CONFIGURATION CONTEXT ARCHITECTURE · PLAYBOOK

Un archivo cubre 10%. Diez capas en tres tiers hacen a Copilot auditable.

La mayoría de los equipos habilita GitHub Copilot en agent mode, escribe un solo .github/copilot-instructions.md y se detiene. Ese archivo cubre alrededor del 10% de la superficie de configuración que VS Code expone para Copilot. Este playbook mapea el otro 90%: diez capas de configuración agrupadas en tres tiers de memoria, hot, warm y cold, para que cada regla cargue cuando es relevante, nada se trunque, y el comportamiento de Copilot se vuelva auditable, versionado y con dueño, en lugar de tribal.

AUTORA

Paula Silva

CARGO

AI-Native Software Engineer

CONTACTO

in/paulanunes

TIEMPO DE LECTURA

~ 45 minutos, de punta a punta

10	3	6
CAPAS DE CONFIGURACIÓN	TIERS DE MEMORIA	CAPÍTULOS

10%

SUPERFICIE QUE UN ARCHIVO CUBRE

CAPÍTULOS

Chapter 00

Diez capas, no un archivo

La trampa del 10%, el estudio empírico de 2.303 archivos y el punto ciego de seguridad del 14,5%, y por qué los manifiestos de archivo único colapsan más allá de 3.000 tokens en una base de 108.000 líneas y 283 sesiones.

Problema →

Chapter 01

El mapa: tres tiers en diez superficies

Las diez capas de configuración en el orden en que VS Code las carga, el modelo de memoria hot, warm y cold de Vasilopoulos 2026, y la regla única para decidir dónde pertenece cada regla propia.

Mapa →

Chapter 02

La capa de workspace: qué es siempre verdad

El copilot-instructions.md mantenido por debajo de 1.000 tokens, las secciones de Seguridad y Performance que la mayoría de los equipos omite, y el glob applyTo que mantiene corto el archivo hot en cuatro patrones de scope.

Workspace →

Chapter 03

La capa de autoría: lo que invocas

Agents, chat modes, skills y prompt files, elegidos por cómo se activa cada uno y no por lo que contiene, con scope de herramientas de menor privilegio y selección de modelo por agent.

Autoría →

Chapter 04

Gobernanza de runtime: qué corre alrededor del agente

Hooks como guardrails determinísticos, conexiones MCP y plugins para datos frescos y bundles versionados, precedencia de settings, y el orden de ensamblado de nueve pasos que dice por qué se ignoró una regla.

Runtime →

Chapter 05

De cero a baseline en una semana

Un commit por día, del copilot-instructions.md a un plugin compartido, expectativas honestas sobre qué hará y qué no hará la jerarquía, y gobernanza en lugar de prompt engineering.

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

Diez capas, no un archivo.

La mayoría de los equipos habilita GitHub Copilot en agent mode, escribe un solo .github/copilot-instructions.md y considera que la configuración está lista. Ese único archivo cubre alrededor del 10% de la superficie de configuración que VS Code expone para Copilot. El otro 90% queda vacío. En un repositorio pequeño la brecha es invisible. En un parque enterprise de miles de repositorios se convierte en la razón por la que el comportamiento de Copilot permanece tribal en lugar de auditable. Este capítulo nombra la falla, la apoya en dos estudios empíricos y muestra por qué un solo archivo no sostiene una base de código enterprise. La corrección no es un archivo más grande. Es una superficie en capas.

La trampa del 10%

El patrón se repite entre los equipos. Alguien habilita GitHub Copilot en agent mode, escribe un solo .github/copilot-instructions.md y se detiene. El archivo lista el stack, un puñado de convenciones, quizá el comando de build. Se lee bien en una demo. En la práctica cubre alrededor del 10% de la superficie de configuración que VS Code expone para Copilot. El 90% restante queda sin configurar, y la superficie sin configurar es donde el comportamiento drifta. Un repositorio pequeño esconde el costo. Un monorepo de decenas de paquetes, replicado en cientos de repositorios, no.

El 90% ausente no es una sola cosa. Son nueve superficies más: instrucciones con scope usando globs applyTo, agents, chat modes, skills, prompt files, hooks, conexiones MCP, plugins y settings de VS Code. Cada una responde una pregunta que el archivo único no responde. Qué reglas aplican solo a la carpeta de pagos. Qué workflow corre el equipo en cada release. Qué comando Copilot nunca puede ejecutar contra producción. Cuando esas preguntas no tienen respuesta escrita, el modelo llena el vacío con una suposición, y la suposición permanece invisible hasta que un revisor la atrapa en un pull request.

El gap empírico: 2.303 archivos de contexto y el punto ciego del 14,5%

La brecha es medible. Un estudio de 2025 analizó 2.303 archivos de contexto reales en 1.925 repositorios (Chatlatanagulchai et al., arXiv 2511.12884). El patrón es consistente. Los equipos escriben lo que hace al agente funcional. Los comandos de build y ejecución aparecen en 62,3% de los archivos. Los detalles de implementación aparecen en 69,9%. La arquitectura aparece en 67,7%. Son los hechos que ponen a un agente a producir código en el primer intento, así que se escriben primero.

Lo que los equipos omiten es el conjunto de reglas que mantiene ese código seguro. Los guardrails de seguridad aparecen en apenas 14,5% de los archivos. Las reglas de performance aparecen en otro 14,5%. El manejo de errores y el comportamiento degradado son raros. El resultado es un agente que escribe código productivo, pero no predecible. El problema no es el modelo. Copilot permite elegir Claude Fable 5, Opus 4.8 o GPT-5.x desde el mismo model picker, y ninguno de ellos puede aplicar una regla que nunca se escribió. El punto ciego de seguridad del 14,5% es un gap de configuración, no un gap de modelo. Cambiar de modelo no cierra ese gap. Escribir las reglas que faltan, sí.

Por qué colapsan los manifiestos de archivo único

El argumento de escala tiene un estudio de caso detrás. Un proyecto documentado construyó 108.000 líneas de C# durante 70 días y 283 sesiones de Copilot (Vasilopoulos, arXiv 2602.20478). Concluyó que los manifiestos de archivo único colapsan pronto. Un .github/copilot-instructions.md plano que supera los 3.000 tokens es truncado por el modelo. Todo lo que queda después del corte se descarta sin aviso. El equipo sigue agregando reglas. El agente sigue leyendo menos de ellas. Las instrucciones se ven completas en el editor y ya están incompletas cuando el modelo las ve.

