SDD, TDD, SPEC KIT E SPECKY · PLAYBOOK

Do vibe coding a uma disciplina spec-driven que entrega código de produção.

Código que roda não é código que entra em produção. O Spec-Driven Development torna a spec a fonte da verdade, EARS e TDD transformam cada requisito em um teste que o agente precisa passar, e uma constituição mantém o código gerado seguro por construção. O Spec Kit fornece a metodologia, o Specky a impõe, e o agent mode do GitHub Copilot roda os dois, do seletor de modelos aos AI Credits.

AUTORA

Paula Silva

CARGO

AI-Native Software Engineer

CONTATO

in/paulanunes

TEMPO DE LEITURA

~ 45 minutos, fim a fim

5

ESTÁGIOS DE MATURIDADE

6

CAPÍTULOS

6

PADRÕES EARS

4

FASES DO SDD

CAPÍTULOS

Chapter 00

Do vibe coding à disciplina de produção

Por que código que roda não é código que entra em produção, a evidência empírica contra a adoção acrítica, a curva de maturidade AI-native de cinco estágios, e por que desenvolvedores experientes não fazem vibe.

Problema →

Chapter 01

A spec é a fonte da verdade

A inversão fundamental do código para a intenção, as quatro fases com donos e saídas, a anatomia de uma spec bem formada, e o SDD como quinto e último estágio de maturidade.

Inversão →

Chapter 02

EARS e TDD, o contrato do agente

A notação EARS e seus seis padrões, transformar um requisito vago em três testáveis, e o ciclo Red-Green-Refactor do TDD mapeado no loop do agente.

Contrato →

Chapter 03

Spec Kit e Specky

A camada de metodologia com cinco slash commands e templates em Markdown, o motor de execução com 53 ferramentas em um pipeline de 10 fases, e a integração com o agent mode do GitHub Copilot.

Ferramentas →

Chapter 04

Constitutional SDD, segurança por construção

Os quatro modos de falha de segurança da IA, o CONSTITUTION.md como princípios inegociáveis na camada de spec, a rastreabilidade de conformidade como evidência de auditoria por construção, e os resultados medidos.

Governança →

Chapter 05

O quadro completo e a adoção

Três tipos de projeto em um pipeline, escolher o trabalho entre agentes e humanos, o ecossistema de 2026 em um só quadro, e um caminho concreto de adoção com métricas.

Adoção →

DICAS DE LEITURA

ATALHOS DE TECLADO

Aperte `/` para buscar. Use `j` e `k` para mover entre capítulos. Aperte `t` para alternar tema. Aperte `g` para ir ao topo.

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 O PROBLEMA

Do vibe coding à disciplina de produção.

Em fevereiro de 2025, Andrej Karpathy nomeou um hábito que engenheiros vinham praticando havia meses. Ele o chamou de vibe coding: você aceita o que a IA gera, roda e segue em frente, sem revisão crítica. O código quase sempre roda. É exatamente essa a armadilha. Código que roda não é o mesmo que código correto, seguro e revisável. Este capítulo nomeia o padrão, pesa a evidência empírica contra adotá-lo sem crítica, o posiciona em uma curva de maturidade de cinco estágios e mostra o que desenvolvedores experientes fazem no lugar.

Vibe coding, nomeado

Andrej Karpathy cunhou o termo vibe coding em fevereiro de 2025, em um post que se espalhou rápido porque descrevia algo real. Você se entrega por completo ao que o modelo produz. Você faz o prompt, aceita, roda, faz outro prompt. O GitHub Copilot em agent mode torna isso sem atrito: o agente edita arquivos, roda comandos e abre um pull request enquanto você observa. Para um protótipo de fim de semana, esse fluxo está de bom tamanho. O problema começa quando o mesmo fluxo vai para produção. O que é inofensivo em escala de protótipo vira um passivo no instante em que outras pessoas dependem do resultado.

A falha é sutil porque o código costuma rodar. Rodar é um sinal fraco. Uma injeção de SQL roda. Continua rodando até alguém a explorar. Código que funciona não é o mesmo que código correto, seguro, sustentável e alinhado ao requisito que deveria atender. O vibe coding otimiza a primeira propriedade e fica cego às outras quatro. Toda nova tentativa acrítica em agent mode também consome GitHub AI Credits, cobrados a um crédito por centavo de dólar desde 1 de junho de 2026, então o hábito não é apenas arriscado, ele é medido.

DEFINIÇÃO

Código que roda não é código que entra em produção. Rodar não prova nada sobre correção, segurança ou revisabilidade. O vibe coding otimiza a única propriedade fácil de observar e ignora as três que decidem se o código pertence à produção.

A evidência empírica contra a adoção acrítica

Quatro estudos definem o tamanho do risco. Pearce e colegas, no IEEE Symposium on Security and Privacy de 2022, constataram que 40% da saída do GitHub Copilot era vulnerável em cenários relevantes para segurança. Liu e colegas, em um estudo de 2026 sobre 31.132 agent skills, constataram que 26,1% continham ao menos uma vulnerabilidade de segurança. Becker e colegas, no estudo da METR de 2025, mediram desenvolvedores open-source experientes trabalhando em repositórios grandes e maduros e constataram que eles ficavam 19% mais lentos com ferramentas de IA do que sem elas, quando as usavam sem disciplina.

O quarto estudo mede a linha de chegada. Huang e colegas, em um estudo de 2025 sobre um agente de issue tracker, constataram que apenas 8% das invocações do agente chegaram a um sucesso completo, ou seja, um pull request mergeado. Lidos em conjunto, esses números descrevem a distância entre gerar e entregar. O modelo consegue produzir código plausível na primeira tentativa. Fazer esse código ser mergeado, seguro e confiável é outro problema, e é o problema que este playbook ataca. Alternar entre os modelos no seletor do GitHub Copilot, do Haiku 4.5 ao Opus 4.8, muda velocidade e custo, não a disciplina que fecha essa distância. A restrição que limita é o processo em torno do agente, não o modelo dentro dele.

