxes, how it rounds, what happens if the coupon expires during checkout, whether it works offline, what message the user sees for each error and whether the use is consumed when applied or when paid. Each of those decisions stays implicit in the code and only shows up in production, or in the demo, which is the same thing but later.

SPEC (SDD INPUT)

```
# 042 · Checkout discount coupon
Value: reduce abandonment at the last step.
## Preconditions
G1 Cart with ≥ 1 item and active session.
G2 Code: 4 to 16 characters [A-Z0-9].
G3 Requires connection. Offline it is not
   validated locally: E4 is shown.
## Transitions
T1 Validate the coupon against the server.
T2 Discount on the product subtotal;
   excludes shipping and taxes.
   Half-up rounding to 2 decimals.
T3 One coupon per order: a new one replaces
   the previous one, notifying the user.
## Postconditions
P1 The total shown matches the one the
   server recalculates at payment.
P2 The use is consumed when payment is
   confirmed, not when the coupon is applied.
## Invariants
I1 The total is never negative.
I2 The code is never written to logs.
## Errors
E1 does not exist · E2 expired · E3 does not
apply to these products · E4 offline.
## Acceptance criteria
AC1 Subtotal 100.00 and 15% coupon: total 85.00.
AC2 Expired coupon: E2 and total unchanged.
AC3 Golden test of the screen with the coupon
    applied, light and dark theme.
## Out of scope
Stackable coupons. Coupons on shipping.
## Unknowns
Is the saving shown in the summary? (PO)
```

From the spec, the tests are derived before generating the implementation. Each acceptance criterion becomes a test with its identifier, so traceability between contract and code is direct:

```
// test/checkout/coupon_test.dart · derived from specs/042, before the code
test('AC1: 15% coupon on subtotal 100.00 leaves total 85.00', () {
  final cart = Cart(subtotal: Money.parse('100.00'), shipping: Money.parse('8.00'));
  final result = applyCoupon(cart, Coupon.percent('PROMO15', 15));
  expect(result.productsTotal, Money.parse('85.00'));   // T2: excludes shipping
});
test('AC2: expired coupon returns E2 and does not change the total', () {
  final cart = Cart(subtotal: Money.parse('100.00'));
  final result = applyCoupon(cart, Coupon.expired('OLD10'));
  expect(result.error, CouponError.expired);           // E2
  expect(result.productsTotal, Money.parse('100.00'));
});
testWidgets('AC3: summary with coupon applied (golden)', (tester) async {
  await tester.pumpWidget(CheckoutSummary.fixture(couponApplied: true));
  await expectLater(find.byType(CheckoutSummary),
      matchesGoldenFile('goldens/042_coupon_applied_light.png'));
});
```

The golden test deserves a special mention. In interfaces generated by agents, manual visual review is the point where validation degrades first, because the reviewer ends up approving "what looks reasonable". Comparing against an approved reference image turns a subjective criterion into an executable one. The reference image is approved once, at G3, and from then on any visual regression blocks integration.

4.4 *From vague to verifiable*

Table 7. Phrasings that do not work for an agent and their replacement.

Vague phrasing	What the agent does with it	Verifiable replacement
"Make it fast"	Picks an arbitrary target and optimizes against it without saying so.	95th percentile response time under a value, measured at a named point.
"Handle errors"	Invents its own list of errors and tends to hide them.	Closed list of errors with code, user message and what gets logged.
"Validate the input"	Validates halfway and leaves security gaps.	Types, ranges and behavior for each violation, field by field.
"Follow best practices"	Brings in patterns foreign to the project.	Reference to the context file that governs that aspect.
"Like the existing one"	Infers the pattern by sampling the repository, with inconsistent results.	The exact file that is the reference and what must be different.
"Responsive"	Decides the breakpoints.	Reference widths and the expected behavior at each one, with a golden test per width.

The other side: a spec has to declare what it does not know. A spec that hides its uncertainty to look complete is more dangerous than one that declares it, because the agent will resolve the unknown silently. The "Unknowns" section blocks G1 until every question has an owner and an answer.

For teams looking for more syntactic rigor there are writing templates such as EARS (Easy Approach to Requirements Syntax; Mavin et al., 2009), which restrict each requirement to a few forms, for example "when <event>, the system shall <response>". They are not mandatory, but they reduce ambiguity and make it possible to review specs automatically before sending them to the agent.

4.5 *Calibration: how the system learns*

The calibration signal is precise: **if a validator has to give the same feedback twice, it is no longer feedback, it is a calibration problem.** The fix does not go to the artifact but to the project context, so the error does not repeat in any future task.

Table 8. Where each type of failure is corrected.