La corrección en ese proyecto fue una estructura en tiers: hot memory cargada en cada solicitud, warm memory cargada por trigger, cold memory consultada vía MCP. Esa estructura mantuvo una razón conocimiento-código de 24,2% en un monorepo

de 14 paquetes durante las 283 sesiones. Organizar en tiers también protege la cuenta. Copilot en agent mode mide el uso en GitHub AI Credits, con un crédito equivalente a un centavo de dólar. Una instrucción truncada aún consume tokens de entrada en el camino hacia el modelo. Pagar por enviar contexto que el modelo después descarta es el peor de los dos escenarios: te cobran por la regla y el agente nunca la aplica. Organizar en tiers convierte ese desperdicio en reuso: contexto computado una vez, cargado solo donde se justifica.

> DEFINICIÓN
>
> Un archivo es una herramienta de prototipo. Una base de código enterprise necesita diez capas de configuración en tres tiers, hot, warm y cold, para que cada regla cargue cuando es relevante y ninguna regla se trunque por falta de espacio.

Tres síntomas de una superficie subutilizada

No necesitas un estudio para detectar el problema. Tres síntomas aparecen en el trabajo diario. Instrucciones duplicadas: la misma regla vive en tres archivos, y Copilot sigue la que por casualidad coincide con el archivo abierto. Drift silencioso: el modelo genera código útil que ignora una convención del equipo, porque la convención existe solo en la cabeza de alguien y nunca en un archivo que el modelo pueda leer. Conocimiento tribal: un prompt sólido para construir un endpoint de API funciona, pero solo una persona del equipo sabe que existe.

Cada síntoma apunta a una capa específica. Las instrucciones duplicadas indican que la regla pertenece a un archivo con scope en .github/instructions con un glob applyTo, no copiada al manifiesto raíz. Las instrucciones con scope se mergean con el archivo de workspace, no lo reemplazan, así la raíz permanece corta mientras la regla carga solo donde aplica. El drift silencioso indica que una convención nunca se codificó en el tier hot. El conocimiento tribal indica que un workflow debería ser un skill o un prompt file, direccionable por todo el equipo. La corrección es estructural, no conversacional. Mapea cada regla a la capa que la gobierna. El objetivo no es más configuración. El objetivo es configuración que sea auditable, versionada e inequívoca sobre quién es responsable de cada capa.

Referencias

Fuentes primarias para la superficie de diez capas, el gap empírico y el estudio de caso detrás de la falla del archivo único.

- Agent READMEs: An Empirical Study of Context Files for Agentic Coding, el gap empírico: 2.303 archivos de contexto, con guardrails de seguridad presentes en apenas 14,5% de ellos.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, el estudio de caso de 108.000 líneas y 283 sesiones sobre por qué colapsan los manifiestos de archivo único.
- Customize AI in Visual Studio Code, la referencia para las diez superficies de configuración mapeadas en este capítulo.
- GitHub Copilot Documentation, la plataforma detrás del agent mode, del copilot-instructions.md y del model picker.
- Model Context Protocol, el estándar abierto que lleva el tier cold de contexto hasta el agente.

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 EL MAPA

Tres tiers en diez superficies: el mapa de cada capa de configuración de GitHub Copilot.

Cada respuesta de GitHub Copilot en VS Code se ensambla a partir de diez superficies de configuración, no de un solo archivo de instrucciones. Esas diez superficies se agrupan en tres tiers de memoria, y los tres tiers responden a tres preguntas sobre la base de código. Este capítulo es el mapa. Nombra las diez capas en el orden en que Copilot las carga, las agrupa en los tiers hot, warm y cold documentados por Vasilopoulos 2026, y entrega la regla única que decide dónde pertenece cada regla propia.

Las diez capas de configuración, en el orden en que Copilot las carga

Cada respuesta de GitHub Copilot se construye a partir de una ventana de contexto en capas, y hay diez superficies que la alimentan. El archivo de instrucciones del workspace en .github/copilot-instructions.md carga siempre. Las instrucciones con scope en .github/instructions/*.instructions.md cargan por match de ruta contra un glob applyTo. Los agents cargan cuando se invocan por nombre, los chat modes cuando se seleccionan, los skills por match de descripción y los prompt files por slash command. Los hooks disparan en eventos de ciclo de vida. Las conexiones MCP se adjuntan por sesión. Los plugins cargan cuando están instalados. Los settings de VS Code aplican siempre. La mayoría de los equipos escribe uno de estos, el copilot-instructions.md, y ahí se detiene, lo que cubre alrededor del diez por ciento de la superficie. El noventa por ciento restante queda sin configurar, y por eso el comportamiento de Copilot permanece tribal en lugar de auditable.

El orden no es cosmético, porque Copilot ensambla la ventana de contexto de forma determinística. VS Code resuelve los settings primero, default luego usuario luego workspace, con el scope más específico ganando. Carga el copilot-instructions.md

incondicionalmente. Evalúa los globs applyTo contra los archivos abiertos y hace merge de cada conjunto de instrucciones que coincide. Adjunta el primitivo de autoría activo, sea un agent, un chat mode, un skill o un prompt file. Ejecuta los hooks on-prompt-submit, que pueden denegar, mutar o anotar la solicitud. Descubre las herramientas expuestas por los servidores MCP conectados. Luego despacha el system prompt ensamblado, el catálogo de herramientas y el mensaje del usuario al modelo seleccionado en el picker de Copilot. Las capas posteriores refinan las anteriores; no pueden contradecirlas. Un skill no ignora una política de organización, y un prompt no sobrescribe un setting de administrador enterprise.

Depura de arriba hacia abajo

Ese orden también es una herramienta de depuración. Cuando Copilot ignora una regla, la falla está casi siempre en uno de tres lugares: el scope de settings incorrecto, un glob applyTo que no coincide con el archivo que tienes abierto, o un hook que bloqueó o mutó la instrucción antes de que llegara al modelo. Recorre el orden de ensamblado de arriba hacia abajo antes de concluir que el modelo tiene un problema. La secuencia es reproducible, así que el bug es localizable.