Cinco estágios de maturidade AI-native

Spec-Driven Development não surge do nada. Ele fica no fim de uma curva de maturidade que a maioria dos times sobe em ordem. Cada estágio acrescenta a estrutura que faltava ao anterior, e cada um absorve o anterior em vez de substituí-lo.

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

O primeiro estágio é o AI Assistant: autocompletar e sugestões inline dentro do editor. O segundo é o Vibe Coding: delegação total sem revisão, o estágio contra o qual este capítulo alerta. O terceiro é o Prompt Engineering: moldar a intenção por meio de instruções escritas com cuidado. O quarto é o Context Engineering: alimentar o agente com os arquivos, padrões e histórico certos, muitas vezes por um arquivo .github/copilot-instructions.md e servidores Model Context Protocol. O quinto e último estágio é o Spec-Driven Development, que acrescenta uma especificação versionada e executável como fonte da verdade. Ele não descarta prompt engineering nem context engineering. Ele os absorve e coloca um contrato revisável por cima.

Por que profissionais não fazem vibe

O achado mais útil do estudo de Huang é qualitativo. Os pesquisadores entrevistaram 99 desenvolvedores experientes sobre como de fato usam agentes de codificação. O padrão foi consistente: profissionais não fazem vibe. Eles planejam antes de delegar e supervisionam enquanto o agente trabalha. Eles mantêm o agente dentro de um limite que definiram, leem o que ele produz e rejeitam o que não serve. Planejamento mais supervisão ativa é o que mantém um humano no controle do resultado.

Essa é a ponte prática do problema para o método. A disciplina que desenvolvedores experientes aplicam por instinto é exatamente o que o Spec-Driven Development torna explícito e repetível. A spec é o plano, escrito antes do código. Os testes são a supervisão, rodados continuamente contra a spec. Em termos de GitHub Copilot, a especificação e a constitution ficam no repositório, o agent mode executa contra elas, e o Specky impõe os gates de fase pelo Model Context Protocol. O restante deste playbook constrói essa disciplina, estágio a estágio.

DEFINIÇÃO

Do estudo de Huang: por ora, as IAs ainda não estão assumindo o controle, os desenvolvedores experientes ainda mandam. A diferença entre vibe coding e prática profissional não é o modelo. É planejar antes de delegar e supervisionar durante a execução.

Referências

Fontes primárias para o padrão do vibe coding e a evidência empírica deste capítulo.

- Andrej Karpathy, the original vibe coding post, fevereiro de 2025, onde o termo foi cunhado.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE Symposium on Security and Privacy 2022: 40% da saída vulnerável em cenários relevantes para segurança.
- Liu et al., Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, 2026: 26,1% de 31.132 agent skills com ao menos uma vulnerabilidade.
- Becker et al., Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, METR 2025: desenvolvedores experientes 19% mais lentos com IA em repositórios grandes e maduros.
- Huang et al., Professional Software Developers Don't Vibe, They Control, 2025: 8% das invocações do agente chegaram a um pull request mergeado, em um estudo com 99 desenvolvedores experientes.
- Spec Kit, toolkit for Spec-Driven Development, a camada de metodologia open-source sobre a qual este playbook se apoia.
- GitHub Copilot Documentation, a plataforma de referência para agent mode, copilot-instructions e o seletor de modelos.

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 A INVERSÃO

A spec é a fonte da verdade: a intenção lidera, o código e os testes seguem.

O desenvolvimento tradicional confia no código e trata todo o resto como comentário. O Spec-Driven Development inverte essa confiança. A intenção é escrita primeiro, como uma especificação versionada, e código, testes e documentação passam a ser artefatos derivados dela. Este capítulo explica a inversão, percorre as quatro fases de Specify a Implement, disseca a anatomia de uma spec bem formada e posiciona o SDD como o quinto e último estágio da maturidade AI-native, aquele que move a revisão para cima, para a spec.

A inversão fundamental: a intenção vira a fonte da verdade

O desenvolvimento tradicional trata o código como a fonte da verdade. A documentação fica desatualizada em semanas. Os testes capturam um comportamento que já aconteceu. As decisões vivem em um histórico de chat que ninguém relê. Cada novo prompt para um agente de codificação começa de um contexto vazio e reconstrói a intenção a partir de quaisquer arquivos que ele por acaso abra. O SDD inverte esse arranjo. A especificação vira a fonte da verdade, e código, testes e documentação passam a ser artefatos derivados, cada um gerado a partir da spec e rastreável de volta a ela.

O ganho prático é um contexto que persiste. Quando a intenção está escrita como um artefato versionado, um agente do GitHub Copilot não reinfere os objetivos a cada sessão; ele os lê. A spec viaja junto com o trabalho como contexto durável, então o raciocínio que moldou a Sprint 1 ainda é legível na Sprint 5. A revisão também se desloca. Em vez de pegar erros em um pull request depois que o código já existe, o time revisa a spec antes de uma linha ser escrita, onde uma correção

custa minutos em vez de uma reescrita. É esse o sentido da inversão: mover as decisões caras para o lado barato do pipeline.

> DEFINIÇÃO
>
> A inversão em uma linha: a spec é a fonte da verdade, e código, testes e documentação são artefatos derivados carregados como contexto persistente do agente. Você revisa a intenção antes da implementação, não depois.

As quatro fases: donos e saídas

Todo projeto SDD flui por quatro fases, e cada saída é a entrada da seguinte. Specify vem primeiro. Um product owner define o que o sistema faz e por quê, e a fase escreve o spec.md, que guarda as user stories, os objetivos e os critérios de sucesso. Plan vem em segundo. Um arquiteto traduz o que em como, e a fase escreve o plan.md, que cobre a arquitetura, o tech stack com sua justificativa, os modelos de dados, os contratos de API, as metas de latência e os detalhes de segurança. Nada em Plan é inventado sem que a spec tenha pedido.

