SDD, TDD, SPEC KIT Y SPECKY · PLAYBOOK

Del vibe coding a una disciplina spec-driven que entrega código de producción.

El código que corre no es código que llega a producción. El Spec-Driven Development hace de la spec la fuente de la verdad, EARS y TDD convierten cada requisito en un test que el agente debe pasar, y una constitución mantiene el código generado seguro por construcción. Spec Kit aporta la metodología, Specky la impone, y el agent mode de GitHub Copilot corre ambos, del selector de modelos a los AI Credits.

AUTORA

Paula Silva

CARGO

AI-Native Software Engineer

CONTACTO

in/paulanunes

TIEMPO DE LECTURA

~ 45 minutos, de punta a punta

5
ETAPAS DE MADUREZ

6
CAPÍTULOS

6
PATRONES EARS

4
FASES DEL SDD

CAPÍTULOS

Chapter 00

Del vibe coding a la disciplina de producción

Por qué el código que corre no es código que llega a producción, la evidencia empírica contra la adopción acrítica, la curva de madurez AI-native de cinco etapas, y por qué los desarrolladores con experiencia no hacen vibe.

Problema →

Chapter 01

La spec es la fuente de la verdad

La inversión fundamental del código a la intención, las cuatro fases con dueños y salidas, la anatomía de una spec bien formada, y el SDD como quinto y último estadio de madurez.

Inversión →

Chapter 02

EARS y TDD, el contrato del agente

La notación EARS y sus seis patrones, convertir un requisito vago en tres testeables, y el ciclo Red-Green-Refactor del TDD mapeado al loop del agente.

Contrato →

Chapter 03

Spec Kit y Specky

La capa de metodología con cinco slash commands y plantillas en Markdown, el motor de ejecución con 53 herramientas en un pipeline de 10 fases, y la integración con el agent mode de GitHub Copilot.

Herramientas →

Chapter 04

Constitutional SDD, seguridad por construcción

Los cuatro modos de falla de seguridad de la IA, el CONSTITUTION.md como principios innegociables en la capa de spec, la trazabilidad de cumplimiento como evidencia de auditoría por construcción, y los resultados medidos.

Gobernanza →

Chapter 05

El cuadro completo y la adopción

Tres tipos de proyecto en un pipeline, elegir el trabajo entre agentes y humanos, el ecosistema de 2026 en un solo cuadro, y un camino concreto de adopción con métricas.

Adopción →

CONSEJOS DE LECTURA

ATAJOS DE TECLADO

Presiona `/` para buscar. Usa `j` y `k` para moverte entre capítulos. Presiona `t` para cambiar tema. Presiona `g` para ir al inicio.

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 EL PROBLEMA

Del vibe coding a la disciplina de producción.

En febrero de 2025, Andrej Karpathy le puso nombre a un hábito que los ingenieros venían practicando desde hacía meses. Lo llamó vibe coding: aceptas lo que la IA genera, lo ejecutas y sigues adelante, sin revisión crítica. El código casi siempre corre. Esa es justamente la trampa. Código que corre no es lo mismo que código correcto, seguro y revisable. Este capítulo nombra el patrón, sopesa la evidencia empírica contra adoptarlo sin crítica, lo ubica en una curva de madurez de cinco etapas y muestra qué hacen en su lugar los desarrolladores con experiencia.

Vibe coding, con nombre

Andrej Karpathy acuñó el término vibe coding en febrero de 2025, en una publicación que se difundió rápido porque describía algo real. Te entregas por completo a lo que produce el modelo. Haces el prompt, aceptas, ejecutas, haces otro prompt. GitHub Copilot en agent mode vuelve esto sin fricción: el agente edita archivos, ejecuta comandos y abre un pull request mientras miras. Para un prototipo de fin de semana, ese flujo está bien. El problema empieza cuando el mismo flujo llega a producción. Lo que es inofensivo a escala de prototipo se vuelve un pasivo en el momento en que otras personas dependen del resultado.

La falla es sutil porque el código suele correr. Correr es una señal débil. Una inyección de SQL corre. Sigue corriendo hasta que alguien la explota. Código que funciona no es lo mismo que código correcto, seguro, mantenible y alineado con el requisito que debía cumplir. El vibe coding optimiza la primera propiedad y queda ciego ante las otras cuatro. Cada reintento acrítico en agent mode también consume GitHub AI Credits, facturados a un crédito por centavo de dólar desde el 1 de junio de 2026, así que el hábito no solo es riesgoso, está medido.

> DEFINICIÓN
>
> Código que corre no es código que llega a producción. Correr no prueba nada sobre corrección, seguridad ni revisabilidad. El vibe coding optimiza la única propiedad fácil de observar e ignora las tres que deciden si el código pertenece a producción.

La evidencia empírica contra la adopción acrítica

Cuatro estudios fijan la magnitud del riesgo. Pearce y colegas, en el IEEE Symposium on Security and Privacy de 2022, encontraron que el 40% de la salida de GitHub Copilot era vulnerable en escenarios relevantes para la seguridad. Liu y colegas, en un estudio de 2026 sobre 31.132 agent skills, encontraron que el 26,1% contenía al menos una vulnerabilidad de seguridad. Becker y colegas, en el estudio de METR de 2025, midieron a desarrolladores open source con experiencia trabajando en repositorios grandes y maduros y encontraron que eran 19% más lentos con herramientas de IA que sin ellas, cuando las usaban sin disciplina.

El cuarto estudio mide la línea de llegada. Huang y colegas, en un estudio de 2025 sobre un agente de issue tracker, encontraron que solo el 8% de las invocaciones del agente llegó a un éxito completo, es decir, un pull request fusionado. Leídos en conjunto, estos números describen la distancia entre generar y entregar. El modelo puede producir código plausible en el primer intento. Lograr que ese código se fusione, sea seguro y genere confianza es otro problema, y es el problema que aborda este playbook. Alternar entre los modelos del selector de GitHub Copilot, del Haiku 4.5 al Opus 4.8, cambia velocidad y costo, no la disciplina que cierra esa distancia. La restricción que limita es el proceso alrededor del agente, no el modelo dentro de él.