Hot, warm, cold: el modelo de memoria en tres tiers de Vasilopoulos 2026

Las diez superficies se agrupan en tres tiers de memoria, una taxonomía documentada por Vasilopoulos (2026, arXiv 2602.20478). Ese estudio de caso construyó e instrumentó un sistema en C# de 108.000 líneas durante 70 días y 283 sesiones en un monorepo de 14 paquetes, y mantuvo una razón conocimiento-código de 24,2 por ciento. El hallazgo que importa aquí es que un manifiesto de archivo único colapsa mucho antes de esa escala. Un copilot-instructions.md plano que supera los 3.000 tokens es truncado por el modelo, y todo lo que queda después del corte drifta en silencio. La corrección no es un archivo más largo. La corrección son tiers.

La memoria hot son las capas 01 y 02 más los settings de VS Code relacionados con Copilot: siempre cargadas, en cada solicitud. La memoria warm son las capas 03 a 07, las instrucciones con scope, agents, chat modes, skills, prompt files y hooks, cargadas bajo demanda cuando un disparador se activa por nombre, selección,

glob, match de descripción, slash command o evento de ciclo de vida. La memoria cold son las capas 08 y 09, las conexiones MCP y plugins, consultadas explícitamente cuando el agente busca datos que viven fuera del repositorio. La memoria hot viaja en cada turno, así que cada token en ella se paga en cada solicitud. Desde el 1 de junio de 2026, GitHub Copilot cobra el uso premium en GitHub AI Credits, un credit equivalente a un centavo de dólar, lo que convierte un tier hot con sobrepeso en un gasto recurrente, no en un costo único. Las memorias warm y cold son baratas porque solo cargan cuando el trabajo lo exige.

> DEFINICIÓN
>
> La memoria hot es cara porque carga en cada solicitud y sus tokens se cobran en cada turno. Las memorias warm y cold son baratas porque solo cargan cuando se activan. El error que la mayoría de los equipos comete es poner todo en hot.

Workspace, autoría, runtime: las tres responsabilidades que sirve cada tier

Los tres tiers también se mapean en tres responsabilidades, y nombrarlas hace obvia la propiedad. La capa de workspace responde a la pregunta "qué es siempre verdad sobre esta base de código." Es el copilot-instructions.md más los archivos .github/instructions/*.instructions.md con scope, guiados por globs applyTo. Juntos son el contrato del equipo, y cargan en cada solicitud. Un subconjunto con scope de ruta refina las reglas globales sin reemplazarlas; cuando varios archivos coinciden con la misma ruta, hacen merge en lugar de reemplazar. Así es como una regla de React con scope en src/components y las reglas globales del workspace cargan lado a lado.

La capa de autoría responde a la pregunta "qué invoco." Agents, chat modes, skills y prompt files son conocimiento reutilizable empaquetado como artefactos y vinculado a una tarea, no a un archivo. La regla de selección es simple: elige el primitivo por el mecanismo de invocación, no por el contenido. Un agent se invoca por nombre para una clase entera de tareas, como revisión o pruebas. Un chat mode cambia la persona por la duración de una conversación. Un skill carga automáticamente

cuando la solicitud del usuario coincide con su descripción. Un prompt file se ejecuta detrás de un slash command que el usuario escribe a propósito.

La capa de runtime responde a la pregunta "qué corre alrededor del agente." Los hooks interceptan cada llamada a herramienta, antes y después, para bloquear, registrar o reescribir. Las conexiones MCP extienden el alcance del agente hacia sistemas externos, bases de datos, issue trackers, observabilidad, APIs internas, a través de un único protocolo descubierto al inicio de la sesión. Los plugins empaquetan capacidades de terceros detrás de un único manifiesto. Los settings de VS Code ajustan el comportamiento determinístico del modelo. Las cuatro superficies son responsabilidad del platform team, configuradas una vez y cambiadas rara vez. Editar la configuración de runtime por feature o por sprint es el antipatrón que esta capa existe para prevenir.

La regla: promueve a hot solo lo que es verdad en cada línea del workspace

Una regla decide dónde pertenece una regla propia. Reserva el tier hot para lo que es verdad en cada línea de código del workspace: convenciones de nomenclatura, política de seguridad, baselines de performance. Si una regla aplica solo a una tarea, un tipo de archivo o un directorio, no pertenece a hot; pertenece al tier warm como una instrucción con scope, un agent, un skill o un prompt file. Si configura el entorno alrededor del agente en lugar del contenido del prompt, pertenece al tier cold. El corolario es operativo: aplica la política en el scope externo y no la policía por usuario. Bloquea el contrato a nivel de organización o de workspace, y deja que los scopes internos agreguen preferencias ergonómicas que no puedan comprometerlo.

El gap empírico es donde esta regla prueba su valor. Un estudio de 2.303 archivos de contexto reales en 1.925 repositorios (Chatlatanagulchai et al., 2025, arXiv 2511.12884) encontró que los equipos escriben lo que hace al agente funcional y omiten lo que lo hace seguro. Los comandos de build y ejecución aparecen en 62,3 por ciento de los archivos, los detalles de implementación en 69,9 por ciento y la arquitectura en 67,7 por ciento, pero las guardas de seguridad aparecen en apenas 14,5 por ciento y las reglas de performance en otro 14,5 por ciento. El resultado es un agente que produce código útil, pero no necesariamente seguro ni predecible. Seguridad y performance son exactamente las reglas que valen en cada línea del

workspace, así que pertenecen al tier hot por la regla anterior, y su ausencia allí es el gap más común y más caro que un rollout de Copilot deja abierto.

> DEFINICIÓN
>
> Promueve a hot solo lo que es verdad en cada línea del workspace: convenciones, política de seguridad, baselines de performance. Todo lo específico de tarea queda warm. Todo lo que configura el runtime queda cold. Aplica en el scope externo; no la policía por usuario.

Referencias

Fuentes primarias para el modelo de memoria en tres tiers, el gap empírico en los archivos de contexto reales y las superficies de configuración que este capítulo mapea.