Tasks vem em terceiro. Um dev lead decompõe o plano em unidades de trabalho atômicas e rastreáveis, e a fase escreve o tasks.md, marcando cada tarefa paralelizável com um marcador [P] para que o trabalho independente possa rodar ao mesmo tempo. Implement vem em quarto. Um desenvolvedor ou um agente do GitHub Copilot em agent mode executa as tarefas, valida continuamente contra a spec e submete o resultado para revisão. O Spec Kit expõe cada fase como um slash command, de /specify a /implement, e o Specky roda a mesma sequência como um servidor MCP com gates que bloqueiam o pulo de fases. A metodologia nomeia as fases; a ferramenta impõe a ordem delas.

Anatomia de uma spec bem formada

Uma spec conquista sua autoridade sendo precisa. Ela abre com um objetivo de uma frase, a North Star que declara o resultado em uma linha. Abaixo dele ficam as user stories numeradas no formato US-001, para que cada requisito tenha um

identificador estável que tarefas e testes posteriores possam citar. Os critérios de sucesso vêm em seguida, e carregam limiares em vez de adjetivos. "Login em menos de 200ms p95" é um critério que um agente e um teste conseguem verificar. "Login rápido" é uma opinião, e um agente que precisa satisfazer uma opinião vai inventar a própria definição de rápido.

Depois a spec traça suas fronteiras. As fronteiras de escopo declaram o que o sistema não faz, o que impede um agente de preencher o silêncio com funcionalidades que ninguém pediu. As dependências nomeiam os serviços externos, as bibliotecas e os sistemas dos quais o trabalho depende, para que nada apareça como surpresa durante a implementação. Um sexto componente fecha o contrato: requisitos no formato EARS, as afirmações testáveis contra as quais o agente compila. Juntas, essas partes transformam um parágrafo de intenção em uma especificação executável, e cada artefato derivado remete de volta a uma linha que o time aceitou.

O SDD como o quinto e último estágio de maturidade

O desenvolvimento AI-native amadureceu em estágios. Primeiro veio o AI Assistant, o autocomplete dentro do editor. Depois o Vibe Coding, aceitar o que o modelo produzisse sem revisão. Depois o Prompt Engineering, moldar a saída por meio de instruções cuidadosamente redigidas. Depois o Context Engineering, alimentar o agente com os arquivos, as skills e as instruções de repositório certas por meio do .github/copilot-instructions.md e de servidores MCP. O SDD é o quinto e último estágio, e é o que torna os quatro anteriores responsáveis por seu resultado.

O SDD não descarta os estágios anteriores. Ele absorve o prompt e o context engineering e adiciona uma camada de spec versionada e executável por cima. Essa camada muda onde a revisão acontece. Quando a intenção é um artefato durável, a revisão mais valiosa se move para cima, para a spec, antes de a implementação começar. O estudo da METR é a evidência de por que isso importa: desenvolvedores experientes levaram 19% mais tempo com ferramentas de IA em repositórios grandes e maduros, porque a restrição que limitava era a intenção estruturada, não a capacidade do modelo. Uma spec revisada fornece exatamente isso, e faz isso antes de um único token de código ser cobrado.

Referências

Fontes primárias para a metodologia Spec-Driven Development, o fluxo de quatro fases e o argumento empírico para mover a revisão para cima.

- GitHub Spec Kit, a toolkit for Spec-Driven Development, a metodologia open-source que define as quatro fases e os artefatos spec.md, plan.md e tasks.md.
- Specky, the MCP engine for Spec-Driven Development, roda a sequência de quatro fases de forma programática com gates que bloqueiam o pulo de fases.
- GitHub Copilot Documentation, a plataforma de referência para rodar a fase Implement em agent mode.
- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, a evidência empírica: desenvolvedores experientes levaram 19% mais tempo com ferramentas de IA em repositórios grandes e maduros.

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 O CONTRATO

Requisitos contra os quais o agente pode compilar: notação EARS e TDD como contrato.

Um requisito que o agente não consegue testar é um requisito que ele não consegue honrar. O GitHub Copilot em agent mode não falha porque escreve código ruim; falha porque preenche as lacunas de um requisito vago com suposições. Este capítulo transforma a intenção em um contrato contra o qual o agente pode compilar. A notação EARS remove a ambiguidade do requisito. O Test-Driven Development transforma cada requisito em um teste que falha e que o agente precisa fazer passar. Juntos, dão ao agente um alvo que ele consegue acertar e um sinal que diz quando acertou.

Notação EARS: seis padrões que removem a ambiguidade

A maioria dos requisitos derrota o agente antes de gerar um único token. "A interface deve ser intuitiva." "O desempenho deve ser adequado." "O sistema deve ser rápido." Nenhum deles pode ser testado, então nenhum restringe o que o agente produz. A Easy Approach to Requirements Syntax, EARS, corrige isso forçando todo requisito a um de seis padrões estruturados. Cada padrão tem uma palavra-gatilho e a mesma espinha dorsal, o sistema DEVE, seguida de um comportamento mensurável. A palavra-gatilho diz qual condição ativa o comportamento; a cláusula DEVE diz exatamente o que o sistema faz. Não sobra espaço para interpretação.

Os seis padrões cobrem toda a gama de comportamento que um sistema exibe, e a palavra-gatilho é o que torna cada um testável. Requisitos ubíquos não usam palavra-gatilho e declaram um invariante sempre verdadeiro: o sistema DEVE hashear todas as senhas com bcrypt cost 12. Requisitos orientados a evento usam QUANDO e disparam em um evento discreto: QUANDO um usuário submete o formulário de login, o sistema DEVE responder em até 200ms p95. Requisitos

orientados a estado usam ENQUANTO e valem durante um estado sustentado: ENQUANTO em modo de manutenção, o sistema DEVE retornar 503 a todas as requisições. Requisitos opcionais usam ONDE e se aplicam apenas quando uma feature está presente: ONDE a autenticação de dois fatores está habilitada, o sistema DEVE exigir um código de uso único após a senha. Requisitos indesejados usam SE e ENTÃO para nomear um caso excepcional e sua resposta: SE o login falhar cinco vezes, ENTÃO o sistema DEVE bloquear a conta. Requisitos complexos combinam um estado e um evento: ENQUANTO processa um pagamento, QUANDO ocorre um timeout, o sistema DEVE enfileirar uma nova tentativa.

