SDD, TDD, SPEC KIT AND SPECKY · PLAYBOOK

From vibe coding to a spec-driven discipline that ships production code.

Code that runs is not code that ships. Spec-Driven Development makes the spec the source of truth, EARS and TDD turn each requirement into a test the agent must pass, and a constitution keeps generated code secure by construction. Spec Kit supplies the methodology, Specky enforces it, and GitHub Copilot agent mode runs both, from the model picker to AI Credits.

AUTHOR

Paula Silva

ROLE

AI-Native Software Engineer

CONTACT

in/paulanunes

READING TIME

~ 45 minutes, end to end

5
MATURITY STAGES

6
CHAPTERS

6
EARS PATTERNS

4
SDD PHASES

CHAPTERS

Chapter 00

From vibe coding to production discipline

Why code that runs is not code that ships, the empirical case against uncritical adoption, the five-stage AI-native maturity curve, and why experienced developers do not vibe.

Problem →

Chapter 01

The spec is the source of truth

The fundamental inversion from code to intent, the four phases with owners and outputs, the anatomy of a well-formed spec, and SDD as the fifth and final maturity stage.

Inversion →

Chapter 02

EARS and TDD, the agent contract

EARS notation and its six patterns, turning one vague requirement into three testable ones, and the TDD Red-Green-Refactor cycle mapped onto the agent loop.

Contract →

Chapter 03

Spec Kit and Specky

The methodology layer of five slash commands and Markdown templates, the execution engine with 53 tools across a 10-phase pipeline, and the wiring into GitHub Copilot agent mode.

Toolchain →

Chapter 04

Constitutional SDD, security by construction

The four AI security failure modes, CONSTITUTION.md as non-negotiable principles in the spec layer, compliance traceability as audit evidence by construction, and the measured results.

Governance →

Chapter 05

The complete picture and adoption

Three project types on one pipeline, choosing the work between agents and humans, the 2026 ecosystem in one frame, and a concrete adoption path with metrics.

Adoption →

READING TIPS

KEYBOARD SHORTCUTS

Press `/` to search. Use `j` and `k` to move between chapters. Press `t` to toggle theme. Press `g` to go to top.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 THE PROBLEM

From vibe coding to production discipline.

In February 2025, Andrej Karpathy named a habit that engineers had been practicing for months. He called it vibe coding: you accept whatever the AI generates, run it, and move on, without critical review. The code usually runs. That is exactly the trap. Code that runs is not the same as code that is correct, secure, and reviewable. This chapter names the pattern, weighs the empirical evidence against adopting it uncritically, places it on a five-stage maturity curve, and shows what experienced developers do instead.

Vibe coding, named

Andrej Karpathy coined vibe coding in February 2025, in a post that spread fast because it described something real. You give in fully to whatever the model produces. You prompt, you accept, you run, you prompt again. GitHub Copilot in agent mode makes this frictionless: the agent edits files, runs commands, and opens a pull request while you watch. For a weekend prototype, that flow is fine. The problem starts when the same flow ships to production. What is harmless at prototype scale becomes a liability the moment other people depend on the result.

The failure is subtle because the code usually runs. Running is a weak signal. An SQL injection runs. It keeps running until someone exploits it. Code that works is not the same as code that is correct, secure, maintainable, and aligned with the requirement it was supposed to satisfy. Vibe coding optimizes for the first property and stays blind to the other four. Every uncritical retry in agent mode also consumes GitHub AI Credits, billed as one credit per US cent since June 1 2026, so the habit is not only risky, it is metered.

> DEFINITION
>
> Code that runs is not code that ships. Running proves nothing about correctness, security, or reviewability. Vibe coding optimizes the one property that is easy to observe and ignores the three that decide whether the code belongs in production.

The empirical case against uncritical adoption

Four studies set the stakes. Pearce and colleagues, at the 2022 IEEE Symposium on Security and Privacy, found that 40% of GitHub Copilot output was vulnerable in security-relevant scenarios. Liu and colleagues, in a 2026 study of 31,132 agent skills, found that 26.1% contained at least one security vulnerability. Becker and colleagues, in the 2025 METR study, measured experienced open-source developers working in large, mature repositories and found they were 19% slower with AI tools than without, when they used them without discipline.

The fourth study measures the finish line. Huang and colleagues, in a 2025 study of an issue tracker agent, found that only 8% of agent invocations reached a complete success, meaning a merged pull request. Read together, these numbers describe the gap between generation and delivery. The model can produce plausible code on the first try. Getting that code merged, secure, and trusted is a different problem, and it is the problem this playbook addresses. Switching between the models in the GitHub Copilot picker, from Haiku 4.5 to Opus 4.8, changes speed and cost, not the discipline that closes the gap. The binding constraint is the process around the agent, not the model inside it.