- Vasilopoulos, Codified Context: Infrastructure for AI Agents in a Complex Codebase, la fuente del modelo de memoria hot, warm, cold: un sistema de 108.000 líneas a lo largo de 283 sesiones a una razón conocimiento-código de 24,2 por ciento.
- Chatlatanagulchai et al., Agent READMEs: An Empirical Study of Context Files for Agentic Coding, el gap empírico en 2.303 archivos de contexto: 62,3 por ciento cubren build y ejecución, solo 14,5 por ciento cubren seguridad.
- Customize AI in Visual Studio Code, la referencia de cómo cargan las superficies de configuración y cómo se resuelven los scopes de settings.
- GitHub Copilot Documentation, la plataforma de referencia para el comportamiento de agent mode, archivo de instrucciones y MCP de este capítulo.
- Model Context Protocol Specification, el estándar abierto detrás de las conexiones MCP del tier cold.

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 CAPA DE WORKSPACE

Qué es siempre verdad: los dos archivos que cargan en cada solicitud.

La capa de workspace responde a una pregunta: qué es siempre verdad sobre esta base de código. Es la fundación que todo proyecto con GitHub Copilot necesita, y son dos archivos con un trabajo cada uno. El primero es copilot-instructions.md, cargado en cada solicitud. El segundo es un conjunto de archivos de instrucción con scope que cargan solo donde aplican. Acierta esta capa y Copilot hace el onboarding del siguiente colaborador por ti. Falla y cada revisión atrapa los mismos errores evitables.

copilot-instructions.md: el archivo que el modelo lee cada vez

copilot-instructions.md vive en .github/copilot-instructions.md. Carga en cada solicitud de GitHub Copilot: chat, edits y agent mode. Es Markdown plano, sin frontmatter, y existe exactamente un archivo por workspace. Sin cascada, sin una segunda copia que lo sobrescriba. Es el archivo de mayor impacto que puedes escribir para Copilot, porque es el único contenido que llega al modelo en cada línea de código.

La restricción es el tamaño. Mantenlo por debajo de 1.000 tokens. Pasado ese presupuesto, Copilot resume el archivo y pierdes el determinismo: el modelo lee una versión comprimida y ya no puedes predecir qué reglas sobreviven. Un archivo plano que supera los 3.000 tokens se trunca de inmediato, y todo lo que queda después del corte drifta en silencio. Reserva este archivo para lo que es verdad en todas partes: el stack, las convenciones, la política de seguridad, la baseline de performance. Si una regla aplica solo a una tarea o una carpeta, no pertenece aquí.

Por qué un solo archivo deja de escalar

Un proyecto documentado en C# de 108.000 líneas, construido durante 70 días y 283 sesiones en un monorepo de 14 paquetes, concluyó que los manifiestos de archivo único colapsan pronto (Vasilopoulos, 2026). El proyecto mantuvo una razón conocimiento-código de 24,2% solo después de migrar a un modelo de memoria en tiers. La lección no es que copilot-instructions.md sea débil. La lección es que tiene un trabajo y un presupuesto, y todo lo que excede ese presupuesto pertenece a las capas por debajo de él.

Un ejemplo real, con las secciones de Seguridad y Performance que la mayoría de los equipos omite

Así se ve el archivo cuando es corto y específico. Nombra el stack en una línea: Next.js 14 App Router, TypeScript en modo estricto, Tailwind, Drizzle ORM sobre PostgreSQL, Vitest. Declara las convenciones como reglas que el modelo puede seguir: archivos en kebab-case, componentes React en PascalCase, pruebas co-localizadas junto al código que cubren, sin imports directos desde node_modules. Lista los comandos clave para dev, build, lint y test. Nada de eso es inusual.

Lo inusual son las dos secciones que la mayoría de los equipos deja fuera. Una sección de Seguridad: nunca registrar cuerpos de solicitud ni headers de autenticación, validar cada entrada externa antes de usarla y verificar la sesión antes de cualquier escritura en la base de datos. Una sección de Performance: prefiere server components, mantén los payloads pequeños y pagina las listas largas. Estas son las reglas que hacen que la salida de Copilot sea segura y predecible, no solo productiva. Cuestan cuatro líneas cada una y cierran la brecha que produce la mayoría de las sugerencias rechazadas.

Una investigación sobre 2.303 archivos de contexto reales en 1.925 repositorios midió exactamente esta brecha (Chatlatanagulchai et al., 2025). Los comandos de build y ejecución aparecieron en 62,3% de los archivos, los detalles de implementación en 69,9%, la arquitectura en 67,7%. Las reglas de seguridad aparecieron en apenas 14,5%. Las reglas de performance en otro 14,5%. El manejo de errores era todavía más raro. El modo de falla por defecto de una adopción de

Copilot no es la ausencia de un archivo de contexto. Es un archivo de contexto que hace al agente productivo y lo deja inseguro.

> DEFINICIÓN
>
> Las dos secciones que cierran la brecha son las dos que la mayoría de los equipos nunca escribe. Solo 14,5% de los archivos de contexto reales incluyen reglas de seguridad, y solo 14,5% incluyen reglas de performance. Agregar un bloque de Seguridad y un bloque de Performance a copilot-instructions.md es la corrección más barata disponible.

Instrucciones con scope: el glob applyTo que mantiene corto el archivo hot

El archivo de workspace se mantiene dentro del presupuesto al mover la orientación especializada fuera de él. Las instrucciones con scope viven en .github/instructions/*.instructions.md. Cada archivo lleva un glob applyTo en su frontmatter, y Copilot carga el archivo solo cuando un archivo abierto coincide con ese glob. Un archivo con scope applyTo: "src/components/**/*.tsx" llega al modelo cuando alguien edita un componente y permanece en silencio el resto del tiempo. Las reglas detalladas existen, pero solo cuestan tokens cuando son relevantes.

La regla que hace esto seguro es que las instrucciones se suman, no se sobrescriben. Varios archivos de instrucción pueden coincidir con la misma ruta, y todos cargan junto con el archivo de workspace. Nada reemplaza el contrato global; los archivos con scope lo refinan. Así es como mantienes copilot-instructions.md pequeño sin perder el detalle que un stack específico necesita. Las reglas de React viven en un archivo, las de SQL en otro, y el archivo hot solo carga lo que es verdad en todas partes.

Cuando una regla con scope no carga

Cuando Copilot ignora una regla con scope, la causa casi siempre es el glob. Verifica que el patrón applyTo realmente coincida con el archivo que tienes abierto antes de concluir que el modelo tiene un problema. Un patrón que apunta a src/components

no se dispara en un archivo dentro de app/components, y una regla escrita para la extensión equivocada nunca llega al modelo. Recorre el glob antes de discutir con el agente.