Type of failure	Symptom	Where it is corrected
Technical hallucination	The agent uses a library or API that does not exist or is not approved.	tech-policy.md
Style violation	It works, but breaks the agreed patterns.	architecture.md or coding standards
Business error	It misreads a rule or an invariant.	The spec and, if it is cross-cutting, a domain file in the context
Security or compliance	Secret in the code, logging of sensitive data, insecure default.	security.md and rules in criteria/
Ambiguous criterion	Two validators approve different things for the same artifact.	criteria/

Procedure. (1) When rejecting an artifact for a repeatable cause, the validator tags it with a calibration label and its category. (2) Once a week, the context owner reviews the labels and prioritizes by pattern, not by volume. (3) The change to the context is written. (4) The fix is tested: the same task that failed is run again in isolation, with the updated context. If the agent produces the correct result without additional guidance and the gate passes cleanly, the change is merged referencing the labels. If it does not pass, it is not merged, even under deadline pressure.

> **CONTEXT FIRST, SYSTEM SECOND**
>
> When something fails, the question is whether the agent is missing context or whether the work system is badly designed. In most cases it is context. Changes to tools, flow or prompts are escalated only when the failure persists with clear rules.

4.6 *Traceability*

Every artifact carries identifiers inherited from the artifact it derives from: a requirement FR-012 originates spec 042, whose criteria AC1..AC3 become tests with the same identifier, and whose design decisions are recorded as architecture decision records (ADR-007). The rule is that no identifier is invented downstream: they are all inherited. That way, when a requirement changes weeks later, it is known which specs, tests and components it affects, and tests for obsolete features are not kept alive.

5 Roles: from Scrum to SDD

SDD does not require new roles. It requires redistributing validation authority and freeing up time to exercise it. This is the most underestimated move of the transition: it is not about renaming positions, but about changing who can say "this does not move forward" and making sure that person has time to say it.

Table 9. Redistribution of responsibilities by role.

Scrum role	Stops doing	Starts doing	Sign it is half done

Product Owner	Describing features on cards.	Declaring outcomes, business rules and verifiable invariants. Validates the business part at G1 and answers the unknowns.	The technical validator has to interpret the intent before being able to validate.
Scrum Master	Facilitating individual impediments and guarding the timebox.	Governing the flow: validation latency, rework, validator load and compute cost. Upholds the "spec before code" rule.	Their metrics dashboard is updated but does not trigger decisions.
Senior developer	Writing most of the code.	Validating plans and results (G2 and G3), calibrating the context and writing the rules in criteria/.	Edits the generated code instead of rejecting it and correcting the spec.
Developer	Implementing tasks end to end.	Writing specs, driving the agent, deriving tests, resolving integration.	Chats with the agent until "it works", with no spec involved.
QA	Verifying at the end of the sprint.	Deriving tests from the spec before the code exists; looking after the golden tests.	Tests are still written after the code.
Architect or tech lead	Reviewing pull requests one by one.	Maintaining the project context and designing the gates. Reviews patterns, not artifacts.	Steps in to unblock tasks by hand: every time they do, the system does not learn.

WHAT TO HAVE SIGNED BEFORE STARTING

- Who validates at each gate and for which type of change, with the explicit statement that their rejection is not reversed because of deadlines.
- Who owns the project context and decides when a repeated failure becomes a rule.
- Who validates, on the client or business side, the spec and the acceptance criteria, not only the demo.
- What percentage of each validator's time is freed from building. Without this number, gates turn into queues.

6 The Scrum ceremonies, converted

The Scrum ceremonies are kept. What changes is their product: each one stops producing verbal agreements and starts producing a versioned artifact. The sprint cadence can also be kept, but as a governance cadence (when priorities, metrics and the work system are reviewed) and not as a unit of delivery. What passes the gates is integrated at once; it does not wait for the end of the sprint.

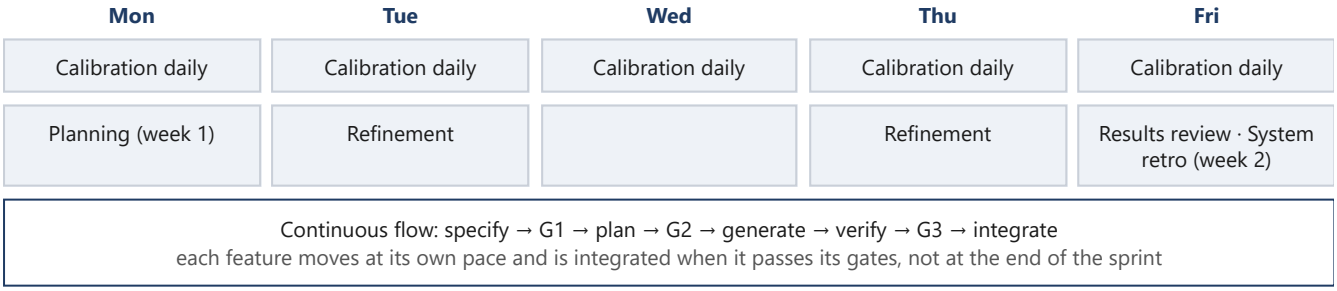


Figure 2. Suggested cadence for a two-week governance cycle. The ceremonies set the pace of decisions; delivery happens continuously inside the flow.