Five stages of AI-native maturity

Spec-Driven Development does not appear from nowhere. It sits at the end of a maturity curve that most teams climb in order. Each stage adds the structure the previous one lacked, and each absorbs the one before it rather than replacing it.

AI Assistant, Vibe Coding, Prompt Engineering, Context Engineering, Spec-Driven Development

The first stage is the AI Assistant: autocomplete and inline suggestions inside the editor. The second is Vibe Coding: full delegation without review, the stage this chapter warns against. The third is Prompt Engineering: shaping intent through carefully written instructions. The fourth is Context Engineering: feeding the agent the right files, standards, and history, often through a .github/copilot-instructions.md file and Model Context Protocol servers. The fifth and final stage is Spec-Driven Development, which adds a versioned, executable specification as the source of truth. It does not discard prompt engineering or context engineering. It absorbs them and puts a reviewable contract on top.

Why professionals do not vibe

The most useful finding in the Huang study is qualitative. The researchers interviewed 99 experienced developers about how they actually use coding agents. The pattern was consistent: professionals do not vibe. They plan before they delegate, and they supervise while the agent works. They keep the agent inside a boundary they defined, they read what it produces, and they reject what does not fit. Planning plus active supervision is what keeps a human in control of the outcome.

This is the practical bridge from problem to method. The discipline that experienced developers apply by instinct is exactly what Spec-Driven Development makes explicit and repeatable. The spec is the plan, written before the code. The tests are the supervision, run continuously against the spec. In GitHub Copilot terms, the specification and the constitution live in the repository, agent mode executes against them, and Specky enforces the phase gates through the Model Context Protocol. The rest of this playbook builds that discipline out, stage by stage.

> DEFINITION
>
> From the Huang study: as of now, the AIs are not taking over yet, experienced developers are still in control. The difference between vibe coding and professional practice is not the model. It is planning before delegation and supervision during execution.

References

Primary sources for the vibe coding pattern and the empirical evidence in this chapter.

- Andrej Karpathy, the original vibe coding post, February 2025, where the term was coined.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE Symposium on Security and Privacy 2022: 40% of output vulnerable in security-relevant scenarios.
- Liu et al., Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, 2026: 26.1% of 31,132 agent skills carried at least one vulnerability.
- Becker et al., Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, METR 2025: experienced developers 19% slower with AI in large, mature repositories.
- Huang et al., Professional Software Developers Don't Vibe, They Control, 2025: 8% of agent invocations reached a merged pull request, from a study of 99 experienced developers.
- Spec Kit, toolkit for Spec-Driven Development, the open-source methodology layer this playbook builds on.
- GitHub Copilot Documentation, the reference platform for agent mode, copilot-instructions, and the model picker.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 THE INVERSION

The spec is the source of truth: intent leads, code and tests follow.

Traditional development trusts the code and treats everything else as commentary. Spec-Driven Development reverses that trust. Intent is written down first, as a versioned specification, and code, tests, and documentation become artifacts derived from it. This chapter explains the inversion, walks the four phases from Specify to Implement, dissects the anatomy of a well-formed spec, and places SDD as the fifth and final stage of AI-native maturity, the one that moves review upstream to the spec.

The fundamental inversion: intent becomes the source of truth

Traditional development treats code as the source of truth. Documentation goes stale within weeks. Tests capture behavior that already happened. Decisions live in a chat history no one re-reads. Every new prompt to a coding agent starts from an empty context and reconstructs intent from whatever files it happens to open. SDD inverts this arrangement. The specification becomes the source of truth, and code, tests, and documentation become derived artifacts, each generated from the spec and traceable back to it.

The practical payoff is context that persists. When intent is written down as a versioned artifact, a GitHub Copilot agent does not re-infer goals on every session; it reads them. The spec travels with the work as durable context, so the reasoning that shaped Sprint 1 is still legible in Sprint 5. Review shifts as well. Instead of catching mistakes in a pull request after the code already exists, the team reviews the spec before a line is written, where a correction costs minutes rather than a rewrite. That is the whole point of the inversion: move the expensive decisions to the cheap end of the pipeline.

> DEFINITION
>
> The inversion in one line: the spec is the source of truth, and code, tests, and documentation are derived artifacts carried as persistent agent context. You review intent before implementation, not after.

The four phases: owners and outputs

Every SDD project flows through four phases, and each output is the input to the next. Specify comes first. A product owner defines what the system does and why, and the phase writes spec.md, which holds the user stories, the goals, and the success criteria. Plan comes second. An architect translates what into how, and the phase writes plan.md, which covers the architecture, the tech stack with its rationale, the data models, the API contracts, the latency targets, and the security specifics. Nothing in Plan is invented that the spec did not ask for.