Cuatro patrones de scope, incluido el patrón de superficie de riesgo que la mayoría ignora

Cuatro patrones cubren la mayoría de los repositorios. El primero da scope por capa de framework: un archivo para src/components, uno para src/server, uno para src/db, cada uno con las reglas que pertenecen a esa capa. El segundo da scope por tipo de archivo: convenciones de pruebas en **/*.test.ts, estilo de documentación en **/*.md, reglas de revisión de query en **/*.sql. El tercero da scope por dominio o bounded context: un archivo por paquete en un monorepo, de modo que packages/billing y packages/auth reciben cada uno su propio contrato.

El cuarto patrón es el que la mayoría de los equipos nunca escribe, y es el de mayor valor. Da scope por superficie de riesgo. Pon las reglas de PCI en **/payments/**. Pon las reglas de endpoint privilegiado en **/api/admin/**. Pon las reglas de cambio destructivo en migrations/**. Estos son los caminos donde una sugerencia equivocada cuesta más caro, y mapean exactamente en la brecha de seguridad de 14,5% que midió la investigación. El archivo de workspace no puede cargar este nivel de detalle sin reventar su presupuesto. Un archivo de instrucción por superficie de riesgo sí puede, y solo carga cuando alguien toca el código que lo justifica.

> DEFINICIÓN
>
> El patrón de superficie de riesgo es la decisión de scope que la mayoría de los equipos deja pasar. Da scope a los guardrails detallados en los caminos donde una sugerencia equivocada es cara: pagos, endpoints de admin, migrations. Es la misma brecha de seguridad que midió la investigación, resuelta a nivel de archivo en lugar de a nivel de prompt.

Referencias

Fuentes primarias para la capa de workspace, la evidencia empírica y las superficies de configuración sobre las que se apoya este capítulo.

- Customize AI in Visual Studio Code, la documentación de referencia para copilot-instructions.md, los archivos de instrucción con scope y el glob applyTo.
- An Empirical Study of Context Files for Agentic Coding, la fuente de la brecha de seguridad y performance de 14,5% en 2.303 archivos de contexto reales.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, el estudio de caso de 108.000 líneas sobre por qué los manifiestos de archivo único colapsan y por qué la memoria en tiers es necesaria.
- GitHub Copilot Documentation, la plataforma de referencia para el agent mode y los patrones de configuración de este playbook.

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 CAPA DE AUTORÍA

Lo que invocas: agents, chat modes, skills y prompt files, y cómo elegir entre ellos.

La capa de autoría es donde empaquetas comportamiento reutilizable como artefactos. Contiene cuatro primitivos: agents, chat modes, skills y prompt files. Los cuatro están en el tier warm, lo que significa que GitHub Copilot los carga solo cuando algo los activa, nunca en cada solicitud. El error que cometen los equipos es elegir el primitivo equivocado porque comparan lo que cada uno contiene. La pregunta correcta es cómo se activa cada uno. Este capítulo recorre la decisión y luego cada primitivo, uno por uno.

El árbol de decisión: elige por cómo se activa, no por lo que contiene

Los cuatro primitivos de autoría se parecen en el papel. Cada uno es un archivo Markdown con frontmatter e instrucciones, cada uno moldea cómo responde Copilot, y cada uno vive bajo .github/. La tentación es elegir entre ellos leyendo su contenido, pero el contenido se superpone tanto que la comparación se estanca. La regla confiable es otra: elige por el mecanismo de invocación. Un agent se invoca por nombre. Un chat mode se selecciona en el picker. Una skill se compara automáticamente con tu solicitud. Un prompt file corre por slash command. El disparador, no el contenido, es el eje que los separa con claridad.

El mecanismo mapea la intención. Si quieres un especialista con nombre para una clase entera de tareas, eso es un agent. Si quieres cambiar el estilo de la conversación o la profundidad de la explicación, eso es un chat mode. Si quieres que Copilot busque conocimiento de dominio empaquetado por su cuenta, eso es una skill. Si quieres una receta de un solo uso que invocas deliberadamente, eso es un prompt file. En el orden de ensamblado, solo un primitivo de autoría se adjunta por turno, en el paso seis, después de que las instrucciones del tier hot y las

instrucciones con scope ya se cargaron. Así que el primitivo que invocas refina el contrato del workspace. No lo reemplaza.

> DEFINICIÓN
>
> La regla de selección en una línea: elige el primitivo de autoría por cómo se activa, no por lo que contiene. Invocado por nombre es un agent. Seleccionado en el picker es un chat mode. Comparado por descripción es una skill. Ejecutado por slash command es un prompt file.

Agents: personas con nombre, con sus propias herramientas y menor privilegio

Los archivos de agent viven en .github/agents/ con la extensión .agent.md. Cada uno define una persona especializada con sus propias instrucciones, su propio alcance de comportamiento y, lo más importante, su propia lista de herramientas permitidas. Un agent está activo a la vez. Cuando lo invocas por nombre, reemplaza la persona de chat por defecto para esa clase de trabajo. El campo tools es lo que distingue a los agents de todos los demás primitivos. Enumeras exactamente las herramientas que el agent puede llamar, y nada más queda disponible para él. Un agent revisor que solo lee y comenta no recibe acceso a shell ni a escritura. Eso es menor privilegio expresado como configuración.

El frontmatter también lleva un campo model, que enruta el agent a un modelo específico en el picker de Copilot. Un agent de clasificación barato puede correr en Haiku 4.5. Un agent de refactorización difícil puede correr en Opus 4.8 o en Claude Fable 5. Como la facturación corre sobre GitHub AI Credits, donde un crédito es un centavo, enrutar agents rutinarios a un modelo más pequeño es una palanca directa de costo, no una preferencia. El estudio de caso empírico sobre contexto codificado encontró diecinueve agents especialistas con un promedio de 711 líneas para los de mayor capacidad y 327 para los estándar, con más de la mitad de cada especificación siendo conocimiento de dominio del proyecto, no instrucción de comportamiento. Un agent es sobre todo lo que sabe, no cómo le dices que actúe.