Cinco etapas de madurez AI-native

El Spec-Driven Development no aparece de la nada. Está al final de una curva de madurez que la mayoría de los equipos sube en orden. Cada etapa agrega la

estructura que le faltaba a la anterior, y cada una absorbe a la anterior en lugar de reemplazarla.

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

La primera etapa es el AI Assistant: autocompletado y sugerencias inline dentro del editor. La segunda es el Vibe Coding: delegación total sin revisión, la etapa contra la que advierte este capítulo. La tercera es el Prompt Engineering: moldear la intención mediante instrucciones escritas con cuidado. La cuarta es el Context Engineering: alimentar al agente con los archivos, estándares e historial correctos, muchas veces a través de un archivo .github/copilot-instructions.md y servidores Model Context Protocol. La quinta y última etapa es el Spec-Driven Development, que agrega una especificación versionada y ejecutable como fuente de verdad. No descarta el prompt engineering ni el context engineering. Los absorbe y pone un contrato revisable encima.

Por qué los profesionales no hacen vibe

El hallazgo más útil del estudio de Huang es cualitativo. Los investigadores entrevistaron a 99 desarrolladores con experiencia sobre cómo usan de verdad los agentes de codificación. El patrón fue consistente: los profesionales no hacen vibe. Planifican antes de delegar y supervisan mientras el agente trabaja. Mantienen al agente dentro de un límite que definieron, leen lo que produce y rechazan lo que no encaja. Planificación más supervisión activa es lo que mantiene a un humano en control del resultado.

Este es el puente práctico del problema al método. La disciplina que los desarrolladores con experiencia aplican por instinto es exactamente lo que el Spec-Driven Development vuelve explícito y repetible. La spec es el plan, escrito antes del código. Las pruebas son la supervisión, ejecutadas de forma continua contra la spec. En términos de GitHub Copilot, la especificación y la constitution viven en el repositorio, el agent mode se ejecuta contra ellas, y Specky impone los gates de fase mediante el Model Context Protocol. El resto de este playbook construye esa disciplina, etapa por etapa.

DEFINICIÓN

Del estudio de Huang: por ahora, las IA todavía no están tomando el control, los desarrolladores con experiencia siguen al mando. La diferencia entre vibe coding y práctica profesional no es el modelo. Es planificar antes de delegar y supervisar durante la ejecución.

Referencias

Fuentes primarias para el patrón del vibe coding y la evidencia empírica de este capítulo.

- Andrej Karpathy, the original vibe coding post, febrero de 2025, donde se acuñó el término.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE Symposium on Security and Privacy 2022: 40% de la salida vulnerable en escenarios relevantes para la seguridad.
- Liu et al., Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, 2026: 26,1% de 31.132 agent skills con al menos una vulnerabilidad.
- Becker et al., Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, METR 2025: desarrolladores con experiencia 19% más lentos con IA en repositorios grandes y maduros.
- Huang et al., Professional Software Developers Don't Vibe, They Control, 2025: 8% de las invocaciones del agente llegó a un pull request fusionado, de un estudio con 99 desarrolladores con experiencia.
- Spec Kit, toolkit for Spec-Driven Development, la capa de metodología open source sobre la que se apoya este playbook.
- GitHub Copilot Documentation, la plataforma de referencia para agent mode, copilot-instructions y el selector 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 LA INVERSIÓN

La spec es la fuente de la verdad: la intención lidera, el código y las pruebas siguen.

El desarrollo tradicional confía en el código y trata todo lo demás como comentario. El Spec-Driven Development invierte esa confianza. La intención se escribe primero, como una especificación versionada, y el código, las pruebas y la documentación pasan a ser artefactos derivados de ella. Este capítulo explica la inversión, recorre las cuatro fases de Specify a Implement, disecciona la anatomía de una spec bien formada y ubica al SDD como el quinto y último estadio de la madurez AI-native, el que mueve la revisión hacia arriba, hacia la spec.

La inversión fundamental: la intención pasa a ser la fuente de la verdad

El desarrollo tradicional trata el código como la fuente de la verdad. La documentación queda obsoleta en semanas. Las pruebas capturan un comportamiento que ya ocurrió. Las decisiones viven en un historial de chat que nadie relee. Cada nuevo prompt para un agente de codificación empieza desde un contexto vacío y reconstruye la intención a partir de cualquier archivo que llegue a abrir. El SDD invierte ese arreglo. La especificación pasa a ser la fuente de la verdad, y el código, las pruebas y la documentación pasan a ser artefactos derivados, cada uno generado a partir de la spec y trazable de vuelta a ella.

La ganancia práctica es un contexto que persiste. Cuando la intención está escrita como un artefacto versionado, un agente de GitHub Copilot no vuelve a inferir los objetivos en cada sesión; los lee. La spec viaja junto con el trabajo como contexto durable, así que el razonamiento que dio forma al Sprint 1 sigue siendo legible en el Sprint 5. La revisión también se desplaza. En lugar de atrapar errores en un pull request después de que el código ya existe, el equipo revisa la spec antes de que se escriba una línea, donde una corrección cuesta minutos en vez de una reescritura.

Ese es el sentido de la inversión: mover las decisiones caras al extremo barato del pipeline.

> DEFINICIÓN
>
> La inversión en una línea: la spec es la fuente de la verdad, y el código, las pruebas y la documentación son artefactos derivados que se cargan como contexto persistente del agente. Revisas la intención antes de la implementación, no después.

Las cuatro fases: dueños y salidas

Todo proyecto SDD fluye por cuatro fases, y cada salida es la entrada de la siguiente. Specify va primero. Un product owner define qué hace el sistema y por qué, y la fase escribe el spec.md, que guarda las user stories, los objetivos y los criterios de éxito. Plan va en segundo. Un arquitecto traduce el qué en cómo, y la fase escribe el plan.md, que cubre la arquitectura, el tech stack con su justificación, los modelos de datos, los contratos de API, las metas de latencia y los detalles de seguridad. Nada en Plan se inventa sin que la spec lo haya pedido.