Tasks comes third. A dev lead decomposes the plan into atomic, traceable units of work, and the phase writes tasks.md, marking every parallelizable task with a [P] marker so independent work can run at the same time. Implement comes fourth. A developer or a GitHub Copilot agent in agent mode executes the tasks, validates continuously against the spec, and submits the result for review. Spec Kit exposes each phase as a slash command, from /specify through /implement, and Specky runs the same sequence as an MCP server with gates that block phase-skipping. The methodology names the phases; the tooling enforces their order.

Anatomy of a well-formed spec

A spec earns its authority by being precise. It opens with a single-sentence goal, the North Star that states the outcome in one line. Below it sit numbered user stories in US-001 format, so every requirement has a stable identifier that later tasks and tests can cite. Success criteria come next, and they carry thresholds instead of adjectives. "Login under 200ms p95" is a criterion an agent and a test can both check. "Fast login" is an opinion, and an agent asked to satisfy an opinion will invent its own definition of fast.

The spec then draws its boundaries. Scope boundaries state what the system does not do, which stops an agent from filling the silence with features no one requested. Dependencies name the external services, libraries, and systems the work relies on, so nothing appears as a surprise during implementation. A sixth component closes the contract: EARS-formatted requirements, the testable statements the agent compiles against. Together these parts turn a paragraph of intent into an executable specification, and every downstream artifact traces back to a line the team agreed on.

SDD as the fifth and final maturity stage

AI-native development matured in stages. First came the AI Assistant, autocomplete inside the editor. Then Vibe Coding, accepting whatever the model produced without review. Then Prompt Engineering, shaping the output through carefully worded instructions. Then Context Engineering, feeding the agent the right files, skills, and repository instructions through .github/copilot-instructions.md and MCP servers. SDD is the fifth and final stage, and it is the one that makes the previous four accountable.

SDD does not discard the earlier stages. It absorbs prompt and context engineering and adds a versioned, executable spec layer on top. That layer changes where review happens. When intent is a durable artifact, the most valuable review moves upstream to the spec, before implementation begins. The METR study is the evidence for why this matters: experienced developers took 19% longer with AI tools in large, mature repositories, because the binding constraint was structured intent, not model capability. A reviewed spec supplies exactly that, and it does so before a single token of code is billed.

References

Primary sources for the Spec-Driven Development methodology, the four-phase workflow, and the empirical case for moving review upstream.

- GitHub Spec Kit, a toolkit for Spec-Driven Development, the open-source methodology that defines the four phases and the spec.md, plan.md, and

tasks.md artifacts.

- Specky, the MCP engine for Spec-Driven Development, runs the four-phase sequence programmatically with gates that block phase-skipping.
- GitHub Copilot Documentation, the reference platform for running the Implement phase in agent mode.
- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, the empirical evidence: experienced developers took 19% longer with AI tools in large, mature repositories.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

02 THE CONTRACT

Requirements the Agent Can Compile Against: EARS notation and TDD as the contract.

A requirement an agent cannot test is a requirement it cannot honor. GitHub Copilot in agent mode does not fail because it writes bad code; it fails because it fills the gaps in a vague requirement with guesses. This chapter turns intent into a contract the agent can compile against. EARS notation removes the ambiguity from the requirement. Test-Driven Development turns each requirement into a failing test the agent has to make pass. Together they give the agent a target it can hit and a signal that tells it when it has.

EARS notation: six patterns that remove ambiguity

Most requirements defeat the agent before a single token is generated. "The interface must be intuitive." "Performance must be adequate." "The system must be fast." None of these can be tested, so none of them constrain what the agent produces. The Easy Approach to Requirements Syntax, EARS, fixes this by forcing every requirement into one of six structured patterns. Each pattern has a trigger keyword and the same backbone, the system SHALL, followed by a measurable behavior. The keyword tells you which condition activates the behavior; the SHALL clause tells you exactly what the system does. There is no room left for interpretation.

The six patterns cover the full range of behavior a system exhibits, and the trigger keyword is what makes each one testable. Ubiquitous requirements use no keyword and state an always-true invariant: the system SHALL hash all passwords with bcrypt cost 12. Event-driven requirements use WHEN and fire on a discrete event: WHEN a user submits the login form, the system SHALL respond within 200ms p95. State-driven requirements use WHILE and hold during a sustained state: WHILE in

maintenance mode, the system SHALL return 503 to all requests. Optional requirements use WHERE and apply only when a feature is present: WHERE two-factor authentication is enabled, the system SHALL require a one-time code after the password. Unwanted requirements use IF and THEN to name an exceptional case and its response: IF login fails five times, THEN the system SHALL lock the account. Complex requirements combine a state and an event: WHILE processing a payment, WHEN a timeout occurs, the system SHALL queue a retry.