Chat modes y skills: estilo de conversación versus conocimiento cargado automáticamente

Los chat modes viven en .github/chatmodes/ y difieren de los agents de tres maneras. No llevan lista propia de herramientas. Se seleccionan en vez de invocarse. Y permanecen activos entre turnos hasta que cambias. Un chat mode cambia el estilo y la autoridad de la explicación, no la capacidad. El modo teach anota cada sugerencia línea por línea. El modo summarize comprime. El modo rubber-duck devuelve preguntas. Cuando el comportamiento que quieres es sobre tono, profundidad o persona para una conversación entera, y no sobre qué herramientas se permiten, un chat mode es el primitivo correcto y un agent es excesivo.

Cuándo recurrir a una skill

Las skills viven en .github/skills/{nombre}/SKILL.md, cada una en su propia carpeta, para llevar archivos de referencia, scripts y plantillas junto a las instrucciones. La distinción que importa es el disparador. Copilot carga una skill automáticamente cuando tu solicitud coincide con la descripción de la skill en el frontmatter. No la nombras y no la seleccionas. Eso hace de las skills la respuesta cuando el mismo flujo de varios pasos se reexplica todo el tiempo. Escríbela una vez, descríbela bien, y Copilot la encuentra por su cuenta. El registro comunitario Awesome GitHub Copilot distribuye skills y prompts de esta forma, así que una buena descripción es lo que hace a una skill descubrible y reutilizable, en tu equipo y más allá.

Prompt files: slash commands de un solo uso para recetas conocidas

Los prompt files viven en .github/prompts/ con la extensión .prompt.md. A diferencia de las skills, nunca se activan automáticamente. El usuario los llama explícitamente con un slash command, así que /release-notes o /pr-summary corre una receta específica y predecible bajo demanda. Usa un prompt file cuando el flujo es discreto, los pasos son conocidos, y quien lo ejecuta ya sabe que quiere ejecutarlo. Un prompt file puede declarar sus propias herramientas y su propio modelo en el frontmatter, igual que un agent, pero es una sola invocación, no una persona persistente. Hace un trabajo y retorna.

La línea entre una skill y un prompt file es, de nuevo, el disparador. Una skill es para conocimiento que Copilot debería buscar por su cuenta cuando el momento lo pide. Un prompt file es para una receta que un humano ejecuta deliberadamente. Si te descubres deseando que Copilot hubiera sabido hacer algo, eso es una skill. Si sigues escribiendo la misma instrucción larga para que ejecute una tarea conocida, eso es un prompt file. Ambos mantienen el tier hot ligero por vivir en el tier warm y cargarse solo cuando se activan, lo que mantiene el contexto siempre cargado pequeño y los AI Credits gastados por turno predecibles.

Referencias

Fuentes primarias para los primitivos de autoría, sus mecanismos de invocación y la evidencia empírica detrás del diseño de agents y skills.

- Customize AI in Visual Studio Code, la documentación de referencia para agents, chat modes, skills y prompt files en VS Code.
- Agent READMEs: An Empirical Study of Context Files for Agentic Coding, la fuente de los 14,5% de los archivos de contexto que traen orientación explícita de seguridad.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, la fuente del benchmark de 711 y 327 líneas por agent en diecinueve agents especialistas.
- GitHub Copilot documentation, la plataforma de referencia para el agent mode y los primitivos de autoría de este capítulo.
- Awesome GitHub Copilot, el registro comunitario que distribuye skills, prompts y agents como artefactos compartibles.

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 GOBERNANZA DE RUNTIME

Qué corre alrededor del agente: la capa de runtime que gobierna cada respuesta de Copilot.

Las primeras partes de este playbook describen lo que GitHub Copilot sabe: las instrucciones de workspace que siempre lee y los artefactos de autoría que carga bajo demanda. Este capítulo describe lo que corre alrededor del agente mientras trabaja. Cuatro superficies de runtime (hooks, conexiones MCP, plugins y settings de VS Code) deciden qué puede hacer el agente, dónde obtiene datos frescos y cómo se bloquea su comportamiento en toda la organización. Ninguna de ellas lleva contenido de prompt. Gobiernan el loop. Bajo responsabilidad del platform team, configuradas una vez y cambiadas rara vez, son la diferencia entre un agent mode productivo y un agent mode seguro.

Hooks: guardrails determinísticos en cada llamada a herramienta

Los hooks viven en .github/hooks/*.json. Se disparan en eventos de ciclo de vida (preToolUse, postToolUse, onSessionStart, onPromptSubmit, onFileChange) y pueden permitir, denegar, mutar o registrar un evento antes o después de que Copilot actúe. Un hook es código que corre alrededor de una acción del agente, no dentro de ella. No le pide al modelo que se comporte. Aplica la regla de forma determinística, igual que el CI aplica un build. Un hook preToolUse que coincide con un comando de shell destructivo y devuelve deny impide que "limpia el cluster" se convierta en kubectl delete namespace prod. El modelo nunca tiene la oportunidad de ser ingenioso.

Los hooks importan sobre todo por un gap medido. En los 2.303 archivos reales de contexto de agente estudiados por Chatlatanagulchai y colegas en 2025, la guía de seguridad explícita aparece en apenas 14,5% de los repositorios. Eso deja 85,5% sin ninguna regla de seguridad escrita que el modelo pudiera seguir. Los hooks son el

backstop determinístico para ese 85,5%. Donde un archivo de instrucciones dice "por favor, no lo hagas", un hook simplemente se niega. Cinco líneas de JSON en un evento preToolUse evitan el incidente que el equipo todavía no ha tenido y, a diferencia de un prompt, la garantía no depende de qué modelo esté en el Copilot picker ni de cómo se formuló la solicitud.

Conexiones MCP y plugins: datos frescos y bundles versionados

MCP, el Model Context Protocol, es el estándar abierto que Copilot usa para alcanzar sistemas externos en runtime: bases de datos, rastreadores de issues, plataformas de observabilidad, APIs internas. Los servidores se declaran en .mcp.json para el workspace o en settings.json para el usuario, se descubren una vez por sesión, y cada uno expone tools, resources y prompts que el agente puede llamar. La regla de cuándo agregar uno es simple. Si el dato vive fuera del repositorio y cambia entre sesiones (issues abiertos, líneas de log, registros en una tabla), pertenece detrás de un servidor MCP, no pegado en el chat. Si la acción que un servidor expone es destructiva, envuélvela con un hook preToolUse. Cada llamada MCP gasta latencia y GitHub AI Credits, así que las credenciales de menor privilegio y una superficie de herramientas estrecha son parte de la configuración, no un detalle posterior.