Tasks va en tercero. Un dev lead descompone el plan en unidades de trabajo atómicas y trazables, y la fase escribe el tasks.md, marcando cada tarea paralelizable con un marcador [P] para que el trabajo independiente pueda correr al mismo tiempo. Implement va en cuarto. Un desarrollador o un agente de GitHub Copilot en agent mode ejecuta las tareas, valida continuamente contra la spec y envía el resultado para revisión. Spec Kit expone cada fase como un slash command, de /specify a /implement, y Specky corre la misma secuencia como un servidor MCP con gates que bloquean el salto de fases. La metodología nombra las fases; la herramienta impone su orden.

Anatomía de una spec bien formada

Una spec gana su autoridad siendo precisa. Abre con un objetivo de una sola frase, la North Star que declara el resultado en una línea. Debajo se ubican las user stories

numeradas en formato US-001, para que cada requisito tenga un identificador estable que las tareas y las pruebas posteriores puedan citar. Los criterios de éxito vienen a continuación, y llevan umbrales en lugar de adjetivos. "Login en menos de 200ms p95" es un criterio que un agente y una prueba pueden verificar. "Login rápido" es una opinión, y un agente al que se le pide satisfacer una opinión inventará su propia definición de rápido.

Luego la spec traza sus fronteras. Las fronteras de alcance declaran lo que el sistema no hace, lo que impide que un agente llene el silencio con funcionalidades que nadie pidió. Las dependencias nombran los servicios externos, las bibliotecas y los sistemas de los que depende el trabajo, para que nada aparezca como sorpresa durante la implementación. Un sexto componente cierra el contrato: requisitos en formato EARS, las afirmaciones testeables contra las que el agente compila. Juntas, estas partes convierten un párrafo de intención en una especificación ejecutable, y cada artefacto derivado remite de vuelta a una línea que el equipo aceptó.

El SDD como el quinto y último estadio de madurez

El desarrollo AI-native maduró en estadios. Primero llegó el AI Assistant, el autocompletado dentro del editor. Después el Vibe Coding, aceptar lo que el modelo produjera sin revisión. Después el Prompt Engineering, moldear la salida mediante instrucciones cuidadosamente redactadas. Después el Context Engineering, alimentar al agente con los archivos, las skills y las instrucciones de repositorio correctas mediante el .github/copilot-instructions.md y servidores MCP. El SDD es el quinto y último estadio, y es el que vuelve responsables a los cuatro anteriores por su resultado.

El SDD no descarta los estadios anteriores. Absorbe el prompt y el context engineering y agrega una capa de spec versionada y ejecutable por encima. Esa capa cambia dónde ocurre la revisión. Cuando la intención es un artefacto durable, la revisión más valiosa se mueve hacia arriba, hacia la spec, antes de que comience la implementación. El estudio de METR es la evidencia de por qué esto importa: los desarrolladores experimentados tardaron 19% más con herramientas de IA en repositorios grandes y maduros, porque la restricción que limitaba era la intención

estructurada, no la capacidad del modelo. Una spec revisada aporta exactamente eso, y lo hace antes de que se facture un solo token de código.

Referencias

Fuentes primarias para la metodología Spec-Driven Development, el flujo de cuatro fases y el argumento empírico para mover la revisión hacia arriba.

- GitHub Spec Kit, a toolkit for Spec-Driven Development, la metodología open-source que define las cuatro fases y los artefactos spec.md, plan.md y tasks.md.
- Specky, the MCP engine for Spec-Driven Development, corre la secuencia de cuatro fases de forma programática con gates que bloquean el salto de fases.
- GitHub Copilot Documentation, la plataforma de referencia para correr la fase Implement en agent mode.
- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, la evidencia empírica: los desarrolladores experimentados tardaron 19% más con herramientas de IA en repositorios grandes y 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 EL CONTRATO

Requisitos contra los que el agente puede compilar: notación EARS y TDD como contrato.

Un requisito que el agente no puede testear es un requisito que no puede honrar. GitHub Copilot en agent mode no falla porque escriba código malo; falla porque rellena los huecos de un requisito vago con suposiciones. Este capítulo transforma la intención en un contrato contra el que el agente puede compilar. La notación EARS quita la ambigüedad del requisito. El Test-Driven Development convierte cada requisito en un test que falla y que el agente debe hacer pasar. Juntos, le dan al agente un objetivo que puede acertar y una señal que le dice cuándo lo acertó.

Notación EARS: seis patrones que quitan la ambigüedad

La mayoría de los requisitos derrota al agente antes de generar un solo token. "La interfaz debe ser intuitiva." "El rendimiento debe ser adecuado." "El sistema debe ser rápido." Ninguno puede testearse, así que ninguno restringe lo que el agente produce. La Easy Approach to Requirements Syntax, EARS, lo corrige forzando cada requisito a uno de seis patrones estructurados. Cada patrón tiene una palabra-disparador y la misma columna vertebral, el sistema DEBE, seguida de un comportamiento medible. La palabra-disparador dice qué condición activa el comportamiento; la cláusula DEBE dice exactamente qué hace el sistema. No queda espacio para la interpretación.

Los seis patrones cubren todo el rango de comportamiento que un sistema exhibe, y la palabra-disparador es lo que hace testeable a cada uno. Los requisitos ubicuos no usan palabra-disparador y declaran un invariante siempre verdadero: el sistema DEBE hashear todas las contraseñas con bcrypt cost 12. Los requisitos orientados a evento usan CUANDO y se disparan ante un evento discreto: CUANDO un usuario

envía el formulario de login, el sistema DEBE responder en 200ms p95. Los requisitos orientados a estado usan MIENTRAS y valen durante un estado sostenido: MIENTRAS está en modo de mantenimiento, el sistema DEBE devolver 503 a todas las solicitudes. Los requisitos opcionales usan DONDE y solo se aplican cuando una feature está presente: DONDE la autenticación de dos factores está habilitada, el sistema DEBE exigir un código de un solo uso después de la contraseña. Los requisitos no deseados usan SI y ENTONCES para nombrar un caso excepcional y su respuesta: SI el login falla cinco veces, ENTONCES el sistema DEBE bloquear la cuenta. Los requisitos complejos combinan un estado y un evento: MIENTRAS procesa un pago, CUANDO ocurre un timeout, el sistema DEBE encolar un reintento.