Por que a palavra-gatilho é o teste

A palavra-gatilho não é enfeite. Ela nomeia a pré-condição que o teste precisa montar antes de verificar o comportamento. QUANDO um usuário submete o formulário de login diz ao teste para preparar um formulário submetido; o sistema DEVE responder em até 200ms p95 diz o que verificar. Como o mapeamento de padrão para teste é mecânico, o requisito é verificável por máquina. O Specky valida a estrutura EARS programaticamente com schemas Zod, então um requisito malformado é pego na spec, não descoberto em produção.

> DEFINIÇÃO
>
> Todo requisito EARS se lê da mesma forma: uma palavra-gatilho opcional, o sistema DEVE e um comportamento mensurável. Se você não consegue escrever um teste que monta o gatilho e verifica o comportamento, o requisito não está pronto.

De vago a testável

Considere um requisito que um revisor rejeitou: o sistema de login deve ser rápido e seguro. Parece uma especificação e não especifica nada. Rápido para quem, medido como? Seguro contra o quê, verificado como? Um agente que recebe esse requisito vai produzir código que parece plausível e não satisfaz nenhuma das duas propriedades, porque nenhuma delas foi declarada em termos que alguém pudesse checar. A correção não é adicionar adjetivos. É decompor o requisito em afirmações

separadas e mensuráveis, cada uma em um padrão EARS, cada uma com um threshold em vez de um adjetivo.

Rápido e seguro vira três requisitos. R-001, orientado a evento: QUANDO um usuário submete login, o sistema DEVE responder em até 200ms no p95. R-002, ubíquo: o sistema DEVE hashear senhas com bcrypt cost 12. R-003, indesejado: SE o login falhar cinco vezes, ENTÃO o sistema DEVE bloquear a conta. Cada um é testável: você consegue escrever uma asserção que passa ou falha. Cada um é rastreável: carrega um identificador que o acompanha da spec ao teste, à linha de código que o implementa e de volta ao pull request que o entregou. Uma frase vaga não produziu nada que o agente pudesse mirar. Três requisitos EARS produzem três testes, e três testes são três alvos que o agente consegue acertar e que um revisor consegue verificar.

TDD na era da IA

O Test-Driven Development é duas décadas mais antigo que os agentes de IA, e seu ciclo mapeia para o loop do agente quase sem atrito. O ciclo tem três passos. RED: escreva um teste que expressa o comportamento desejado e veja-o falhar. Se não falhar, o teste está errado ou o comportamento já existe. GREEN: escreva o mínimo de código necessário para o teste passar, nada além disso. REFACTOR: melhore o design e remova duplicação enquanto os testes seguem verdes, e nunca adicione comportamento neste passo. A disciplina é que comportamento só entra por um teste que falha.

Onde o agente entra no ciclo

O SDD fornece a spec, então os critérios de aceite em EARS traduzem-se diretamente em esqueletos de teste. O desenvolvedor revisa os esqueletos e os roda: RED. O GitHub Copilot em agent mode gera a implementação contra os testes que falham, e RED vira GREEN. Essa é a divisão natural de trabalho. O agente é bom em produzir código que satisfaz um alvo explícito, e um teste que falha é o alvo mais explícito que existe. O refactoring permanece liderado por humano, porque decidir o que é bom design é um julgamento que a spec não codifica, e porque um refactor que muda o comportamento é um bug que os testes deveriam ter evitado. O humano é dono do design; o agente preenche a implementação; a suíte de testes arbitra.

Testes como entrada do agente

Sem TDD, o código gerado pelo agente não tem rede de segurança. Não há nada para pegar um refactor que muda o comportamento em silêncio, então o refactoring vira algo arriscado e raro. Casos de borda para os quais ninguém escreveu teste surgem em produção, onde são mais caros de corrigir. Os testes, quando são escritos, chegam depois do código, o que significa que codificam o que o código já faz em vez do que o requisito exigia. O agente é rápido, e sem testes ele é rápido em produzir código que ninguém consegue mudar com segurança.

Com TDD, o teste é a entrada do agente, não um pensamento tardio. O teste que falha é o loop de feedback: o agente escreve código, roda o teste, lê a falha e corrige, sem um humano no loop interno a cada tentativa. A cobertura é nativa desde o início porque o teste existia antes do código. Casos de borda são nomeados na spec como requisitos indesejados e viram testes antes de virarem incidentes. A evidência a favor da disciplina está registrada: a METR constatou que desenvolvedores experientes ficaram mais lentos com ferramentas de IA em repositórios grandes e maduros, e a causa era a falta de estrutura, não a fraqueza dos modelos. Uma suíte de testes é essa estrutura. É o contrato contra o qual o agente compila, e o sinal que diz a todos quando o contrato foi cumprido.

> DEFINIÇÃO
>
> O teste não é uma verificação que você roda depois que o agente termina. É a especificação em direção à qual o agente trabalha. Escreva o teste primeiro e o agente tem um alvo; escreva-o por último e você tem apenas uma descrição do que o agente acabou produzindo.

Referências

Fontes primárias para a notação EARS, o toolchain de SDD e a evidência empírica a favor da disciplina orientada a testes com agentes de codificação.

- Spec Kit, toolkit para Spec-Driven Development, a metodologia open-source que transforma requisitos EARS e passos de TDD no contrato executável que os agentes seguem.

- Specky, o motor MCP de execução de SDD, valida a estrutura EARS programaticamente e conduz o loop Red-Green-Refactor.
- Documentação do GitHub Copilot, o agent mode, runtime de referência que gera implementações contra testes que falham.
- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, a evidência empírica: desenvolvedores experientes ficaram 19% mais lentos com ferramentas de IA em repositórios grandes e maduros, por falta de estrutura.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE S&P 2022: 40% do output do Copilot foi vulnerável em cenários relevantes de segurança, o argumento para requisitos de segurança testáveis.

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 A CADEIA DE FERRAMENTAS

Spec Kit e Specky: a camada de metodologia e o motor de execução.

Spec-Driven Development precisa de duas coisas para funcionar na prática: uma metodologia que os agentes consigam seguir, e um motor que a imponha. O GitHub Spec Kit fornece a primeira. O Specky fornece o segundo. Este capítulo percorre a divisão de trabalho entre eles, as dez fases pelas quais passam, e como a cadeia de ferramentas se conecta ao agent mode do GitHub Copilot.

Spec Kit: a camada de metodologia

O GitHub Spec Kit é a camada de metodologia do Spec-Driven Development. Ele é distribuído como um toolkit open-source em github.com/github/spec-kit, e funciona adicionando um pequeno conjunto de slash commands a qualquer agente de IA que você já use. Cinco comandos conduzem o fluxo principal. O /speckit.constitution escreve os princípios inegociáveis, as políticas de segurança e as restrições de arquitetura que todo artefato posterior precisa honrar. O /speckit.specify transforma a descrição de um problema de negócio em uma spec de user stories e requisitos EARS. O /speckit.plan traduz essa spec em um plano técnico com uma stack escolhida e sua justificativa. O /speckit.tasks decompõe o plano em tarefas atômicas, rastreáveis, com marcadores de paralelismo e passos de TDD. O /speckit.implement executa essas tarefas com um loop de teste-que-falha-primeiro. Cada comando lê a saída do anterior, então o fluxo é uma cadeia, não um conjunto de prompts desconexos.

O Spec Kit carrega esse fluxo como templates em Markdown, não como código, e é por isso que os mesmos comandos rodam em mais de 30 agentes de IA sem alteração. A saída é uma estrutura de projeto versionada de cinco arquivos. Um constitution.md fica em .specify/, uma pasta de feature numerada em specs/ guarda o spec.md, o plan.md e o tasks.md daquela feature, e o código e os testes gerados vão para src/ e tests/. Cada arquivo é Markdown puro, comitado no Git e revisável em um pull request comum. A revisão sobe para o topo: um revisor pode rejeitar uma

spec antes de existir uma linha de código, o que é muito mais barato do que pegar o mesmo erro depois que o agente já escreveu a implementação. O Spec Kit define o que escrever e em que ordem. Ele não impõe que alguém de fato siga a ordem.

Specky: o motor de execução

O Specky é o motor de execução que fornece a imposição que o Spec Kit deixa de fora. Ele é distribuído em github.com/paulasilvatech/specky como um servidor MCP, então qualquer agente que fale o Model Context Protocol consegue chamá-lo. O Specky expõe 53 ferramentas organizadas em um pipeline de 10 fases, e uma máquina de estados fica por baixo delas. A máquina de estados bloqueia o pulo de fases: você não pode saltar de Init para Design sem concluir Specify primeiro, e não pode marcar uma fase como concluída enquanto sua saída obrigatória estiver faltando. Onde o Spec Kit confia que o agente vai seguir os passos, o Specky torna os passos uma precondição que o agente tem de satisfazer. O enquadramento que o projeto usa é que a IA é o operador e o Specky é o motor, de modo que o agente roda por um pipeline validado em vez de produzir palpites sem estrutura.

A validação é programática, não mais um prompt. O Specky roda um validador EARS de verdade, que checa cada requisito contra os seis padrões EARS e sinaliza termos vagos como rápido, bom ou intuitivo antes que cheguem ao código. Toda entrada de ferramenta é validada com Zod schemas, então uma spec malformada falha no schema, não três fases depois em um build quebrado. Essa é a diferença entre um agente que tenta seguir instruções e um motor que as verifica: a máquina de estados, o regex do EARS e os Zod schemas são código que passa ou não passa. O resultado é um pipeline que você pode auditar. Cada fase produz um artefato nomeado, cada gate tem uma decisão registrada, e nada avança na base de o modelo soar confiante.

As dez fases, os gates e as seis portas de entrada

Todo projeto Specky roda as mesmas dez fases: Init, Discover, Specify, Clarify, Design, Tasks, Analyze, Implement, Verify e Release. O Init cria a estrutura e a constituição e escaneia a base de código. O Discover faz sete perguntas

estruturadas sobre escopo, usuários e restrições. O Specify escreve os requisitos EARS com critérios de aceitação. O Clarify resolve ambiguidades. O Design produz a arquitetura, o modelo de dados e os contratos de API. O Tasks quebra o trabalho por user story. O Analyze roda a análise de consistência entre artefatos e as verificações de compliance. O Implement executa as tarefas em ordem, com checkpoints. O Verify procura drift entre a spec e o código. O Release gera o pull request, exporta os work items e monta a documentação. O pipeline é o mesmo para trabalho greenfield, brownfield e de modernização; só o ponto de partida muda.

A supervisão humana está embutida no pipeline como LGTM gates. Depois de Specify, depois de Design e depois de Tasks, o agente para e pede que você revise o artefato que acabou de escrever; você responde LGTM para prosseguir ou descreve as mudanças que quer. O Verify é o quarto checkpoint: se detecta drift entre a spec e a implementação, o Specky te encaminha de volta ao Specify para corrigir a divergência antes que qualquer coisa vá para produção. Os gates são onde uma pessoa mantém o controle do trabalho. Revisar uma especificação leva minutos, e é o ponto do pipeline em que uma decisão errada é a mais barata de reverter.

Seis formas de iniciar um pipeline

Um pipeline pode começar por qualquer uma de seis entradas. Linguagem natural é a mais simples: você digita a ideia e o agente rascunha a spec. Uma transcrição de reunião pode ser importada e convertida em requisitos. Um documento em PDF ou DOCX é convertido para Markdown e alimentado no pipeline. Um design do Figma é transformado em uma spec de requisitos através do Figma MCP. Um scan da base de código inicializa um projeto brownfield com contexto sobre o código que já existe. Não importa por onde você comece, as dez fases que seguem são idênticas, então a disciplina não depende de onde vieram os requisitos.

Conectando a cadeia de ferramentas ao GitHub Copilot

Dentro do GitHub Copilot, a cadeia de ferramentas roda pelo agent mode. O agent mode é o modo em que o Copilot planeja uma tarefa, chama ferramentas e edita arquivos por todo o repositório em vez de completar uma única linha, e é ele que

conduz o Specky pelo pipeline. Você registra o motor comitando um .vscode/mcp.json na raiz do repositório que aponta o servidor specky para npx specky-sdd. Como o arquivo está no Git, todo mundo no time recebe o mesmo servidor MCP automaticamente, sem configuração por desenvolvedor. A orientação a nível de repositório vai em .github/copilot-instructions.md, então a constituição e as convenções que o agente deve respeitar viajam junto com o código.

A escolha de modelo acontece no picker do Copilot, e as fases do Specky te dão um lugar natural para gastar nisso. O picker lista Claude Fable 5, Opus 4.8, Sonnet 5 e Haiku 4.5, entre outros, e você seleciona por fase. As fases que carregam o raciocínio, Specify e Design, valem um modelo mais forte como Opus 4.8 ou Claude Fable 5; as fases mecânicas, como Tasks ou Release, rodam bem em um modelo mais rápido como Haiku 4.5. O uso é cobrado em GitHub AI Credits, onde um crédito é um centavo de dólar, então o custo de uma execução fica visível e atribuível à fase que o gerou. Casar o modelo com a fase é como você mantém o pipeline ao mesmo tempo rigoroso e acessível.

> DEFINIÇÃO
>
> A divisão de trabalho é simples: o Spec Kit define a metodologia, o Specky a impõe, e o agent mode do GitHub Copilot roda os dois. O Spec Kit te dá os cinco comandos e os templates em Markdown. O Specky te dá as 53 ferramentas, a máquina de estados e a validação programática de EARS e Zod que tornam o pipeline auditável. O Copilot fornece o agente, os modelos do picker e a cobrança em AI Credits que fazem tudo rodar no dia a dia.

Referências

Fontes primárias para a metodologia do Spec Kit, o motor Specky, o Model Context Protocol que ele fala e a integração com o GitHub Copilot.

- GitHub Spec Kit, toolkit para Spec-Driven Development, a camada de metodologia open-source: cinco slash commands e os templates em Markdown padronizados em mais de 30 agentes de IA.

- Specky, o motor MCP para Spec-Driven Development, o motor de execução: um servidor MCP com 53 ferramentas, uma máquina de estados de 10 fases e validação programática de EARS e Zod.
- Model Context Protocol Specification, o padrão aberto que o Specky fala para que qualquer agente compatível chame suas ferramentas.
- GitHub Copilot Documentation, a plataforma de referência para agent mode, a configuração do .vscode/mcp.json, os modelos do picker e a cobrança em GitHub AI Credits.

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 CAMADA DE GOVERNANÇA

Segurança por construção, não por inspeção.

Código de IA que roda não é o mesmo que código de IA que é seguro. Uma injeção de SQL roda até alguém explorá-la. A maioria dos times ainda trata segurança como uma etapa de revisão que acontece depois que o agente escreve o código, o que significa que todo defeito é pego tarde ou não é pego. Este capítulo leva a segurança para a camada de spec, onde uma constituição restringe o que o agente pode gerar antes de a primeira linha existir.

Quatro modos de falha recorrentes em código gerado por IA

Agentes de codificação de IA falham em segurança de formas previsíveis. Quatro modos se repetem entre bases de código. O primeiro são os vetores de injeção: concatenação de strings em SQL, HTML sem escape e template literals construídos a partir de entrada do usuário. O código compila e roda. Também entrega a um atacante um caminho para dentro do banco de dados. O segundo são os padrões criptográficos fracos: MD5, SHA-1, modo ECB e chaves fixas no código. O agente recorre ao que é comum nos seus dados de treinamento, e comum não é o mesmo que seguro. Uma demo que passa esconde a falha, porque um exploit precisa de um atacante, não de um teste de caminho feliz.

O terceiro modo são as lacunas de autorização: checagens de acesso ausentes, referências diretas inseguras a objetos e caminhos de escalonamento de privilégio. O agente implementa o caminho feliz e pula a checagem de que quem chama é dono do registro que está editando. O quarto modo é o tratamento silencioso de erros: um catch-all que retorna um 500 genérico, engole o stack trace e não escreve nada em uma trilha de auditoria. Cada modo produz código que funciona em uma demo. Cada modo entrega uma vulnerabilidade. Nenhum dos quatro é uma falha de raciocínio do modelo. São lacunas nas restrições que o agente recebeu. Pearce e

colegas constataram que 40% da saída do GitHub Copilot era vulnerável em cenários relevantes para segurança. Um estudo de 2026 com 31.132 agent skills constatou que 26,1% continham ao menos uma vulnerabilidade de segurança. Os modos de falha são medidos, não hipotéticos.

CONSTITUTION.md: princípios inegociáveis na camada de spec

A correção é restringir a geração, não inspecionar a saída. O Spec-Driven Development coloca a segurança em um arquivo CONSTITUTION.md que vive na camada de spec, acima de spec, plan e tasks. O Spec Kit o cria com o comando /constitution. A constituição guarda princípios inegociáveis que governam todo artefato que o projeto produz, e a geração de código por IA deixa de operar em um espaço sem restrições. A intuição é simples. Assim como a lei constitucional restringe o que um governo pode fazer, uma constituição de software restringe o que o agente pode gerar.

Cada princípio segue uma estrutura fixa. Nomeia o comportamento exigido, por exemplo que consultas ao banco de dados devem usar declarações parametrizadas. Nomeia a referência CWE que o comportamento endereça, por exemplo CWE-89 para injeção de SQL. Nomeia o caminho onde vive o rastro, o arquivo e as linhas que satisfazem o princípio. Um princípio carrega um nível de imposição, MUST, SHOULD ou MAY, e um padrão de implementação que mostra a forma aceita e a rejeitada. A constituição é versionada. Uma mudança em um princípio incrementa sua versão, e todo pull request checa a conformidade contra ela.

Levando a constituição para o agent mode

Uma constituição que vive só na camada de spec para na fronteira da sessão do agente. O GitHub Copilot em agent mode lê o .github/copilot-instructions.md em cada tarefa. Os princípios da constituição pertencem ali também, reescritos como instruções que o agente aplica antes de escrever código. As referências CWE e os padrões exigidos passam a fazer parte do contexto que o agente carrega para cada sessão, não uma lista de verificação que um revisor aplica depois. Os mesmos princípios chegam a qualquer modelo no seletor do Copilot, seja Claude Fable 5, Opus 4.8, Sonnet 5, Gemini 3.1 Pro ou um modelo GPT-5.x.

Rastreabilidade de conformidade: evidência de auditoria por construção

Todo artefato downstream precisa honrar a constituição. A spec referencia os princípios de que depende. O plan nomeia as técnicas que os satisfazem. As tasks os levam para as etapas de implementação. O código mapeia de volta para eles por uma matriz de rastreabilidade de conformidade: princípio, CWE, arquivo, linhas e técnica, em uma única tabela. O mapeamento é bidirecional. A partir de um princípio você encontra sua implementação. A partir de um arquivo você encontra os princípios que ele precisa sustentar.

Essa matriz faz quatro coisas. Apoia a auditoria, porque um auditor pode confirmar que cada requisito de segurança tem uma implementação correspondente. Apoia a análise de impacto de mudanças, porque um desenvolvedor consegue ver quais princípios uma edição de arquivo toca. Revela lacunas, porque um mapeamento ausente é um requisito não implementado. E habilita testes de regressão direcionados quando o código muda. Um revisor deixa de caçar o que o agente possa ter esquecido e passa a confirmar um mapeamento que já existe. A evidência existe porque o processo a produziu, não porque alguém a reconstruiu na semana anterior a uma auditoria.

> DEFINIÇÃO
>
> Segurança por construção significa que a restrição precede o código. Segurança por inspeção significa que a revisão vem depois dele. A primeira produz evidência de auditoria como subproduto de construir. A segunda a produz como uma correria antes de um prazo.

Resultados medidos

Marri 2026 relata um estudo de caso controlado que compara um fluxo constitucional com um fluxo sem restrições sobre a mesma aplicação bancária. Os defeitos de segurança detectados caíram de 11 para 3, uma redução de 73%. O tempo até o primeiro build seguro caiu de 9 dias para 4 dias, 56% mais rápido. A cobertura de documentação de conformidade subiu de 23% para 100%. As

iterações de revisão de segurança caíram de 4 para 1. O fluxo também produziu mais código de segurança, 612 linhas contra 847, um tratamento 38% mais completo dos mesmos requisitos.

Os números apontam em uma direção. Levar a segurança para a camada de spec não desacelera a entrega para comprar segurança. Produz as duas coisas, porque o agente gera dentro das restrições em vez de contra elas, e a revisão confirma a conformidade em vez de descobrir violações. A falsa escolha entre andar rápido com IA e andar com cuidado com segurança se dissolve quando as restrições ficam a montante do código. Construa a coisa certa com a spec. Construa-a corretamente com testes. Construa-a com segurança com a constituição.

Referências

Fontes primárias para os modos de falha, o método constitucional e os resultados medidos.

- Marri, Constitutional Spec-Driven Development: Enforcing Security by Construction in AI-Assisted Code Generation, o estudo de caso controlado por trás dos resultados medidos: defeitos de segurança de 11 para 3.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot Code Contributions, a evidência de que 40% da saída do Copilot era vulnerável em cenários relevantes para segurança.
- Agent Skills in the Wild, An Empirical Study of Security Vulnerabilities at Scale, 26,1% de 31.132 agent skills continham ao menos uma vulnerabilidade de segurança.
- MITRE CWE Top 25 Most Dangerous Software Weaknesses, o catálogo de referência para os identificadores CWE que cada princípio constitucional nomeia.
- OWASP Top 10, o contexto de por que injeção segue um risco de primeira ordem em código de produção.
- GitHub Spec Kit, o toolkit cujo comando /constitution cria o arquivo de princípios na camada de spec.
- GitHub Copilot Documentation, a referência para o agent mode e o arquivo .github/copilot-instructions.md que leva a constituição para cada sessão.

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 CAMINHO DE ADOÇÃO

O quadro completo: SDD mais TDD em produção.

Spec-Driven Development, Test-Driven Development, Spec Kit, Specky e Constitutional SDD não são cinco práticas separadas. São uma disciplina de produção. Este capítulo coloca todas em um único pipeline, mostra como escolher o trabalho entre agentes e humanos, enquadra o ecossistema inteiro em uma frase, e dá um caminho concreto de adoção que você pode começar esta semana com o GitHub Copilot em agent mode. O objetivo não é código mais rápido. O objetivo é o código certo, correto e seguro, mais rápido.

Três tipos de projeto, um pipeline

A maioria dos times supõe que trabalho greenfield e trabalho legado exigem ferramentas diferentes. Não exigem. O Specky roda o mesmo pipeline de 10 fases para todo tipo de projeto. Só o ponto de entrada muda. Greenfield começa do zero: o agente chama sdd_init e depois sdd_discover faz um conjunto de perguntas estruturadas sobre escopo, usuários, restrições, integrações, desempenho, segurança e implantação. Brownfield começa de uma base de código que já existe: o agente escaneia o repositório, inicializa o projeto com esse contexto e importa documentos existentes para que a spec reflita o que já está lá. Modernização começa de um sistema legado em migração: o agente escaneia o código, roda uma importação em lote de documentos e ingere transcrições de reuniões para que a intenção que vivia apenas na conversa vire parte da spec.

O valor de um único pipeline é que a disciplina não muda quando o projeto muda. Um desenvolvedor que aprendeu o fluxo specify, plan, tasks, implement em uma feature greenfield aplica o mesmo fluxo a uma migração legada. No GitHub Copilot, o agent mode conduz o pipeline por MCP, e um arquivo .github/copilot-instructions.md no nível do repositório carrega as convenções que o agente lê em cada tarefa. O ponto de partida é contexto. O pipeline é constante.

Escolhendo o trabalho: o que os agentes fazem, o que os humanos guardam

Fluxos agênticos de produção seguem nove princípios de Bandara e colegas. Prefira chamadas diretas de função a tool calls para operações determinísticas como commits e timestamps. Prefira tool calls a MCP quando o determinismo importa. Dê a um agente uma ferramenta e uma responsabilidade em vez de sobrecarregar um único agente com geração, validação e transformação. Guarde os prompts fora do código, em arquivos versionados, para que o comportamento mude sem mudar o código; no GitHub Copilot esse arquivo é o .github/copilot-instructions.md. Distribua o trabalho por mais de um modelo e consolide as saídas; o seletor de modelos do Copilot torna isso prático, permitindo enviar uma tarefa para Claude Opus 4.8, Claude Sonnet 5, Gemini 3.1 Pro ou GPT-5.x. Separe o workflow do servidor MCP, containerize a implantação e mantenha o design simples. As execuções de agente no Copilot são cobradas em GitHub AI Credits, em que um crédito equivale a um centavo, então um workflow mais simples também é mais barato.

A divisão entre agente e humano

Uma pesquisa com 99 desenvolvedores experientes mostra uma linha consistente entre o trabalho que serve para agentes e o que não serve. Agentes vão bem em scaffolding e boilerplate, escrever testes, refatoração geral e documentação; na pesquisa essas tarefas registraram bons índices de votos de adequação sobre inadequação, e scaffolding e documentação não receberam nenhum voto de inadequação. Os humanos guardam o trabalho em que um palpite errado é caro: lógica de negócio e conhecimento de domínio, código crítico de segurança e design de banco de dados ficaram firmemente do lado inadequado. A regra é simples. Delegue o trabalho que é bem definido e barato de verificar. Guarde o trabalho em que julgamento, contexto e consequência dominam.

O ecossistema de 2026 em um só quadro

As peças cabem em uma frase. O Spec Kit é a metodologia open-source: slash commands e templates em Markdown que padronizam como uma spec é escrita. O Specky é o motor de enforcement: um servidor MCP com 53 ferramentas, uma máquina de estados que bloqueia o pulo de fases e validação programática de

EARS. O Constitutional SDD é a camada de segurança: princípios inegociáveis colocados na spec para que o código gerado seja seguro por construção, não por inspeção. O TDD é a camada de qualidade: testes escritos primeiro, como input do agente, para que a cobertura seja embutida, não anexada depois. Cada camada responde a uma pergunta diferente. Juntas, cobrem toda a superfície do trabalho de produção.

Leia as três perguntas em ordem e os papéis param de se sobrepor. Você está construindo a coisa certa? Isso é a spec. Você está construindo corretamente? Isso é o teste. Você está construindo com segurança? Isso é a constituição. Um time pode adotá-las uma de cada vez, e cada uma ganha o seu lugar antes de a próxima ser adicionada.

> DEFINIÇÃO
>
> O SDD garante que você construa a coisa certa. O TDD garante que você a construa corretamente. O Constitutional SDD garante que você a construa com segurança.

Um caminho concreto de adoção

Não migre uma base de código inteira de uma vez. Comece com uma feature isolada, de preferência greenfield, e rode o fluxo completo do Spec Kit de ponta a ponta: specify, plan, tasks, implement. Aprenda EARS em seguida. Os seis padrões EARS são a habilidade de maior retorno neste caminho, porque um requisito escrito em EARS é um requisito contra o qual um agente pode compilar e um teste pode verificar. Depois adote uma constituição: comece com 5 a 10 princípios inegociáveis para o seu domínio, cada um nomeando o comportamento exigido e o CWE que ele endereça, e expanda conforme aprende. Quando o time estiver confortável, adicione o Specky para enforcement programático e as 53 ferramentas; a migração é incremental, não uma reescrita. Ao longo de todo o processo, o GitHub Copilot em agent mode é o runtime, lendo a constituição e a spec pelo .github/copilot-instructions.md e por MCP.

O que medir

Duas métricas dizem se a disciplina está funcionando. A primeira é o tempo até o primeiro build seguro. No estudo de caso bancário por trás do Constitutional SDD, ele caiu de 9 dias para 4, uma redução de 56%. A segunda é a quantidade de defeitos capturados antes da produção. No mesmo estudo, os defeitos de segurança caíram de 11 para 3, uma redução de 73%, as iterações de revisão de segurança caíram de 4 para 1, e a cobertura de documentação de conformidade subiu de 23% para 100%. Acompanhe esses números desde a sua primeira feature. Eles são a evidência de que spec, teste e constituição estão ganhando o seu lugar, não apenas somando processo.

Referências

Fontes primárias para o pipeline compartilhado, a divisão do trabalho entre agentes e humanos, as métricas de adoção e o ecossistema de ferramentas descritos neste capítulo.

- GitHub Spec Kit, a toolkit for Spec-Driven Development, a camada de metodologia open-source: slash commands e templates em Markdown para escrever specs.
- Specky, the MCP engine for Spec-Driven Development, o motor de enforcement: 53 ferramentas, uma máquina de estados e validação programática de EARS.
- Bandara et al., A Practical Guide for Designing, Developing, and Deploying Production-Grade Agentic AI Workflows, a fonte dos nove princípios para fluxos agênticos.
- Huang et al., Professional Software Developers Do Not Vibe, They Control, a pesquisa com 99 desenvolvedores experientes por trás da divisão de trabalho entre agente e humano.
- Marri, Constitutional Spec-Driven Development: Enforcing Security by Construction, o estudo de caso por trás do tempo até o primeiro build seguro e dos números de redução de defeitos.
- GitHub Copilot Documentation, o runtime de referência: agent mode, instruções de repositório e MCP.

- Model Context Protocol Specification, o padrão aberto que o Specky usa para expor suas ferramentas aos agentes.

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