Los plugins resuelven la distribución. Un plugin empaqueta agents, chat modes, skills, prompt files, hooks y declaraciones MCP detrás de un único manifiesto plugin.json, fijado en una versión. Es como un estándar viaja por muchos repositorios sin copy-paste. En el momento en que un equipo se descubre copiando el contenido de .github/ de un repositorio a otro, el drift ya empezó: cuarenta repositorios con cuarenta archivos de instrucciones editados a mano divergen en un trimestre. Un plugin, fijado por repositorio, convierte eso en un único pull request contra el plugin. El registro de la comunidad, Awesome GitHub Copilot, distribuye el trabajo compartido de la misma manera. Los plugins son configuración cold: instalados una vez, versionados y cambiados por revisión, nunca editados por feature.

Precedencia de settings: el contrato del equipo baja, el gusto personal sube

Todo comportamiento de Copilot que no es un archivo es un setting, y los settings se resuelven en orden estricto, del menos específico al más específico: default (integrado en el producto), user (~/.../User/settings.json, nunca commiteado), workspace (.vscode/settings.json, commiteado en el repositorio) y workspace folder (para un paquete en un setup multi-root). El scope más específico gana, con una excepción que importa. En la cima está la política de GitHub Enterprise, que fija floors: exclusión de contenido, telemetría y un bloqueo de modelo que ningún setting de workspace o de usuario puede sobrescribir. Una organización que bloquea la elección de modelo en el Copilot picker, por ejemplo entre Claude Fable 5, Opus 4.8, Sonnet 5, Gemini 3.1 Pro o un modelo GPT-5.x, lo hace aquí, y el bloqueo se mantiene hasta el nivel más interno.

La herencia es unidireccional por diseño. La organización fija floors, el workspace fija el contrato del equipo, el usuario personaliza, y la sesión (archivos fijados, pestañas abiertas, la selección actual) es la capa más interna y más efímera. El scope externo no puede sobrescribirse desde adentro, y los settings internos nunca se filtran hacia afuera, así que una preferencia personal nunca se commitea al repositorio por accidente. El principio operativo se sigue de la forma: aplica en el scope externo, no vigiles por usuario. Bloquea la política a nivel de organización o de workspace, pon useInstructionFiles e instructionsFilesLocations en .vscode/settings.json para que el contrato del equipo sea real y no aspiracional, y deja que cada desarrollador agregue preferencias ergonómicas que no puedan comprometer el contrato.

Orden de ensamblado: los nueve pasos que dicen por qué se ignoró una regla

Cuando un desarrollador envía un prompt, Copilot ensambla el contexto efectivo en una secuencia determinística de nueve pasos. Los pasos uno a tres resuelven los settings en orden de precedencia, default luego user luego workspace. El paso cuatro carga la hot memory, .github/copilot-instructions.md, incondicionalmente. El paso cinco hace match de las instrucciones con scope, evaluando cada glob applyTo contra los archivos abiertos y mergeando los conjuntos que coinciden. El

paso seis adjunta el primitivo de autoría activo, sea el agent, chat mode, skill o prompt file en juego. El paso siete corre los hooks onPromptSubmit, que pueden denegar, mutar o anotar la solicitud. El paso ocho descubre las tools MCP, listando los servidores conectados y exponiendo su catálogo. El paso nueve envía al modelo el system prompt ensamblado, el catálogo de herramientas y el mensaje del usuario.

El valor de conocer el orden es diagnóstico. Cuando Copilot parece ignorar una regla, el modelo casi nunca es la causa. El bug está en el paso tres (el setting resuelto en el scope equivocado), en el paso cinco (el glob applyTo no coincide con el archivo que está de hecho abierto) o en el paso siete (un hook mutó o bloqueó la instrucción en silencio). Recorre los nueve pasos de arriba hacia abajo antes de discutir con el modelo. El mismo determinismo que hace la secuencia depurable también la hace auditable: el mismo repositorio en el mismo commit ensambla el mismo contexto cada vez, así que una regla que funcionó ayer y no hoy apunta a un cambio en uno de nueve lugares nombrados, no a un estado de ánimo del modelo.

> DEFINICIÓN
>
> Gobernanza de runtime en una línea: los hooks deciden qué puede hacer el agente, MCP decide dónde obtiene datos frescos, los plugins deciden cómo viaja el estándar, y los settings deciden quién puede cambiar cualquiera de ellos. Cuando se ignora una regla, recorre los nueve pasos de ensamblado de arriba hacia abajo. La respuesta está casi siempre en el paso tres, cinco o siete, no en el modelo.

Referencias

Fuentes primarias para las superficies de runtime: el gap de seguridad que justifica los hooks, el estándar MCP y su modelo de amenazas, el registro de plugins, la precedencia de settings y la evidencia de herencia.

- Chatlatanagulchai et al., An Empirical Study of Agent Context Files, el gap medido: la guía explícita de seguridad aparece en apenas 14,5% de 2.303 archivos reales de contexto, el 85,5% que los hooks respaldan.

- Model Context Protocol Specification, el estándar abierto que Copilot usa para alcanzar sistemas externos en runtime.
- MCP Security, Threat Model and Mitigations, la base para envolver cualquier acción destructiva de MCP en un hook preToolUse.
- Awesome GitHub Copilot, el registro de la comunidad que distribuye plugins, agents, chat modes y prompts.
- VS Code, Settings Precedence, la referencia de cómo se resuelven los scopes default, user, workspace y folder.
- Vasilopoulos, Codified Context Infrastructure, el estudio de caso de 108.000 líneas que sostuvo una única constitución de workspace a lo largo de 283 sesiones con razón conocimiento-código de 24,2%.

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 LUNES POR LA MAÑANA

De cero a baseline en una semana.

La superficie de diez capas no vale nada si nadie la toca. Este capítulo es la secuencia mínima de commits que lleva a un equipo de cero a una baseline de configuración real en una semana. Un commit por día, cada uno cerrando una brecha específica que la investigación documenta. Para el viernes, las capas de workspace, autoría y runtime ya tienen un primer inquilino, y el comportamiento de GitHub Copilot en el repositorio pasa a ser auditable en lugar de tribal.

La secuencia de commits, día a día

Lunes: crea el .github/copilot-instructions.md con cuatro secciones: Stack, Convenciones, Seguridad y Performance. Mantén el archivo por debajo de unos 1.000 tokens. Las secciones de Seguridad y Performance son las que más importan, porque el estudio de 2.303 archivos de contexto reales encontró guía de seguridad en apenas 14,5% de ellos y reglas de performance en otro 14,5%. Escribir esas dos secciones el primer día cierra la brecha que produce código productivo pero inseguro.

Martes: agrega el .vscode/settings.json con useInstructionFiles activado e instructionsFilesLocations apuntando a .github/instructions. Es el paso que hace real el contrato del equipo en lugar de aspiracional. Commitea las settings de workspace para que cada colaborador cargue el mismo comportamiento. Deja las preferencias personales en las settings de usuario, que nunca llegan al repositorio.

De miércoles a viernes: scope, guardrails, reúso

Miércoles: escribe un archivo de instrucción con scope en .github/instructions/ con un glob applyTo apuntando al directorio más riesgoso del repositorio, migrations, pagos o una superficie de API administrativa. Es el paso de un único archivo monolítico, que deja de escalar alrededor de 3.000 tokens, al modelo de tres tiers. Jueves: escribe un hook en .github/hooks/ que deniegue comandos de shell

destructivos antes de que se ejecuten. Cinco líneas de JSON son el respaldo determinístico para los caminos de código que el modelo no siempre protege por sí solo. Viernes: extrae un flujo recurrente de varios pasos a un skill en .github/skills/ con un campo de descripción fuerte, para que el agent mode lo encuentre por match de descripción en lugar de que un colega lo reexplique cada sprint.

A la semana siguiente, empaqueta los cinco artefactos en un plugin e instálalo en los tres repositorios principales. Un plugin reúne agents, chat modes, skills, prompt files, hooks y declaraciones MCP detrás de un único manifiesto, fijado a una versión. Los cambios pasan entonces por un único pull request en el plugin, en lugar de cuarenta ediciones de copiar y pegar en cuarenta repositorios.

> DEFINICIÓN
>
> Un commit por día: lunes el copilot-instructions.md, martes el settings.json, miércoles una instrucción con scope, jueves un hook, viernes un skill, a la semana siguiente un plugin. Cada commit da a una capa su primer inquilino y cierra una brecha que la investigación nombra.

Expectativas honestas: qué hará y qué no hará la jerarquía

La baseline no va a convertir a un equipo junior en un equipo senior. GitHub Copilot amplifica lo que está escrito. Si una convención vive solo en la cabeza de alguien, el modelo no puede seguirla, y la jerarquía no cambia nada hasta que la convención se codifica. Las capas son un lugar para poner conocimiento, no un sustituto de tenerlo.

Lo que la baseline sí hará es volver reproducible el comportamiento. La evaluación de Zoominfo sobre un rollout con más de 400 desarrolladores reportó una tasa de aceptación de sugerencias sostenida cerca del 30% una vez que las settings de nivel de workspace se estandarizaron, en lugar de resultados que variaban de una máquina a otra. Una baseline reproducible del 30% es un número que un equipo puede defender en un code review y mejorar de forma deliberada. El enfoque de

archivo único no ofrece eso, porque drifta en silencio en el momento en que el archivo crece más allá del presupuesto del modelo.

> DEFINICIÓN
>
> Lo que no hará: hacer aparecer convenciones no documentadas. Lo que hará: dar al equipo una baseline de aceptación reproducible cerca del 30% (Zoominfo 2025) en lugar de historias de funciona-en-mi-máquina.

Gobernanza, no prompt engineering: hacer cada capa auditable y con dueño

La jerarquía no es una checklist de features. Es un modelo de gobernanza para IA en ingeniería de software: quién declara el contexto, quién autora instrucciones y qué corre en cada prompt. Cada una de las diez capas tiene un dueño. Las instrucciones de workspace y las instrucciones con scope pertenecen al equipo como un contrato, commiteadas en el repositorio y revisadas como cualquier otro código. Agents, chat modes, skills y prompt files pertenecen a quien es dueño de la tarea que codifican. Hooks, conexiones MCP, plugins y settings pertenecen al platform team, porque configuran el runtime alrededor del agente, no una feature aislada.

La auditabilidad se desprende de esa titularidad. Cada regla vive en un archivo versionado, revisado en un pull request, atribuido a un commit. Cuando Copilot ignora una regla, la corrección es recorrer el orden de ensamblado de arriba hacia abajo, verificando el scope de las settings, luego el glob applyTo, luego cualquier hook que mute la solicitud, en lugar de discutir con el modelo. Aplica la política en el scope externo y no vigiles por usuario: bloquea los floors de la organización y del workspace, y deja que los scopes internos agreguen preferencias ergonómicas sin romper el contrato. Los equipos que tratan la jerarquía como gobernanza entregan. Los equipos que la tratan como prompt engineering se estancan.

Referencias y lecturas adicionales

Fuentes primarias para la baseline día a día, las brechas empíricas que cierra y los números de aceptación que apunta.

- Agent READMEs: An Empirical Study of Context Files for Agentic Coding (Chatlatanagulchai et al., 2025), la fuente para la brecha de 14,5% en seguridad y performance que el commit del lunes cierra, en 2.303 archivos de contexto reales.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase (Vasilopoulos, 2026), el estudio de caso de 108.000 líneas y 283 sesiones detrás del modelo de tres tiers y del colapso del archivo único por encima de 3.000 tokens.
- A Four-Phase Evaluation of GitHub Copilot Across 400+ Developers (Zoominfo, 2025), la fuente para la tasa de aceptación reproducible cerca del 30% una vez que las settings de workspace se estandarizan.
- Customize AI in Visual Studio Code, la referencia para archivos de instrucción, instrucciones con scope y las settings que hacen real el contrato del equipo.
- GitHub Copilot Documentation, la referencia de plataforma para agent mode, copilot-instructions.md y las capas de runtime que este capítulo configura.
- Model Context Protocol Specification, el estándar abierto para las conexiones MCP que un plugin puede empaquetar junto con hooks y skills.

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