Por qué la palabra-disparador es el test

La palabra-disparador no es adorno. Nombra la precondición que el test debe montar antes de verificar el comportamiento. CUANDO un usuario envía el formulario de login le dice al test que prepare un formulario enviado; el sistema DEBE responder en 200ms p95 le dice qué verificar. Como el mapeo de patrón a test es mecánico, el requisito es verificable por máquina. Specky valida la estructura EARS programáticamente con schemas Zod, así que un requisito malformado se atrapa en la spec, no se descubre en producción.

> DEFINICIÓN
>
> Todo requisito EARS se lee igual: una palabra-disparador opcional, el sistema DEBE y un comportamiento medible. Si no puedes escribir un test que monte el disparador y verifique el comportamiento, el requisito no está terminado.

De vago a testeable

Considera un requisito que un revisor rechazó: el sistema de login debe ser rápido y seguro. Parece una especificación y no especifica nada. ¿Rápido para quién, medido cómo? ¿Seguro contra qué, verificado cómo? Un agente que recibe este requisito producirá código que parece plausible y no satisface ninguna de las dos propiedades, porque ninguna se declaró en términos que alguien pudiera comprobar. La corrección no es agregar adjetivos. Es descomponer el requisito en

afirmaciones separadas y medibles, cada una en un patrón EARS, cada una con un umbral en vez de un adjetivo.

Rápido y seguro se convierte en tres requisitos. R-001, orientado a evento: CUANDO un usuario envía login, el sistema DEBE responder en 200ms en el p95. R-002, ubicuo: el sistema DEBE hashear contraseñas con bcrypt cost 12. R-003, no deseado: SI el login falla cinco veces, ENTONCES el sistema DEBE bloquear la cuenta. Cada uno es testeable: puedes escribir una aserción que pasa o falla. Cada uno es rastreable: lleva un identificador que lo acompaña de la spec al test, a la línea de código que lo implementa y de vuelta al pull request que lo entregó. Una frase vaga no produjo nada que el agente pudiera apuntar. Tres requisitos EARS producen tres tests, y tres tests son tres objetivos que el agente puede acertar y que un revisor puede verificar.

TDD en la era de la IA

El Test-Driven Development es dos décadas más antiguo que los agentes de IA, y su ciclo mapea al loop del agente casi sin fricción. El ciclo tiene tres pasos. RED: escribe un test que exprese el comportamiento deseado y velo fallar. Si no falla, el test está mal o el comportamiento ya existe. GREEN: escribe el mínimo código necesario para que el test pase, nada más. REFACTOR: mejora el diseño y elimina duplicación mientras los tests siguen en verde, y nunca agregues comportamiento en este paso. La disciplina es que el comportamiento solo entra por un test que falla.

Dónde entra el agente en el ciclo

SDD provee la spec, así que los criterios de aceptación en EARS se traducen directamente en esqueletos de test. El desarrollador revisa los esqueletos y los corre: RED. GitHub Copilot en agent mode genera la implementación contra los tests que fallan, y RED se vuelve GREEN. Esta es la división natural del trabajo. El agente es bueno produciendo código que satisface un objetivo explícito, y un test que falla es el objetivo más explícito que existe. El refactoring permanece liderado por humano, porque decidir qué es buen diseño es un juicio que la spec no codifica, y porque un refactor que cambia el comportamiento es un bug que los tests deberían haber evitado. El humano es dueño del diseño; el agente rellena la implementación; la suite de tests arbitra.

Tests como entrada del agente

Sin TDD, el código generado por el agente no tiene red de seguridad. No hay nada que atrape un refactor que cambia el comportamiento en silencio, así que el refactoring se vuelve arriesgado y raro. Los casos de borde para los que nadie escribió un test aparecen en producción, donde son más caros de corregir. Los tests, si es que se escriben, llegan después del código, lo que significa que codifican lo que el código ya hace en vez de lo que el requisito exigía. El agente es rápido, y sin tests es rápido produciendo código que nadie puede cambiar con seguridad.

Con TDD, el test es la entrada del agente, no una ocurrencia tardía. El test que falla es el loop de feedback: el agente escribe código, corre el test, lee el fallo y corrige, sin un humano en el loop interno en cada intento. La cobertura es nativa desde el inicio porque el test existía antes del código. Los casos de borde se nombran en la spec como requisitos no deseados y se vuelven tests antes de volverse incidentes. La evidencia a favor de la disciplina está registrada: METR constató que los desarrolladores experimentados fueron más lentos con herramientas de IA en repositorios grandes y maduros, y la causa era la falta de estructura, no la debilidad de los modelos. Una suite de tests es esa estructura. Es el contrato contra el que el agente compila, y la señal que le dice a todos cuándo se cumplió el contrato.

> DEFINICIÓN
>
> El test no es una comprobación que corres después de que el agente termina. Es la especificación hacia la que el agente trabaja. Escribe el test primero y el agente tiene un objetivo; escríbelo al final y solo tienes una descripción de lo que el agente terminó produciendo.

Referencias

Fuentes primarias para la notación EARS, el toolchain de SDD y la evidencia empírica a favor de la disciplina orientada a tests con agentes de codificación.

- Spec Kit, toolkit para Spec-Driven Development, la metodología open-source que transforma requisitos EARS y pasos de TDD en el contrato ejecutable que los agentes siguen.

- Specky, el motor MCP de ejecución de SDD, valida la estructura EARS programáticamente y conduce el loop Red-Green-Refactor.
- Documentación de GitHub Copilot, el agent mode, runtime de referencia que genera implementaciones contra tests que fallan.
- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, la evidencia empírica: desarrolladores experimentados fueron 19% más lentos con herramientas de IA en repositorios grandes y maduros, por falta de estructura.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE S&P 2022: el 40% del output de Copilot fue vulnerable en escenarios relevantes de seguridad, el argumento para requisitos de seguridad testeables.