The cards below describe each ceremony with the same structure: what purpose it keeps, what changes, how the agenda is organized, which artifact it leaves and what its most frequent anti-pattern is.

6.1 Sprint Planning → Intent planning

every 2 weeks · 60 to 90 min

Purpose it keeps	Agree on what value is pursued in the next cycle and commit capacity realistically.
What changes	Building effort is not estimated and technical tasks are not broken down: the agent does that in the planning phase, and it is validated at G2. The capacity committed is validation capacity, not development capacity. Estimation changes its object: ambiguity, the logical complexity of the domain and integration risk are quantified. What is isolated and well understood estimates low; what touches security or legacy systems estimates high.
Agenda	(1) Review of the previous cycle's metrics, 10 min. (2) Business goals for the cycle, 15 min. (3) Selection of backlog intents that move to specification, estimated by ambiguity and risk, 30 min. (4) Check of each validator's load against their ceiling, 10 min. (5) Assignment of spec owners, 10 min.
Artifact it leaves	List of specs to write in the cycle, each with an owner, assigned validators and risk level.
Anti-pattern	Committing specs based on the agent's speed. The agent is never the bottleneck; the validator is. If nobody ends up above their validation ceiling, the planning is realistic.

6.2 Refinement → Specification workshop

twice a week · 60 min

Purpose it keeps	Bring backlog items to a state where they can be executed without surprises.
What changes	It becomes the central ceremony of the method. The work happens on the text of the spec, with the agent as an assistant to detect ambiguities and edge cases. It ends with the spec ready for G1 or with written questions that have an owner.
Agenda	(1) Reading of the draft spec, 10 min. (2) Asking the agent to list ambiguities, assumptions and uncovered edge cases, 10 min. (3) Resolving each one or recording it as an unknown with an owner, 25 min. (4) Two-engineer test, 10 min. (5) Deriving acceptance criteria, 5 min.
Artifact it leaves	spec.md ready for G1, or blocked with explicit unknowns.
Anti-pattern	Approving an "almost ready" spec in refinement so as not to slow the flow down. An unresolved unknown becomes a decision made by the agent.

6.3 Daily → Calibration daily

daily · 15 min

Purpose it keeps	Synchronize the team and detect blockers early.
What changes	The question stops being "what am I working on" and becomes "what failed in the system yesterday and which rule do we correct". The state of the flow is reviewed, not the state of the people.
Agenda	(1) Artifacts rejected yesterday and their cause: spec, execution or context, 5 min. (2) Repeated errors that require calibration, 5 min. (3) Agent deviations and conflicts with the context; work waiting for validation for more than a day and who unblocks it, 5 min.
Artifact it leaves	A correction to the project context, or an assigned calibration task. If a daily leaves neither of the two for a whole week, the team is not learning or not recording.
Anti-pattern	Going back to the individual status round. A task open for more than a few hours without progress is not progress: it is a blocked agent or missing context.

6.4 Sprint Review → Results review

every 2 weeks · 45 min

Purpose it keeps	Show stakeholders what was delivered and collect their opinion.
What changes	It stops being the moment of technical validation, which already happened at the gates, and of discovery, which already happened in the spec. Business results are shown and decisions are made about the next intents.
Agenda	(1) Features delivered in the cycle, against the spec that originated them, 20 min. (2) Business indicators affected, 10 min. (3) Changes of mind from the business, recorded as spec changes or new intents, 15 min.
Artifact it leaves	Business decisions recorded as new intents or as changes to existing specs.
Anti-pattern	The client using the demo as an approval gate again. If that happens, the sprint becomes the unit of delivery again and the method goes back to Scrum.

6.5 Retrospective → System retro

every 2 weeks · 60 min

Purpose it keeps	Improve the way the team works.
What changes	It gains weight and focus. Besides the team climate, it audits the system: quality of the specs, effectiveness of the gates, calibrations in the cycle and validator load.
Fixed agenda	(1) First-pass acceptance rate and its trend. (2) Rework and its root cause. (3) Validator load and latency. (4) Compute cost per validated feature. (5) Walk through the partial adoption table (section 10). (6) A single improvement action with an owner.
Artifact it leaves	Concrete changes in the context, in criteria/ or in the gate configuration.
Anti-pattern	The retro discussing the method in the abstract instead of the data of the cycle.