Why the keyword is the test

The trigger keyword is not decoration. It names the precondition the test has to set up before it asserts the behavior. WHEN a user submits the login form tells the test to arrange a submitted form; the system SHALL respond within 200ms p95 tells it what to assert. Because the mapping from pattern to test is mechanical, the requirement is machine-checkable. Specky validates EARS structure programmatically with Zod schemas, so a malformed requirement is caught in the spec, not discovered in production.

> DEFINITION
>
> Every EARS requirement reads the same way: an optional trigger keyword, the system SHALL, and a measurable behavior. If you cannot write a test that sets up the trigger and asserts the behavior, the requirement is not done.

From vague to testable

Consider one requirement a reviewer rejected: the login system must be fast and secure. It reads like a specification and specifies nothing. Fast for whom, measured how? Secure against what, verified how? An agent handed this requirement will produce code that looks plausible and satisfies neither property, because neither property was ever stated in terms anyone could check. The fix is not to add adjectives. It is to decompose the requirement into separate, measurable claims, each in an EARS pattern, each with a threshold instead of an adjective.

Fast and secure becomes three requirements. R-001, event-driven: WHEN a user submits login, the system SHALL respond within 200ms at p95. R-002, ubiquitous:

the system SHALL hash passwords with bcrypt cost 12. R-003, unwanted: IF login fails five times, THEN the system SHALL lock the account. Each one is testable: you can write an assertion that passes or fails. Each one is traceable: it carries an identifier that follows it from spec to test to the line of code that implements it and back to the pull request that shipped it. One vague sentence produced nothing an agent could target. Three EARS requirements produce three tests, and three tests are three targets the agent can hit and a reviewer can verify.

TDD in the AI era

Test-Driven Development predates AI agents by two decades, and its cycle maps onto the agent loop with almost no friction. The cycle has three steps. RED: write a test that expresses the desired behavior and watch it fail. If it does not fail, the test is wrong or the behavior already exists. GREEN: write the minimum code needed to make the test pass, nothing more. REFACTOR: improve the design and remove duplication while the tests stay green, and never add behavior in this step. The discipline is that behavior only enters through a failing test.

Where the agent fits the cycle

SDD supplies the spec, so the EARS acceptance criteria translate directly into test scaffolds. The developer reviews the scaffolds and runs them: RED. GitHub Copilot in agent mode generates the implementation against the failing tests, and RED becomes GREEN. This is the natural division of labor. The agent is good at producing code that satisfies an explicit target, and a failing test is the most explicit target there is. Refactoring stays human-led, because deciding what good design looks like is a judgment the spec does not encode, and because a refactor that changes behavior is a bug the tests should have prevented. The human owns the design; the agent fills in the implementation; the test suite arbitrates.

Tests as agent input

Without TDD, agent-generated code has no safety net. There is nothing to catch a refactor that quietly changes behavior, so refactoring becomes risky and rare. Edge cases that no one wrote a test for surface in production, where they are most

expensive to fix. Tests, if they get written at all, arrive after the code, which means they encode whatever the code already does rather than what the requirement demanded. The agent is fast, and without tests it is fast at producing code no one can safely change.

With TDD, the test is the agent's input, not an afterthought. The failing test is the feedback loop: the agent writes code, runs the test, reads the failure, and corrects, without a human in the inner loop for every attempt. Coverage is built in from the start because the test existed before the code. Edge cases are named in the spec as Unwanted requirements and become tests before they become incidents. The empirical case for discipline is on record: METR found that experienced developers were slower with AI tools in large, mature repositories, and the cause was missing structure, not weak models. A test suite is that structure. It is the contract the agent compiles against, and the signal that tells everyone when the contract is met.

> DEFINITION
>
> The test is not a check you run after the agent finishes. It is the specification the agent works toward. Write the test first and the agent has a target; write it last and you have only a description of whatever the agent happened to produce.

References

Primary sources for EARS notation, the SDD toolchain, and the empirical case for test-driven discipline with coding agents.

- Spec Kit, toolkit for Spec-Driven Development, the open-source methodology that turns EARS requirements and TDD steps into the executable contract agents follow.
- Specky, the MCP engine for SDD execution, validates EARS structure programmatically and drives the Red-Green-Refactor loop.
- GitHub Copilot Documentation, agent mode, the reference runtime that generates implementations against failing tests.
- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, the empirical evidence: experienced developers were

19% slower with AI tools in large, mature repositories, for lack of structure.

- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE S&P 2022: 40% of Copilot output was vulnerable in security-relevant scenarios, the case for testable security requirements.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

03 THE TOOLCHAIN

Spec Kit and Specky: the methodology layer and the execution engine.

Spec-Driven Development needs two things to work in practice: a methodology that agents can follow, and an engine that enforces it. GitHub Spec Kit supplies the first. Specky supplies the second. This chapter walks the division of labor between them, the ten phases they run through, and how the toolchain plugs into GitHub Copilot agent mode.

Spec Kit: the methodology layer

GitHub Spec Kit is the methodology layer of Spec-Driven Development. It ships as an open-source toolkit at github.com/github/spec-kit, and it works by adding a small set of slash commands to whatever AI agent you already use. Five commands drive the core workflow. /speckit.constitution writes the non-negotiable principles, security policies, and architectural constraints that every later artifact must honor. /speckit.specify turns a business problem statement into a spec of user stories and EARS requirements. /speckit.plan translates that spec into a technical plan with a chosen stack and its rationale. /speckit.tasks decomposes the plan into atomic, traceable tasks with parallel markers and TDD steps. /speckit.implement executes those tasks with a failing-test-first loop. Each command reads the output of the one before it, so the workflow is a chain, not a set of disconnected prompts.

Spec Kit carries this workflow as Markdown templates rather than as code, which is why the same commands run across more than 30 AI agents without change. The output is a five-file versioned project structure. A constitution.md lives under .specify/, a numbered feature folder under specs/ holds the spec.md, plan.md, and tasks.md for that feature, and generated code and tests land in src/ and tests/. Every file is plain Markdown, committed to Git, and reviewable in a normal pull request. The review moves upstream: a reviewer can reject a spec before a line of code exists,

which is far cheaper than catching the same mistake after the agent has written the implementation. Spec Kit defines what to write and in what order. It does not enforce that anyone actually follows the order.

Specky: the execution engine

Specky is the execution engine that supplies the enforcement Spec Kit leaves out. It ships at github.com/paulasilvatech/specky as an MCP server, so any agent that speaks the Model Context Protocol can call it. Specky exposes 53 tools organized across a 10-phase pipeline, and a state machine sits underneath them. The state machine blocks phase-skipping: you cannot jump from Init to Design without completing Specify first, and you cannot mark a phase done while its required output is missing. Where Spec Kit trusts the agent to follow the steps, Specky makes the steps a precondition the agent has to satisfy. The framing is that the AI is the operator and Specky is the engine: the agent runs through a validated pipeline instead of producing unstructured guesswork.

The validation is programmatic, not another prompt. Specky runs a real EARS validator that checks every requirement against the six EARS patterns and flags vague terms like fast, good, or intuitive before they reach code. Every tool input is validated with Zod schemas, so a malformed spec fails at the schema, not three phases later in a broken build. This is the difference between an agent that tries to follow instructions and an engine that verifies them: the state machine, the EARS regex, and the Zod schemas are code that either passes or does not. The result is a pipeline you can audit. Each phase produces a named artifact, each gate has a recorded decision, and nothing advances on the strength of the model sounding confident.

The ten phases, the gates, and the six ways in

Every Specky project runs the same ten phases: Init, Discover, Specify, Clarify, Design, Tasks, Analyze, Implement, Verify, and Release. Init creates the structure and the constitution and scans the codebase. Discover asks seven structured questions about scope, users, and constraints. Specify writes the EARS requirements with acceptance criteria. Clarify resolves ambiguities. Design produces the architecture,

data model, and API contracts. Tasks breaks the work down by user story. Analyze runs the cross-artifact consistency and compliance checks. Implement executes the tasks in order with checkpoints. Verify looks for drift between the spec and the code. Release generates the pull request, exports work items, and assembles the documentation. The pipeline is the same for greenfield, brownfield, and modernization work; only the starting point changes.

Human oversight is built into the pipeline as LGTM gates. After Specify, after Design, and after Tasks, the agent stops and asks you to review the artifact it just wrote; you reply LGTM to proceed or describe the changes you want. Verify is the fourth checkpoint: if it detects drift between the spec and the implementation, Specky routes you back to Specify to correct the divergence before anything ships. The gates are where a person stays in control. Reviewing a specification takes minutes, and it is the point in the pipeline where a wrong decision is cheapest to reverse.

Six ways to start a pipeline

A pipeline can begin from any of six inputs. Natural language is the simplest: you type the idea and the agent drafts the spec. A meeting transcript can be imported and parsed into requirements. A document in PDF or DOCX is converted to Markdown and fed in. A Figma design is turned into a requirements spec through the Figma MCP. A codebase scan initializes a brownfield project with context about the code that already exists. Whichever way you start, the ten phases that follow are identical, so the discipline does not depend on where the requirements came from.