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 LA CADENA DE HERRAMIENTAS

Spec Kit y Specky: la capa de metodología y el motor de ejecución.

Spec-Driven Development necesita dos cosas para funcionar en la práctica: una metodología que los agentes puedan seguir, y un motor que la imponga. GitHub Spec Kit aporta la primera. Specky aporta el segundo. Este capítulo recorre la división del trabajo entre ambos, las diez fases por las que pasan, y cómo la cadena de herramientas se conecta con el agent mode de GitHub Copilot.

Spec Kit: la capa de metodología

GitHub Spec Kit es la capa de metodología de Spec-Driven Development. Se distribuye como un toolkit open-source en github.com/github/spec-kit, y funciona agregando un pequeño conjunto de slash commands a cualquier agente de IA que ya uses. Cinco comandos conducen el flujo principal. /speckit.constitution escribe los principios innegociables, las políticas de seguridad y las restricciones de arquitectura que todo artefacto posterior debe honrar. /speckit.specify convierte la descripción de un problema de negocio en una spec de user stories y requisitos EARS. /speckit.plan traduce esa spec en un plan técnico con un stack elegido y su justificación. /speckit.tasks descompone el plan en tareas atómicas, rastreables, con marcadores de paralelismo y pasos de TDD. /speckit.implement ejecuta esas tareas con un loop de prueba-que-falla-primero. Cada comando lee la salida del anterior, así que el flujo es una cadena, no un conjunto de prompts inconexos.

Spec Kit lleva este flujo como plantillas en Markdown, no como código, y por eso los mismos comandos corren en más de 30 agentes de IA sin cambios. La salida es una estructura de proyecto versionada de cinco archivos. Un constitution.md vive en .specify/, una carpeta de feature numerada en specs/ guarda el spec.md, el plan.md y el tasks.md de esa feature, y el código y las pruebas generadas van a src/ y tests/. Cada archivo es Markdown plano, commiteado en Git y revisable en un pull request normal. La revisión sube al inicio: un revisor puede rechazar una spec antes de que exista una línea de código, lo que es mucho más barato que atrapar el mismo error

después de que el agente ya escribió la implementación. Spec Kit define qué escribir y en qué orden. No impone que alguien de hecho siga el orden.

Specky: el motor de ejecución

Specky es el motor de ejecución que aporta la imposición que Spec Kit deja fuera. Se distribuye en github.com/paulasilvatech/specky como un servidor MCP, así que cualquier agente que hable el Model Context Protocol puede llamarlo. Specky expone 53 herramientas organizadas en un pipeline de 10 fases, y una máquina de estados está por debajo de ellas. La máquina de estados bloquea el salto de fases: no puedes saltar de Init a Design sin completar Specify primero, y no puedes marcar una fase como completa mientras falte su salida obligatoria. Donde Spec Kit confía en que el agente siga los pasos, Specky vuelve los pasos una precondición que el agente tiene que satisfacer. El encuadre que usa el proyecto es que la IA es el operador y Specky es el motor, de modo que el agente corre por un pipeline validado en lugar de producir conjeturas sin estructura.

La validación es programática, no otro prompt. Specky corre un validador EARS de verdad, que revisa cada requisito contra los seis patrones EARS y señala términos vagos como rápido, bueno o intuitivo antes de que lleguen al código. Toda entrada de herramienta se valida con Zod schemas, así que una spec malformada falla en el schema, no tres fases después en un build roto. Esa es la diferencia entre un agente que trata de seguir instrucciones y un motor que las verifica: la máquina de estados, el regex de EARS y los Zod schemas son código que pasa o no pasa. El resultado es un pipeline que puedes auditar. Cada fase produce un artefacto nombrado, cada gate tiene una decisión registrada, y nada avanza por la sola fuerza de que el modelo suene seguro.

Las diez fases, los gates y las seis puertas de entrada

Todo proyecto Specky corre las mismas diez fases: Init, Discover, Specify, Clarify, Design, Tasks, Analyze, Implement, Verify y Release. Init crea la estructura y la constitución y escanea la base de código. Discover hace siete preguntas estructuradas sobre alcance, usuarios y restricciones. Specify escribe los requisitos

EARS con criterios de aceptación. Clarify resuelve ambigüedades. Design produce la arquitectura, el modelo de datos y los contratos de API. Tasks descompone el trabajo por user story. Analyze corre el análisis de consistencia entre artefactos y las verificaciones de compliance. Implement ejecuta las tareas en orden, con checkpoints. Verify busca drift entre la spec y el código. Release genera el pull request, exporta los work items y arma la documentación. El pipeline es el mismo para trabajo greenfield, brownfield y de modernización; solo cambia el punto de partida.

La supervisión humana está incorporada al pipeline como LGTM gates. Después de Specify, después de Design y después de Tasks, el agente se detiene y te pide revisar el artefacto que acaba de escribir; respondes LGTM para continuar o describes los cambios que quieres. Verify es el cuarto checkpoint: si detecta drift entre la spec y la implementación, Specky te devuelve a Specify para corregir la divergencia antes de que algo llegue a producción. Los gates son donde una persona mantiene el control del trabajo. Revisar una especificación toma minutos, y es el punto del pipeline en el que una decisión equivocada es la más barata de revertir.

Seis formas de iniciar un pipeline

Un pipeline puede comenzar desde cualquiera de seis entradas. El lenguaje natural es la más simple: escribes la idea y el agente redacta la spec. Una transcripción de reunión puede importarse y convertirse en requisitos. Un documento en PDF o DOCX se convierte a Markdown y se alimenta al pipeline. Un diseño de Figma se convierte en una spec de requisitos a través del Figma MCP. Un scan de la base de código inicializa un proyecto brownfield con contexto sobre el código que ya existe. Sin importar por dónde empieces, las diez fases que siguen son idénticas, así que la disciplina no depende de dónde vinieron los requisitos.