6.6 *The Scrum artifacts, converted*

Table 10. Equivalences between artifacts.

Scrum artifact	SDD equivalent	What changes
Product Backlog	Intent backlog + specs	Two states are separated: intent (what value) and spec (what behavior). Only approved specs are executable.
Sprint Backlog	Specs committed for the cycle	Validation capacity is committed, not building capacity.
User story	Value statement inside the spec	It stops being the work order; it is the first line of the contract.
Definition of Ready	Gate G1 and the two-engineer test	From verbal checklist to blocking control.
Definition of Done	Gate G3: executable criteria passing	From team agreement to tests that block integration.
Increment	Validated feature	It is delivered when it passes its gates, without waiting for the end of the sprint.
Burndown	Flow and validation board	It shows where work is waiting, not how much is left to build.

7 Metrics

The change of metrics is what defines whether the transition is real. What gets measured gets optimized, and if story points and velocity are still measured, the team will optimize the volume of generated code, which is exactly what should not be optimized.

Table 11. Replacing the metrics.

In Scrum	Why it stops working	What is measured in SDD
Velocity	With almost free generation it measures volume, not value: it can go up while quality goes down.	Features validated and delivered per cycle.
Story points	They measure the effort of building, which is no longer the expensive part.	Validation effort estimated by risk and complexity.
Capacity in hours	Building does not consume team hours; validation does.	Validation capacity per person, with a sustainable ceiling.
Burndown	The sprint is no longer the unit of delivery.	Waiting time at each gate.

Table 12. Minimum SDD dashboard. Thresholds are indicative and each team adjusts them with its own baseline.

Metric	Definition	Indicative threshold	Decision it triggers
First-pass acceptance	Artifacts approved on the first attempt / artifacts presented to the gate.	≥ 80%	Below: review spec and context quality before the agent.
Rework	Artifacts regenerated or corrected / total artifacts.	≤ 20%	Above: find the root cause in the daily and calibrate.
Validation latency	Time between an artifact reaching a gate and its decision.	< 1 day	High: validators lack time. Very low on complex artifacts: possible approval without review.
Validator utilization	Validation load assigned / declared capacity.	≤ 85%	Above: redistribute or lower the cycle's commitment.
Drift rate	Commits that change behavior without a prior spec change / behavior commits.	0%	Any higher value: an immediate conversation; it is the early sign of sliding back to Scrum.
Repeated errors	Rejections with the same cause as a previous rejection.	Downward trend	Flat or rising: calibration is not working.
Escaped defects	Production defects per delivered feature.	≤ baseline	Above: the gates approve without enough criteria.
Cost per validated feature	Compute cost and validation hours / features delivered.	Downward trend	Flat: the system is not learning; review calibration.

One pattern deserves special attention: **velocity rising and rework rising at the same time**. It is the signature of a team that generated more code without improving its decisions. If the board still shows velocity, it is the first number to look at.

8 Maturity model

Adopting SDD is not a binary event. It is a continuum, and what defines where a team stands is a single variable: **how much authority the spec has over the code over time**. Böckeler (2025), based on an analysis of

the tools that implement the approach today, distinguishes three levels: spec-first, spec-anchored and spec-as-source. This paper adopts that scale and takes an explicit position on each level.

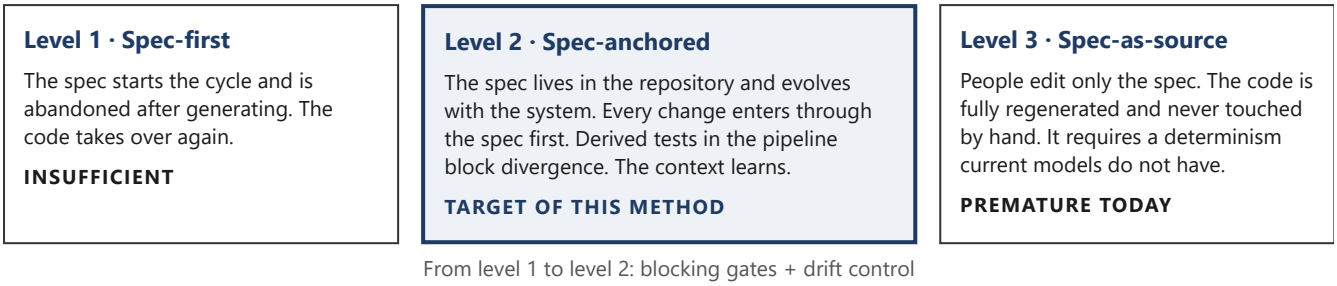


Figure 3. The three maturity levels according to the authority of the spec over the code. The step from level 1 to level 2 is built with two mechanisms: gates that block and automatic drift detection.

Level 1 · Spec-first

"The spec as a starting point" · insufficient

Definition	Every feature starts with a spec written before the agent generates code. The spec gives the agent the clarity it needs to solve the immediate task precisely.
What characterizes it	The spec is ephemeral. Once the code is generated and approved, the spec loses its role. When a requirement change or a defect appears, the code is fixed directly and the original document stays frozen.
Consequence	Within a few weeks the spec is an archaeological artifact: it reflects the original intent, not the state of the system. The code becomes the default source of truth again and drift builds up without anyone measuring it.
Where it appears	It is the most common way current tools are used: specs on temporary branches that are discarded after the feature is merged. It is also where teams without gate discipline end up.
Value	Real but limited: it removes ambiguity from the initial generation. It is the entry door and the mandatory starting point of the transition, not a destination.
Minimum practices	Spec template with the six sections (Table 6). Project context with the four minimum files. Tests derived from the acceptance criteria. Story points retired.

Level 2 · Spec-anchored

"The spec anchored and alive" · target

Definition	The spec is not discarded after generating: it stays anchored in the repository and evolves in parallel with the software for the whole life of the feature.
The rule that defines it	For any change (a new requirement, a defect, a production fix, a business decision halfway through) the spec is updated first to reflect the new desired state, and only then is the code regenerated or modified. The code is never the source of truth: the spec is.
How it is guaranteed	The tests derived from the spec run in the continuous integration pipeline and act as the bridge between intent and technical reality: if the code diverges from the specified behavior, the tests fail and integration is blocked. An additional control rejects commits that change behavior without an associated spec change.
What else it requires	Gates G1, G2 and G3 configured as technical controls, not as agreements. Named validators with time for it. A criteria/ folder. Context calibration for repeated errors (section 4.5). Converted ceremonies. The dashboard in Table 12. Identifier traceability between spec, tests and code.
Why it is the target	It is the pragmatic balance between formal structure and operational flexibility. It captures almost all the value of the approach (the contract as the source of truth, accumulated learning, auditability) without betting on a determinism models do not offer yet. It is the most robust option to keep architectural hygiene over time.
Observable evidence	Drift rate at zero. Production fixes that start with the spec. A new team member understands a feature by reading its spec, without reading the code. Cost per validated feature steadily going down.

Level 3 · Spec-as-source

"The spec as source code" · premature

Definition	The spec becomes the only valid source file. People edit only specs, in structured natural language or declarative syntax, and never modify or inspect the code. Every change to the spec triggers an automatic regeneration of components, interfaces and schemas.
What it looks like	There are experimental environments where every generated file carries a header forbidding manual edits, because any manual change is overwritten in the next generation.
Why it is not the target	First, it assumes a determinism models do not have: the same spec can produce different implementations in different runs, which forces everything to be re-verified on every regeneration. Second, it requires a very mature generation and verification infrastructure before there is evidence that the method works for the team.
Position	Do not design the adoption aiming at spec-as-source "just in case". It is premature optimization on unvalidated infrastructure. The minimum viable is spec-anchored working end to end on a real slice. Validate the simple thing before scaling the design.

8.1 *How to move from spec-first to spec-anchored*

The difference between the two levels is not in the quality of the specs, but in **what happens after the first delivery**. A team can write excellent specs and stay at level 1 if it fixes the code directly when something changes. That is why the step is built with mechanisms, not with good intentions:

Table 13. Mechanisms that turn spec-first into spec-anchored.

Mechanism	What it prevents	How it is implemented

Blocking gates	An "almost ready" spec or an unreviewed plan reaching generation.	Branch protection, mandatory reviews by type of change, a pipeline that fails.
Derived tests in CI	The code diverging from the specified behavior without anyone noticing.	Each criterion ACn has its test with the same identifier; integration is blocked if it fails.
Drift control	"Small" fixes made directly in the code.	A rule in the pipeline or in review: a behavior commit must reference a spec change.
Fast lane for urgencies	The pressure of an incident justifying skipping the spec.	A shortened version of G1 (minimum spec of the fix, one validator) with a full review afterwards.
Rebuilding legacy specs	Existing code staying outside the method forever.	When a feature without a spec is touched, its current spec is written first, with assisted code reading, and then the change is specified.
Context calibration	Paying for the same error over and over.	Section 4.5: the fix goes to the context, is tested in isolation and is versioned.

> **THE DIRECTIVE THAT KEEPS THE TEAM SPEC-ANCHORED**
>
> If an anomaly is detected, or the business changes the design during implementation, the spec or the context is updated first. The code is not patched. This is what prevents sliding back to spec-first with drift.

8.2 *Self-assessment*

The questionnaire places the team. A team is at a level when it answers yes to the whole corresponding block. A team with gaps in block 1 has not adopted SDD yet: it has adopted its vocabulary.

Table 14. Maturity questionnaire.

#	Block	Question (yes / no)
1	Spec-first	Does every new feature have a spec written before generating code?
2	Spec-first	Do the specs include preconditions, transitions, postconditions, criteria, out of scope and unknowns?
3	Spec-first	Is the project context versioned in the repository, and does the agent read it on every task?
4	Spec-first	Were story points and velocity retired from every board and report?
5	Spec-anchored	Are there named validators per gate, with time freed to validate?
6	Spec-anchored	Is it technically impossible to integrate without passing G3?
7	Spec-anchored	Are tests derived from the spec before the code exists, and do they run in CI?
8	Spec-anchored	Do production fixes and maintenance also start with the spec?
9	Spec-anchored	Does the pipeline or the review detect behavior changes without a spec change?
10	Spec-anchored	Did every repeated error end up as a context rule, tested before being merged?
11	Spec-anchored	Does every ceremony produce its artifact, and does the metrics dashboard trigger decisions?
12	Spec-anchored	Can a new team member understand a feature by reading only its spec?

9 Transition plan

The transition is not a process change that gets announced. It is a sequence of stages with verifiable exit criteria, run on a bounded pilot (one team and one product line) before it is extended. The sequence has four

stages: preparation, kickoff, calibration and steady operation. Along the way, six organizational change moves are completed.

9.1 *Stage 0 · Preparation: starting conditions*

The pilot does not start until the four conditions below are met. They can be worked on in parallel, but partial compliance does not allow the start: each missing condition shows up later as a more expensive failure.

Table 15. Starting conditions.

Condition	What is verified	If it fails
1. Role commitment	Named validators per gate, context owner and business validator, with their authority accepted in writing. "Named" means a specific person with authority to say "this is not valid".	Do not start. A gate without a real validator is paperwork.
2. Validation capacity	That the named people have real time to validate at the expected pace. It is the usual bottleneck. Capacity is declared as an assumption and measured at kickoff.	The pilot scope is reduced, not the validation.
3. Tools and environment	Coding agent, repository, CI pipeline and test environment working end to end. Any team member can run the cycle without waiting for permissions. The agent's access to models goes through an approved channel, with no personal keys.	The kickoff turns into a week of configuration.
4. Method setup	Repository structure created (section 4.1), minimum context versioned, spec template and G1 checklist available, branch protection configured.	The first specs are written without a standard and have to be redone.

In this same stage the team's baseline with Scrum is measured: cycle time, escaped defects and rework. Without a baseline, the improvement cannot be demonstrated and the conversation with the business stays in the realm of perceptions.

9.2 *Stage 1 · Kickoff: a vertical slice*

The smallest piece of real scope that meets three conditions is chosen: it goes through every layer of the system (interface, logic, data), it activates the three gates and it produces measurable artifacts. The kickoff is not a demo: it is a diagnosis. Its goal is to check that the work system works and to measure the assumptions the plan was built on.

Mandatory outputs: the full cycle run end to end; the first measurement of first-pass acceptance, rework, validation latency and compute cost; an open calibration backlog with the first failures recorded; and the measured validation capacity, which replaces the assumed one.

9.3 *Stage 2 · Calibration: stabilizing the system*

For four to six weeks the team scales to three or four specs in parallel, and each cycle follows the sequence plan, execute, validate, measure and calibrate. In this stage the mechanisms in Table 13 are installed and the team moves from spec-first to spec-anchored. Calibration (section 4.5) is the main activity: every repeated failure becomes a context rule, tested before being merged.

Exit criteria: the team runs the cycle without outside support; validators apply the criteria consistently with each other; validation capacity stays stable across cycles; there is no cost or quality drift; and the paths to scale to other teams are defined.

9.4 *Stage 3 · Steady operation*

Every new requirement enters as a traceable item and goes through the cycle in Figure 1. The daily is a calibration daily from day one, the retro audits the system and the dashboard in Table 12 triggers decisions. A simple operational signal: a task that has been "in progress" for more than a few hours is a red flag. Work in SDD flows; when it stalls, there is almost always a blocked agent or missing context.

9.5 *The six change moves*

In parallel with the stages, six moves are completed, each with an owner and a closing criterion. If any of them stays halfway, the result is a partial adoption, which fails worse than not transitioning at all (section 10).

Table 16. The six moves.

#	Move	What it implies	Closing criterion
1	**Reassign validation authority**	Name in writing who can say "this does not move forward" at each gate and free their time from building.	Named validators, with veto power and a percentage of time assigned.
2	**Convert the backlog gradually**	Classify by destination: what comes in soon, what is undefined intent and what no longer applies. Convert only the first group into specs. If there is not enough backlog, do the product definition first (brief, PRD, epics).	First specs approved at G1; the rest stays as intents.
3	**Write the minimum context**	The four context files, short, plus whatever the kickoff slice actually touches. The rest arrives through calibration.	Files versioned and reviewed by a person.
4	**Change the metrics**	Retire story points and velocity from every board and report on the same day. Publish the dashboard in Table 12.	Zero story points; new dashboard in use.
5	**Convert the ceremonies**	Apply the cards in section 6. They are not shortened: they change their product.	Each ceremony leaves its artifact for two cycles in a row.
6	**Prepare the client or the business**	Explain that the first weeks are for calibration, and bring someone from the business into validating the spec, not only the demo.	Business validator named and calibration curve accepted.

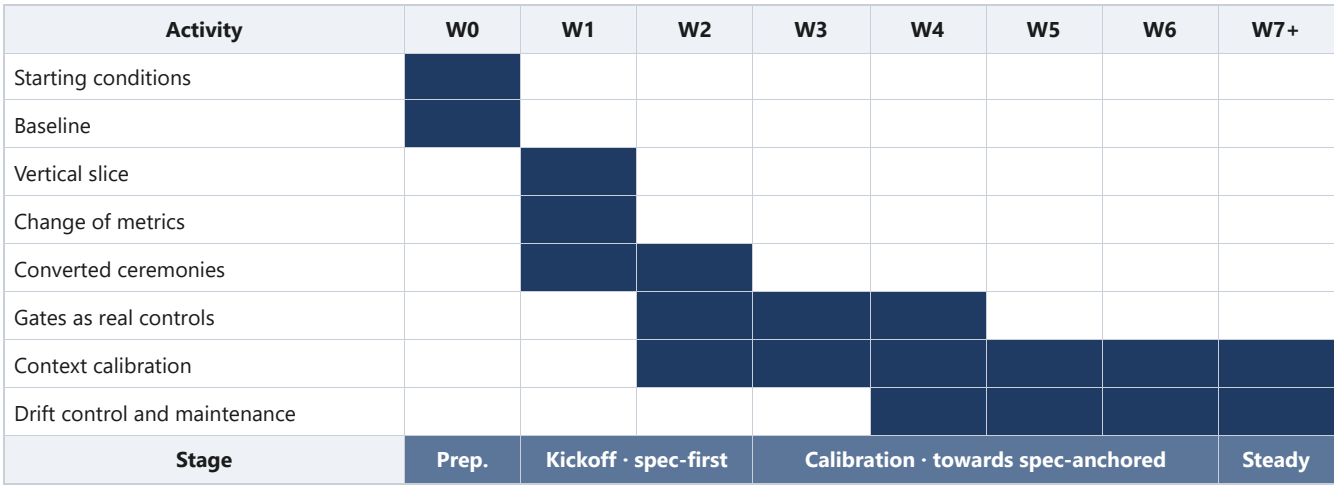


Figure 4. Indicative schedule for a pilot. The bottom band shows the stage and the expected maturity level. Spec-anchored consolidates when the first maintenance cycle arrives and fixes also enter through the spec.

ONE SPEC, ONE RUN, CALIBRATION, AND ONLY THEN VOLUME

Converting the whole backlog into specs before the first run is the most common kickoff mistake: twenty specs get written with the same defects the first run would have revealed.

10 Risks: partial adoption

The most frequent failure mode is not rejecting SDD: it is adopting it halfway. Each row in the table below describes a combination that looks like a transition but is not, together with the signal that makes it possible to detect it. It is recommended to walk through it in every retrospective during the first three months.

Table 17. Partial adoption combinations.

What was done halfway	What breaks	How it is detected
Specs yes, blocking gates no	Nobody reads the spec. Within two weeks the code becomes the source of truth again.	Commits that fix behavior without a prior spec change.
Gates yes, real authority no	The validator approves based on availability, not judgement. Risk is transferred with a signature.	Very low validation latency on complex artifacts.
Context yes, calibration no	The same error is paid for over and over; the cost per feature does not go down.	Same rejection reason repeated within the cycle.
Specs yes, story points too	Volume gets optimized, not validation.	Velocity and rework rising at the same time.
Specs yes, sprint as unit of delivery	The sprint becomes a queue that holds back finished work.	Approved features waiting for the end of the sprint to be integrated.
Specs yes, review criteria no	Validation by intuition: two validators approve different things.	Disagreements between validators on the same type of artifact.
"For urgencies" exception	The exception becomes the default path in the first difficult quarter.	Long conversations with the agent without a spec; direct fixes in production.
Team transitioned, business not	The demo becomes the de facto gate again, and with it the sprint.	Demo requests outside the flow; conversations about sprint dates.
Roles renamed, time not freed	The validator is one in title and a builder in practice. Gates turn into queues.	Validator utilization above the ceiling; latency rising.
Spec-as-source from the start	Over-engineering of tools before validating the method.	Weeks spent on tooling without a single feature end to end.

10.1 *Operational risks*

Besides adoption risks, working with agents brings technical risks of its own. None of them is solved with individual discipline: all of them require a written rule.

Table 18. Operational risks and mitigation.

Risk	What happens	Mitigation
Vibe coding	A vague instruction produces code that compiles and looks competent, but invents patterns: different spacing in neighboring components, off-brand colors, dependencies foreign to the stack. The speed is temporary; the debt, permanent.	Anchor the intent in a restrictive spec and in the context. G2 blocks it.

Context saturation	Long sessions fill the model's context window. Compaction interrupts the work and produces "amnesia": the agent forgets constraints, including security ones.	Short sessions per task; compact on purpose before the limit; keep decisions in files, not in the conversation.
Late security	Changing authentication, credentials or access when the generated logic is already large forces whole parts to be rebuilt.	security.md from day one; security in the first specs.
Inference cost	Heavy models in long sessions exhaust the budget.	Reasoning model for critical specs and plans; light model for structural implementation. Cost per validated feature is a health signal.
Agent permissions	An agent with direct file editing and broad permissions has a huge blast radius.	Progressive, auditable permissions. Never disable confirmations at the start; widen them only with demonstrated precision.
Key-person dependency	The flow works because someone has it in their head; if they leave, it cannot be replicated.	Method codified and versioned: templates, gates, context and criteria. What is not written does not exist.
Documentation fatigue	Exhaustive specs too early, with diminishing returns.	The retro adjusts the level of detail: the spec has to be enough for two engineers to build the same thing, no more.

11 Conclusion

SDD does not replace agility: it moves it to where it is cheap today. A team coming from Scrum already has most of what it needs: cadence, ceremonies, a culture of incremental delivery and the habit of inspecting and adapting. What changes is the object it iterates on (the spec), the mechanism that guarantees agreements (the gates) and what it measures (validation). The maturity model makes that transition observable: at spec-first the team writes before generating; at spec-anchored the spec is the living source of truth of the system and every change enters through it; spec-as-source stays as a horizon, not as a target.

The indicator of success is not perceived speed or the volume of generated code.

"The indicator of success is the absence of surprises. If the team feels calm, predictable and slightly bored, the method is working as designed."

A Annex: spec template

```
# <id> · <feature name>
Value: <user story or business outcome in one line>
Owner: <name> · Validators: business <name>, technical <name> · Risk: low | medium | high
## Context             what exists today and which part of the system is touched
## Preconditions       G1..Gn  what must be true before
## Transitions         T1..Tn  what changes, in what order, with which calculation rules
## Postconditions      P1..Pn  what must be true afterwards
## Invariants          I1..In  what can never stop being true
## Errors              E1..En  code · user message · what gets logged
## Acceptance criteria AC1..ACn  concrete examples, each with its test
## Non-functional requirements  performance, accessibility, security (measurable)
## Out of scope        what is explicitly not done
## Unknowns            question · owner · deadline  (blocks G1 until resolved)
## History             date · change · reason  (mandatory at spec-anchored)
```

B Annex: gate G1 checklist

✓	**Condition**
☐	The value statement says which business outcome is sought, not which screen is built.
☐	Every transition has explicit calculation rules (rounding, priorities, order).
☐	Every error has a code, a user message and logging behavior.
☐	Every acceptance criterion is a concrete example with values, not a description.
☐	What is not done is written in "Out of scope".
☐	No unknowns are left without an owner and an answer.
☐	The spec does not contradict any file in the project context.
☐	Two people who read it separately describe the same implementation.

References

Böckeler, B. (2025). Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl. martinfowler.com.
Boehm, B. W. (1981). Software Engineering Economics. Prentice Hall.
Brooker, M. (2026). Spec Driven Development isn't Waterfall. brooker.co.za.
GitHub (2025). Spec Kit: toolkit for Spec-Driven Development. github.com/github/spec-kit.
Hoare, C. A. R. (1969). An axiomatic basis for computer programming. Communications of the ACM, 12(10), 576–580.
Mavin, A., Wilkinson, P., Harwood, A. and Novak, M. (2009). Easy Approach to Requirements Syntax (EARS). 17th IEEE International Requirements Engineering Conference, 317–322.
Meyer, B. (1992). Applying "Design by Contract". IEEE Computer, 25(10), 40–51.
North, D. (2006). Introducing BDD. Better Software.
Schwaber, K. and Sutherland, J. (2020). The Scrum Guide. scrumguides.org.

© 2026 Jorge Roberts. All rights reserved. Cite as: Roberts, J. (2026). From Scrum to Spec-Driven Development (version 1.0). English translation of "De Scrum a Spec-Driven Development".