Wiring the toolchain into GitHub Copilot

Inside GitHub Copilot, the toolchain runs through agent mode. Agent mode is the setting where Copilot plans a task, calls tools, and edits files across the repository rather than completing a single line, and it is what drives Specky through the pipeline. You register the engine by committing a .vscode/mcp.json to the repository root that points the specky server at npx specky-sdd. Because the file is in Git, everyone on the team gets the same MCP server automatically, with no per-developer setup. Repository-level guidance goes in .github/copilot-instructions.md, so the constitution and the conventions the agent should respect travel with the code.

Model choice happens in the Copilot picker, and Specky's phases give you a natural place to spend on it. The picker lists Claude Fable 5, Opus 4.8, Sonnet 5, and Haiku 4.5, among others, and you select per phase. The phases that carry the reasoning, Specify and Design, are worth a stronger model such as Opus 4.8 or Claude Fable 5; the mechanical phases, like Tasks or Release, run fine on a faster model such as Haiku 4.5. Usage is billed in GitHub AI Credits, where one credit is one US cent, so the cost of a run is visible and attributable to the phase that incurred it. Matching the model to the phase keeps the pipeline both rigorous and affordable.

> DEFINITION
>
> The division of labor is simple: Spec Kit defines the methodology, Specky enforces it, and GitHub Copilot agent mode runs both. Spec Kit gives you the five commands and the Markdown templates. Specky gives you the 53 tools, the state machine, and the programmatic EARS and Zod validation that make the pipeline auditable. Copilot supplies the agent, the picker models, and the AI Credits billing.

References

Primary sources for the Spec Kit methodology, the Specky engine, the Model Context Protocol it speaks, and the GitHub Copilot integration.

- GitHub Spec Kit, toolkit for Spec-Driven Development, the open-source methodology layer: five slash commands and the Markdown templates standardized across 30+ AI agents.
- Specky, the MCP engine for Spec-Driven Development, the execution engine: an MCP server with 53 tools, a 10-phase state machine, and programmatic EARS and Zod validation.
- Model Context Protocol Specification, the open standard Specky speaks so any compatible agent can call its tools.
- GitHub Copilot Documentation, the reference platform for agent mode, .vscode/mcp.json wiring, picker models, and GitHub AI Credits billing.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

04 GOVERNANCE LAYER

Security by construction, not by inspection.

AI code that runs is not the same as AI code that is safe. An SQL injection runs until someone exploits it. Most teams still treat security as a review step that happens after the agent writes the code, which means every defect is caught late or not at all. This chapter moves security into the spec layer, where a constitution constrains what the agent can generate before the first line exists.

Four recurring failure modes in AI-generated code

AI coding agents fail on security in predictable ways. Four modes recur across codebases. The first is injection vectors: string concatenation in SQL, unescaped HTML, and template literals built from user input. The code compiles and runs. It also hands an attacker a path into the database. The second is weak cryptographic defaults: MD5, SHA-1, ECB mode, and hardcoded keys. An agent reaches for whatever is common in its training data, and common is not the same as safe. A passing demo hides the flaw, because an exploit needs an attacker, not a happy path test.

The third mode is authorization gaps: missing access checks, insecure direct object references, and privilege escalation paths. The agent implements the happy path and skips the check that a caller owns the record it is editing. The fourth mode is silent error handling: a catch-all that returns a generic 500, swallows the stack trace, and writes nothing to an audit trail. Each mode produces code that works in a demo. Each mode ships a vulnerability. None of the four is a model reasoning failure. They are gaps in the constraints the agent was given. Pearce and colleagues found that 40% of GitHub Copilot output was vulnerable in security-relevant scenarios. A 2026 study of 31,132 agent skills found that 26.1% contained at least one security vulnerability. The failure modes are measured, not hypothetical.

CONSTITUTION.md: non-negotiable principles in the spec layer

The fix is to constrain the generation, not to inspect the output. Spec-Driven Development puts security in a CONSTITUTION.md file that lives in the spec layer, above spec, plan, and tasks. Spec Kit creates it with the /constitution command. The constitution holds non-negotiable principles that govern every artifact the project produces, and AI code generation stops operating in an unconstrained space. The intuition is plain. Just as constitutional law constrains what a government may do, a software constitution constrains what the agent may generate.

Each principle follows a fixed structure. It names the required behavior, for example that database queries must use parameterized statements. It names the CWE reference the behavior addresses, for example CWE-89 for SQL injection. It names the path where the trace lives, the file and lines that satisfy the principle. A principle carries an enforcement level, MUST, SHOULD, or MAY, and an implementation pattern that shows the accepted and the rejected forms. The constitution is versioned. A change to a principle increments its version, and every pull request checks compliance against it.