Conectando la cadena de herramientas a GitHub Copilot

Dentro de GitHub Copilot, la cadena de herramientas corre por el agent mode. El agent mode es el modo en el que Copilot planifica una tarea, llama herramientas y edita archivos en todo el repositorio en lugar de completar una sola línea, y es el que

conduce a Specky por el pipeline. Registras el motor commiteando un .vscode/mcp.json en la raíz del repositorio que apunta el servidor specky a npx specky-sdd. Como el archivo está en Git, todo el equipo recibe el mismo servidor MCP de forma automática, sin configuración por desarrollador. La orientación a nivel de repositorio va en .github/copilot-instructions.md, así que la constitución y las convenciones que el agente debe respetar viajan junto con el código.

La elección de modelo ocurre en el picker de Copilot, y las fases de Specky te dan un lugar natural donde gastar en ello. El picker lista Claude Fable 5, Opus 4.8, Sonnet 5 y Haiku 4.5, entre otros, y seleccionas por fase. Las fases que cargan el razonamiento, Specify y Design, valen un modelo más fuerte como Opus 4.8 o Claude Fable 5; las fases mecánicas, como Tasks o Release, corren bien en un modelo más rápido como Haiku 4.5. El uso se factura en GitHub AI Credits, donde un crédito es un centavo de dólar, así que el costo de una corrida queda visible y atribuible a la fase que lo generó. Emparejar el modelo con la fase es como mantienes el pipeline a la vez riguroso y asequible.

> DEFINICIÓN
>
> La división del trabajo es simple: Spec Kit define la metodología, Specky la impone, y el agent mode de GitHub Copilot corre ambos. Spec Kit te da los cinco comandos y las plantillas en Markdown. Specky te da las 53 herramientas, la máquina de estados y la validación programática de EARS y Zod que vuelven el pipeline auditable. Copilot aporta el agente, los modelos del picker y la facturación en AI Credits que hacen que todo corra el día a día.

Referencias

Fuentes primarias para la metodología de Spec Kit, el motor Specky, el Model Context Protocol que habla y la integración con GitHub Copilot.

- GitHub Spec Kit, toolkit para Spec-Driven Development, la capa de metodología open-source: cinco slash commands y las plantillas en Markdown estandarizadas en más de 30 agentes de IA.

- Specky, el motor MCP para Spec-Driven Development, el motor de ejecución: un servidor MCP con 53 herramientas, una máquina de estados de 10 fases y validación programática de EARS y Zod.
- Model Context Protocol Specification, el estándar abierto que Specky habla para que cualquier agente compatible llame a sus herramientas.
- GitHub Copilot Documentation, la plataforma de referencia para agent mode, la configuración del .vscode/mcp.json, los modelos del picker y la facturación en 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 CAPA DE GOBERNANZA

Seguridad por construcción, no por inspección.

El código de IA que corre no es lo mismo que el código de IA que es seguro. Una inyección de SQL corre hasta que alguien la explota. La mayoría de los equipos todavía trata la seguridad como un paso de revisión que ocurre después de que el agente escribe el código, lo que significa que cada defecto se detecta tarde o no se detecta. Este capítulo lleva la seguridad a la capa de spec, donde una constitución restringe lo que el agente puede generar antes de que exista la primera línea.

Cuatro modos de falla recurrentes en código generado por IA

Los agentes de codificación con IA fallan en seguridad de formas predecibles. Cuatro modos se repiten entre bases de código. El primero son los vectores de inyección: concatenación de strings en SQL, HTML sin escapar y template literals construidos a partir de entrada del usuario. El código compila y corre. También le entrega a un atacante un camino hacia dentro de la base de datos. El segundo son los valores criptográficos débiles por defecto: MD5, SHA-1, modo ECB y claves fijas en el código. El agente recurre a lo que es común en sus datos de entrenamiento, y común no es lo mismo que seguro. Una demo que pasa esconde la falla, porque un exploit necesita un atacante, no una prueba de camino feliz.

El tercer modo son las brechas de autorización: verificaciones de acceso ausentes, referencias directas inseguras a objetos y caminos de escalamiento de privilegios. El agente implementa el camino feliz y se salta la verificación de que quien llama es dueño del registro que está editando. El cuarto modo es el manejo silencioso de errores: un catch-all que devuelve un 500 genérico, se traga el stack trace y no escribe nada en un rastro de auditoría. Cada modo produce código que funciona en una demo. Cada modo entrega una vulnerabilidad. Ninguno de los cuatro es una falla

de razonamiento del modelo. Son brechas en las restricciones que el agente recibió. Pearce y colegas encontraron que el 40% de la salida de GitHub Copilot era vulnerable en escenarios relevantes para la seguridad. Un estudio de 2026 sobre 31.132 agent skills encontró que el 26,1% contenía al menos una vulnerabilidad de seguridad. Los modos de falla están medidos, no son hipotéticos.

CONSTITUTION.md: principios innegociables en la capa de spec

La corrección es restringir la generación, no inspeccionar la salida. El Spec-Driven Development coloca la seguridad en un archivo CONSTITUTION.md que vive en la capa de spec, por encima de spec, plan y tasks. El Spec Kit lo crea con el comando /constitution. La constitución guarda principios innegociables que gobiernan cada artefacto que el proyecto produce, y la generación de código por IA deja de operar en un espacio sin restricciones. La intuición es simple. Así como la ley constitucional restringe lo que un gobierno puede hacer, una constitución de software restringe lo que el agente puede generar.

Cada principio sigue una estructura fija. Nombra el comportamiento requerido, por ejemplo que las consultas a la base de datos deben usar sentencias parametrizadas. Nombra la referencia CWE que el comportamiento aborda, por ejemplo CWE-89 para inyección de SQL. Nombra el camino donde vive el rastro, el archivo y las líneas que satisfacen el principio. Un principio lleva un nivel de imposición, MUST, SHOULD o MAY, y un patrón de implementación que muestra la forma aceptada y la rechazada. La constitución está versionada. Un cambio en un principio incrementa su versión, y cada pull request verifica el cumplimiento contra ella.