Carrying the constitution into agent mode

A constitution that lives only in the spec layer stops at the boundary of the agent session. GitHub Copilot in agent mode reads .github/copilot-instructions.md on every task. The constitution principles belong there too, restated as instructions the agent applies before it writes code. The CWE references and the required patterns become part of the context the agent carries into each session, not a checklist a reviewer applies afterward. The same principles reach any model in the Copilot picker, whether Claude Fable 5, Opus 4.8, Sonnet 5, Gemini 3.1 Pro, or a GPT-5.x model.

Compliance traceability: audit evidence by construction

Every downstream artifact must honor the constitution. The spec references the principles it depends on. The plan names the techniques that satisfy them. The tasks

carry them into implementation steps. The code maps back to them through a compliance traceability matrix: principle, CWE, file, lines, and technique, in one table. The mapping is bidirectional. From a principle you can find its implementation. From a file you can find the principles it must uphold.

This matrix does four things. It supports audit, because an auditor can confirm that each security requirement has a corresponding implementation. It supports change impact analysis, because a developer can see which principles a file edit touches. It reveals gaps, because a missing mapping is an unimplemented requirement. And it enables targeted regression testing when code changes. A reviewer stops hunting for what the agent might have missed and instead confirms a mapping that already exists. The evidence exists because the process produced it, not because someone reconstructed it in the week before an audit.

> DEFINITION
>
> Security by construction means the constraint precedes the code. Security by inspection means the review follows it. The first produces audit evidence as a byproduct of building. The second produces it as a scramble before a deadline.

Measured results

Marri 2026 reports a controlled case study that compares a constitutional workflow against an unconstrained one on the same banking application. The security defects detected fell from 11 to 3, a 73% reduction. The time to first secure build fell from 9 days to 4 days, 56% faster. Compliance documentation coverage rose from 23% to 100%. Security review iterations fell from 4 to 1. The workflow also produced more security code, 612 lines against 847, a 38% more thorough treatment of the same requirements.

The numbers point in one direction. Moving security into the spec layer does not slow delivery to buy safety. It produces both, because the agent generates inside the constraints instead of against them, and the review confirms compliance instead of discovering violations. The false choice between moving fast with AI and moving carefully with security dissolves once the constraints sit upstream of the code.

Build the right thing with the spec. Build it correctly with tests. Build it safely with the constitution.

References

Primary sources for the failure modes, the constitutional method, and the measured results.

- Marri, Constitutional Spec-Driven Development: Enforcing Security by Construction in AI-Assisted Code Generation, the controlled case study behind the measured results: security defects down from 11 to 3.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot Code Contributions, the evidence that 40% of Copilot output was vulnerable in security-relevant scenarios.
- Agent Skills in the Wild, An Empirical Study of Security Vulnerabilities at Scale, 26.1% of 31,132 agent skills contained at least one security vulnerability.
- MITRE CWE Top 25 Most Dangerous Software Weaknesses, the reference catalog for the CWE identifiers each constitutional principle names.
- OWASP Top 10, context for why injection stays a first-order risk in production code.
- GitHub Spec Kit, the toolkit whose /constitution command creates the principles file in the spec layer.
- GitHub Copilot Documentation, the reference for agent mode and the .github/copilot-instructions.md file that carries the constitution into each session.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 ADOPTION PATH

The complete picture: SDD plus TDD in production.

Spec-Driven Development, Test-Driven Development, Spec Kit, Specky, and Constitutional SDD are not five separate practices. They are one production discipline. This chapter puts them on a single pipeline, shows how to choose the work between agents and humans, frames the whole ecosystem in one sentence, and gives a concrete adoption path you can start this week with GitHub Copilot in agent mode. The goal is not faster code. The goal is the right code, correct and secure, faster.

Three project types, one pipeline

Most teams assume greenfield work and legacy work need different tools. They do not. Specky runs the same 10-phase pipeline for every project type. Only the entry point changes. Greenfield starts from nothing: the agent calls sdd_init, then sdd_discover asks a set of structured questions about scope, users, constraints, integrations, performance, security, and deployment. Brownfield starts from a codebase that already exists: the agent scans the repository, initializes the project with that context, and imports existing documents so the spec reflects what is already there. Modernization starts from a legacy system under migration: the agent scans the code, runs a batch import of documents, and ingests meeting transcripts so intent that lived only in conversation becomes part of the spec.

The value of one pipeline is that the discipline does not change when the project does. A developer who learned the specify, plan, tasks, implement flow on a greenfield feature applies the same flow to a legacy migration. In GitHub Copilot, agent mode drives the pipeline through MCP, and a repository-level .github/copilot-instructions.md file carries the conventions the agent reads on every task. The starting point is context. The pipeline is constant.