Llevar la constitución al agent mode

Una constitución que vive solo en la capa de spec se detiene en la frontera de la sesión del agente. GitHub Copilot en agent mode lee el .github/copilot-instructions.md en cada tarea. Los principios de la constitución pertenecen ahí también, reescritos como instrucciones que el agente aplica antes de escribir código. Las referencias CWE y los patrones requeridos pasan a formar parte del contexto que el agente carga en cada sesión, no una lista de verificación que un revisor aplica después. Los mismos principios llegan a cualquier modelo en el

selector de Copilot, sea Claude Fable 5, Opus 4.8, Sonnet 5, Gemini 3.1 Pro o un modelo GPT-5.x.

Trazabilidad de cumplimiento: evidencia de auditoría por construcción

Todo artefacto downstream debe honrar la constitución. La spec referencia los principios de los que depende. El plan nombra las técnicas que los satisfacen. Las tasks los llevan a los pasos de implementación. El código mapea de vuelta a ellos mediante una matriz de trazabilidad de cumplimiento: principio, CWE, archivo, líneas y técnica, en una sola tabla. El mapeo es bidireccional. A partir de un principio encuentras su implementación. A partir de un archivo encuentras los principios que debe sostener.

Esa matriz hace cuatro cosas. Apoya la auditoría, porque un auditor puede confirmar que cada requisito de seguridad tiene una implementación correspondiente. Apoya el análisis de impacto de cambios, porque un desarrollador puede ver qué principios toca la edición de un archivo. Revela brechas, porque un mapeo ausente es un requisito no implementado. Y habilita pruebas de regresión dirigidas cuando el código cambia. Un revisor deja de cazar lo que el agente pudo haber olvidado y pasa a confirmar un mapeo que ya existe. La evidencia existe porque el proceso la produjo, no porque alguien la reconstruyó en la semana previa a una auditoría.

> DEFINICIÓN
>
> Seguridad por construcción significa que la restricción precede al código. Seguridad por inspección significa que la revisión viene después de él. La primera produce evidencia de auditoría como subproducto de construir. La segunda la produce como una carrera contra un plazo.

Resultados medidos

Marri 2026 reporta un estudio de caso controlado que compara un flujo constitucional con uno sin restricciones sobre la misma aplicación bancaria. Los

defectos de seguridad detectados bajaron de 11 a 3, una reducción del 73%. El tiempo hasta el primer build seguro bajó de 9 días a 4 días, 56% más rápido. La cobertura de documentación de cumplimiento subió de 23% a 100%. Las iteraciones de revisión de seguridad bajaron de 4 a 1. El flujo también produjo más código de seguridad, 612 líneas contra 847, un tratamiento 38% más completo de los mismos requisitos.

Los números apuntan en una dirección. Llevar la seguridad a la capa de spec no frena la entrega para comprar seguridad. Produce ambas, porque el agente genera dentro de las restricciones en lugar de contra ellas, y la revisión confirma el cumplimiento en lugar de descubrir violaciones. La falsa elección entre avanzar rápido con IA y avanzar con cuidado con seguridad se disuelve cuando las restricciones quedan aguas arriba del código. Construye la cosa correcta con la spec. Constrúyela correctamente con pruebas. Constrúyela de forma segura con la constitución.

Referencias

Fuentes primarias para los modos de falla, el método constitucional y los resultados medidos.

- Marri, Constitutional Spec-Driven Development: Enforcing Security by Construction in AI-Assisted Code Generation, el estudio de caso controlado detrás de los resultados medidos: defectos de seguridad de 11 a 3.
- Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot Code Contributions, la evidencia de que el 40% de la salida de Copilot era vulnerable en escenarios relevantes para la seguridad.
- Agent Skills in the Wild, An Empirical Study of Security Vulnerabilities at Scale, el 26,1% de 31.132 agent skills contenía al menos una vulnerabilidad de seguridad.
- MITRE CWE Top 25 Most Dangerous Software Weaknesses, el catálogo de referencia para los identificadores CWE que cada principio constitucional nombra.
- OWASP Top 10, el contexto de por qué la inyección sigue siendo un riesgo de primer orden en código de producción.

- GitHub Spec Kit, el toolkit cuyo comando /constitution crea el archivo de principios en la capa de spec.
- GitHub Copilot Documentation, la referencia para el agent mode y el archivo .github/copilot-instructions.md que lleva la constitución a cada sesión.

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 CAMINO DE ADOPCIÓN

El cuadro completo: SDD más TDD en producción.

Spec-Driven Development, Test-Driven Development, Spec Kit, Specky y Constitutional SDD no son cinco prácticas separadas. Son una disciplina de producción. Este capítulo las coloca en un único pipeline, muestra cómo elegir el trabajo entre agentes y humanos, enmarca todo el ecosistema en una frase, y da un camino concreto de adopción que puedes empezar esta semana con GitHub Copilot en agent mode. El objetivo no es código más rápido. El objetivo es el código correcto, bien hecho y seguro, más rápido.

Tres tipos de proyecto, un pipeline

La mayoría de los equipos supone que el trabajo greenfield y el trabajo legado necesitan herramientas distintas. No las necesitan. Specky ejecuta el mismo pipeline de 10 fases para todo tipo de proyecto. Solo cambia el punto de entrada. Greenfield empieza desde cero: el agente llama a sdd_init y luego sdd_discover hace un conjunto de preguntas estructuradas sobre alcance, usuarios, restricciones, integraciones, rendimiento, seguridad y despliegue. Brownfield empieza desde una base de código que ya existe: el agente escanea el repositorio, inicializa el proyecto con ese contexto e importa documentos existentes para que la spec refleje lo que ya está ahí. Modernización empieza desde un sistema legado en migración: el agente escanea el código, ejecuta una importación por lotes de documentos e ingiere transcripciones de reuniones para que la intención que vivía solo en la conversación pase a formar parte de la spec.

El valor de un único pipeline es que la disciplina no cambia cuando cambia el proyecto. Un desarrollador que aprendió el flujo specify, plan, tasks, implement en una feature greenfield aplica el mismo flujo a una migración legada. En GitHub Copilot, el agent mode conduce el pipeline mediante MCP, y un archivo .github/copilot-instructions.md a nivel de repositorio lleva las convenciones que el agente lee en cada tarea. El punto de partida es el contexto. El pipeline es constante.

Elegir el trabajo: lo que hacen los agentes, lo que guardan los humanos

Los flujos agénticos de producción siguen nueve principios de Bandara y colegas. Prefiere llamadas directas a funciones antes que tool calls para operaciones determinísticas como commits y timestamps. Prefiere tool calls antes que MCP cuando el determinismo importa. Da a un agente una herramienta y una responsabilidad en lugar de sobrecargar a un solo agente con generación, validación y transformación. Guarda los prompts fuera del código, en archivos versionados, para que el comportamiento cambie sin cambiar el código; en GitHub Copilot ese archivo es .github/copilot-instructions.md. Reparte el trabajo entre más de un modelo y consolida las salidas; el selector de modelos de Copilot lo hace práctico, permitiendo enviar una tarea a Claude Opus 4.8, Claude Sonnet 5, Gemini 3.1 Pro o GPT-5.x. Separa el workflow del servidor MCP, containeriza el despliegue y mantén el diseño simple. Las ejecuciones de agente en Copilot se facturan en GitHub AI Credits, donde un crédito equivale a un centavo, así que un workflow más simple también es más barato.

La división entre agente y humano

Una encuesta a 99 desarrolladores experimentados muestra una línea consistente entre el trabajo que sirve para agentes y el que no. Los agentes rinden bien en scaffolding y boilerplate, escribir pruebas, refactorización general y documentación; en la encuesta esas tareas registraron buenos índices de votos de adecuación sobre inadecuación, y scaffolding y documentación no recibieron ningún voto de inadecuación. Los humanos guardan el trabajo donde una suposición equivocada sale cara: lógica de negocio y conocimiento de dominio, código crítico de seguridad y diseño de base de datos quedaron firmemente del lado inadecuado. La regla es simple. Delega el trabajo que está bien definido y es barato de verificar. Guarda el trabajo donde dominan el juicio, el contexto y la consecuencia.

El ecosistema de 2026 en un solo cuadro

Las piezas caben en una frase. Spec Kit es la metodología open-source: slash commands y plantillas en Markdown que estandarizan cómo se escribe una spec. Specky es el motor de enforcement: un servidor MCP con 53 herramientas, una

máquina de estados que bloquea el salto de fases y validación programática de EARS. Constitutional SDD es la capa de seguridad: principios innegociables colocados en la spec para que el código generado sea seguro por construcción, no por inspección. TDD es la capa de calidad: pruebas escritas primero, como input del agente, para que la cobertura quede incorporada y no añadida después. Cada capa responde a una pregunta distinta. Juntas cubren toda la superficie del trabajo de producción.

Lee las tres preguntas en orden y los papeles dejan de solaparse. ¿Estás construyendo lo correcto? Eso es la spec. ¿Lo estás construyendo bien? Eso es la prueba. ¿Lo estás construyendo de forma segura? Eso es la constitución. Un equipo puede adoptarlas una por una, y cada una se gana su lugar antes de que se añada la siguiente.

> DEFINICIÓN
>
> SDD garantiza que construyes lo correcto. TDD garantiza que lo construyes bien. Constitutional SDD garantiza que lo construyes de forma segura.

Un camino concreto de adopción

No migres una base de código entera de una sola vez. Empieza con una feature aislada, preferiblemente greenfield, y ejecuta el flujo completo de Spec Kit de principio a fin: specify, plan, tasks, implement. Aprende EARS a continuación. Los seis patrones EARS son la habilidad de mayor retorno en este camino, porque un requisito escrito en EARS es uno contra el que un agente puede compilar y una prueba puede verificar. Después adopta una constitución: empieza con 5 a 10 principios innegociables para tu dominio, cada uno nombrando el comportamiento exigido y el CWE que aborda, y amplía a medida que aprendes. Cuando el equipo esté cómodo, añade Specky para enforcement programático y las 53 herramientas; la migración es incremental, no una reescritura. A lo largo de todo el proceso, GitHub Copilot en agent mode es el runtime, leyendo la constitución y la spec a través de .github/copilot-instructions.md y de MCP.

Qué medir

Dos métricas dicen si la disciplina está funcionando. La primera es el tiempo hasta el primer build seguro. En el estudio de caso bancario detrás de Constitutional SDD, cayó de 9 días a 4, una reducción del 56%. La segunda es la cantidad de defectos detectados antes de producción. En el mismo estudio, los defectos de seguridad bajaron de 11 a 3, una reducción del 73%, las iteraciones de revisión de seguridad cayeron de 4 a 1, y la cobertura de documentación de cumplimiento subió del 23% al 100%. Sigue esos números desde tu primera feature. Son la evidencia de que spec, prueba y constitución se están ganando su lugar, no solo sumando proceso.

Referencias

Fuentes primarias para el pipeline compartido, la división del trabajo entre agentes y humanos, las métricas de adopción y el ecosistema de herramientas descritos en este capítulo.

- GitHub Spec Kit, a toolkit for Spec-Driven Development, la capa de metodología open-source: slash commands y plantillas en Markdown para escribir specs.
- Specky, the MCP engine for Spec-Driven Development, el motor de enforcement: 53 herramientas, una máquina de estados y validación programática de EARS.
- Bandara et al., A Practical Guide for Designing, Developing, and Deploying Production-Grade Agentic AI Workflows, la fuente de los nueve principios para flujos agénticos.
- Huang et al., Professional Software Developers Do Not Vibe, They Control, la encuesta a 99 desarrolladores experimentados detrás de la división de trabajo entre agente y humano.
- Marri, Constitutional Spec-Driven Development: Enforcing Security by Construction, el estudio de caso detrás del tiempo hasta el primer build seguro y de los números de reducción de defectos.
- GitHub Copilot Documentation, el runtime de referencia: agent mode, instrucciones de repositorio y MCP.
- Model Context Protocol Specification, el estándar abierto que Specky usa para exponer sus herramientas a los 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