Choosing the work: what agents do, what humans keep

Production agentic workflows follow nine principles drawn from Bandara and colleagues. Prefer direct function calls over tool calls for deterministic operations like commits and timestamps. Prefer tool calls over MCP when determinism matters. Give one agent one tool and one responsibility rather than overloading a single agent with generation, validation, and transformation. Store prompts outside the code, in versioned files, so behavior changes without a code change; in GitHub Copilot that file is .github/copilot-instructions.md. Route work across more than one model and consolidate the outputs; the Copilot model picker makes this practical, letting you send a task to Claude Opus 4.8, Claude Sonnet 5, Gemini 3.1 Pro, or GPT-5.x. Separate the workflow from the MCP server, containerize the deployment, and keep the design simple. Agent runs in Copilot bill against GitHub AI Credits, where one credit equals one cent, so a simpler workflow is also a cheaper one.

The agent and human split

A survey of 99 experienced developers shows a consistent line between work that suits agents and work that does not. Agents do well on scaffolding and boilerplate, writing tests, general refactoring, and documentation; in the survey these tasks reported strong ratios of suitable to unsuitable votes, and scaffolding and documentation drew no unsuitable votes at all. Humans keep the work where a wrong guess is expensive: business logic and domain knowledge, security-critical code, and database design all landed firmly on the unsuitable side. The rule is simple. Delegate the work that is well defined and cheap to verify. Keep the work where judgment, context, and consequence dominate.

The 2026 ecosystem in one frame

The pieces fit into one sentence. Spec Kit is the open-source methodology: slash commands and Markdown templates that standardize how a spec is written. Specky is the enforcement engine: an MCP server with 53 tools, a state machine that blocks phase-skipping, and programmatic EARS validation. Constitutional SDD is the security layer: non-negotiable principles placed in the spec so that generated code is secure by construction, not by inspection. TDD is the quality layer: tests written

first, as agent input, so coverage is built in rather than bolted on. Each layer answers a different question. Together they cover the whole surface of production work.

Read the three questions in order and the roles stop overlapping. Are you building the right thing? That is the spec. Are you building it correctly? That is the test. Are you building it safely? That is the constitution. A team can adopt them one at a time, and each one earns its place before the next is added.

> DEFINITION
>
> SDD ensures you build the right thing. TDD ensures you build it correctly. Constitutional SDD ensures you build it safely.

A concrete adoption path

Do not migrate an entire codebase at once. Start with one isolated feature, ideally greenfield, and run the full Spec Kit workflow end to end: specify, plan, tasks, implement. Learn EARS next. The six EARS patterns are the highest-return skill on this path, because a requirement written in EARS is one an agent can compile against and a test can check. Then adopt a constitution: begin with 5 to 10 non-negotiable principles for your domain, each naming the behavior required and the CWE it addresses, and expand as you learn. When the team is comfortable, add Specky for programmatic enforcement and the 53 tools; the migration is incremental, not a rewrite. Throughout, GitHub Copilot in agent mode is the runtime, reading the constitution and the spec through .github/copilot-instructions.md and MCP.

What to measure

Two metrics tell you whether the discipline is working. The first is time to first secure build. In the banking case study behind Constitutional SDD, it fell from 9 days to 4, a 56% reduction. The second is defects caught before production. In the same study, security defects dropped from 11 to 3, a 73% reduction, security review iterations fell from 4 to 1, and compliance documentation coverage rose from 23% to 100%. Track those numbers from your first feature. They are the evidence that spec, test, and constitution are earning their place, not just adding process.

References

Primary sources for the shared pipeline, the split of work between agents and humans, the adoption metrics, and the tooling ecosystem described in this chapter.

- GitHub Spec Kit, a toolkit for Spec-Driven Development, the open-source methodology layer: slash commands and Markdown templates for writing specs.
- Specky, the MCP engine for Spec-Driven Development, the enforcement engine: 53 tools, a state machine, and programmatic EARS validation.
- Bandara et al., A Practical Guide for Designing, Developing, and Deploying Production-Grade Agentic AI Workflows, the source of the nine principles for agentic workflows.
- Huang et al., Professional Software Developers Do Not Vibe, They Control, the survey of 99 experienced developers behind the agent and human task split.
- Marri, Constitutional Spec-Driven Development: Enforcing Security by Construction, the case study behind the time to first secure build and the defect reduction numbers.
- GitHub Copilot Documentation, the reference runtime: agent mode, repository instructions, and MCP.
- Model Context Protocol Specification, the open standard Specky uses to expose its tools to agents.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps