SEMANTIC CONTEXT LAYER FOR AGENTIC DEVOPS · PLAYBOOK

El contexto determinístico decide si los agentes entregan o alucinan.

Los agentes de codificación con IA fallan por contexto, no por capacidad. SCL-AD trata el contexto semántico como un artefacto de build versionado: derivado del código con precisión de compilador, almacenado en un registry gobernado y entregado a GitHub Copilot vía copilot-instructions.md y MCP. La meta es 40 a 70 por ciento menos tokens por tarea, menos alucinaciones y código que pasa la revisión a la primera.

AUTORA

Paula Silva

CARGO

Software Global Black Belt

CONTACTO

paulasilva@microsoft.com

TIEMPO DE LECTURA

~ 45 minutos, de punta a punta

4	6	8
CAPAS DE ARQUITECTURA	CAPÍTULOS	RECETAS DE CONTEXTO

40-70%

META DE REDUCCIÓN DE TOKENS

CAPÍTULOS

Chapter 00

El problema y el concepto

Por qué los agentes fallan por contexto, cuánto cuesta esa falla en tokens, alucinaciones y confianza, por qué RAG y prompt engineering no alcanzan, y qué es SCL-AD en una frase.

Concepto →

Chapter 01

Arquitectura e inteligencia de código

La arquitectura de referencia de cuatro capas, los pilares de capacidad de cada capa y el modelo de código con precisión de compilador que ancla a GitHub Copilot en hechos.

Arquitectura →

Chapter 02

Recetas de contexto

Los ocho extractores canónicos, de Structural Inventory a Deployment Context: la pregunta que cada uno responde, el formato que emite y cuándo ejecutarlo.

Recetas →

Chapter 03

Registry, distribución e integración

El Context Registry que almacena y versiona artefactos, las políticas de refresh y acceso, y los patrones que entregan contexto a GitHub Copilot.

Distribución →

Chapter 04

Gobernanza, adopción y métricas

Doce controles para entornos regulados, la hoja de ruta en cinco fases del piloto a la plataforma y las métricas que prueban que Copilot mejoró, no solo se abarató.

Gobernanza →

Chapter 05

Build vs buy, y referencia

Los tres caminos de implementación en el stack de GitHub y Microsoft, el framework de decisión de seis preguntas, el glosario esencial y las referencias consolidadas.

Decisión →

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

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 PARTE I, EL PROBLEMA Y EL CONCEPTO

El problema y el concepto: por qué fallan los agentes, y qué es SCL-AD.

Los agentes de codificación con IA no fallan porque el modelo sea débil. Fallan porque operan sobre contexto incompleto, reconstruido o fabricado. En un repositorio pequeño esa falla es invisible; en un parque enterprise de miles de repositorios es el principal motor de costo. Este capítulo nombra el patrón de falla, mide su costo y define la corrección en una frase.

El patrón de falla: los agentes fallan por contexto, no por capacidad

Los despliegues enterprise de agentes de codificación con IA, con GitHub Copilot en agent mode liderando el campo, comparten un patrón de falla común. El agente reconstruye el contexto en cada interacción: relee archivos, reanaliza imports y vuelve a inferir fronteras arquitectónicas en cada tarea. Después genera código que viola convenciones que la base de código ya codifica. La causa raíz no es el agente, ni el modelo. Es la ausencia de una representación autoritativa y legible por máquina de cómo funciona realmente la base de código.

Tres patrones de falla se repiten entre despliegues. El impuesto de reconstrucción de tokens se paga cada vez que un agente abre una sesión: sin contexto semántico persistente, todo lo que la sesión anterior aprendió se descarta y se recomputa, cómputo desperdiciado y facturado por token. La brecha de inferencia aparece cuando el agente adivina hechos que la base de código ya codifica: resolución de símbolos, versiones de dependencias transitivas y fronteras de servicios no son opiniones, son propiedades determinísticas del código, y cuando el agente las infiere a partir de texto crudo introduce alucinaciones en un razonamiento por lo demás sólido. La deriva de convenciones se manifiesta en la calidad del código: cada repositorio tiene reglas implícitas para el manejo de excepciones, el logging y la

organización de pruebas, y un agente que no ve esas reglas produce código que compila pero no pertenece.

El costo: tokens, alucinaciones y confianza erosionada

Estos patrones se componen, y la cuenta llega en tres monedas. La primera son los tokens: a medida que la adopción escala de equipos piloto a toda la plataforma, el costo de tokens crece de forma superlineal con el número de repositorios, porque cada sesión en cada repositorio vuelve a pagar el impuesto de reconstrucción. La segunda son las alucinaciones: hechos inferidos que nunca fueron hechos, errores de tipo, uso de APIs obsoletas y violaciones de convención que se cuelan en código por lo demás plausible. La tercera es la confianza: cada cambio alucinado que un revisor atrapa hace más difícil aceptar la siguiente sugerencia del agente, y la deriva de convenciones crea una carga de revisión y retrabajo que anula las ganancias de productividad que el agente debía entregar.

La evidencia externa le pone un número al problema. Un estudio de 2025 de METR encontró que desarrolladores experimentados tardaron 19% más en completar tareas usando herramientas de IA en repositorios grandes y maduros, con menos de la mitad de las salidas de IA aceptadas tal cual. La causa no era la capacidad de razonamiento del modelo; era la ausencia de contexto estructurado sobre cómo funciona realmente la base de código. Ese resultado es la formulación más nítida de la tesis de este playbook: en bases enterprise, el contexto, no la calidad del modelo, es la restricción que limita la codificación agéntica.

Por qué los enfoques existentes no alcanzan

Cuatro enfoques comunes atacan partes del problema, y ninguno lo resuelve estructuralmente. La generación aumentada por recuperación (RAG) recupera fragmentos de código fuente por similitud con el prompt; el resultado es parcial y pesado en tokens, el agente todavía infiere las relaciones entre fragmentos, y la propia recuperación es probabilística. RAG sirve para documentación y preguntas y respuestas, no para anclar modificaciones de código. La búsqueda por embeddings mapea el código a un espacio vectorial y recupera por proximidad, lo que produce

coincidencias aproximadas sin garantías semánticas: dos funciones con embeddings parecidos pueden tener comportamientos de ejecución completamente distintos, y los embeddings no distinguen un método obsoleto de su reemplazo.

El prompt engineering codifica contexto a mano en system prompts, archivos de instrucciones del repositorio o plantillas de chat. Funciona a escala pequeña y se rompe a escala enterprise: la curaduría manual es frágil, inconsistente entre equipos y ciega a los hechos estructurales que cambian a medida que el código evoluciona. Las integraciones a medida, llamar a language servers, grafos de dependencias o analizadores estáticos desde el loop del agente, dan al agente acceso vivo a los hechos, pero cada llamada es un viaje de ida y vuelta en latencia y tokens, y el agente todavía arma el cuadro desde muchas consultas pequeñas. La limitación común es que los cuatro producen contexto bajo demanda, por sesión, por tarea. Tratan el contexto como efímero. La corrección estructural es tratarlo como un artefacto durable, versionado y determinístico, que existe con independencia de cualquier sesión de agente.

Qué es SCL-AD, en una frase

La palabra contexto está sobrecargada, así que SCL-AD usa una definición precisa. El contexto semántico es una representación legible por máquina de propiedades que pueden derivarse determinísticamente del código fuente: información de tipos resuelta, referencias de símbolos con sus sitios de definición y uso, grafos de dependencias directas y transitivas, grafos de llamadas entre funciones y servicios, interfaces de servicio y puntos de integración, configuración que afecta el comportamiento en ejecución, cobertura de pruebas mapeada al código de producción, fronteras arquitectónicas y las convenciones que la base de código codifica. Es distinto del contexto sintáctico (el texto crudo de los archivos), del contexto conversacional (el chat actual) y del contexto probabilístico (recuperación por similitud). Es lo que un ingeniero senior le diría a un recién llegado que preguntara cómo funciona realmente la base de código, y su propiedad definitoria es el determinismo: el mismo código en el mismo commit produce el mismo contexto cada vez que se computa.

Sobre esa definición, la Semantic Context Layer for Agentic DevOps es una capacidad de plataforma que deriva, almacena, distribuye y refresca contexto

semántico determinístico para agentes de codificación con IA, tratando el contexto como un artefacto de build versionado y no como un insumo efímero de sesión. El framework prescribe qué derivar del código, cómo derivarlo mediante análisis con precisión de compilador en lugar de inferencia, dónde almacenarlo, cómo exponerlo mediante interfaces estándar como el Model Context Protocol, y cómo gobernarlo en miles de repositorios. Operacionaliza ese comportamiento en cuatro pilares de capacidad: Precompute, Target, Execute y Coordinate. Y, deliberadamente, no es un producto: es una metodología neutral respecto de proveedores, que una organización puede implementar con herramientas nativas de GitHub alrededor de Copilot, con analizadores open source, con plataformas comerciales, o cualquier combinación, sin cambiar el comportamiento de los agentes.

> DEFINICIÓN
>
> SCL-AD en una frase: una capacidad de plataforma que deriva, almacena, distribuye y refresca contexto semántico determinístico para agentes de codificación con IA, tratando el contexto como un artefacto de build versionado y no como un insumo efímero de sesión. Determinismo sobre inferencia, reutilización sobre recomputación, neutralidad de proveedor sobre lock-in, auditabilidad sobre magia, refresco sobre obsolescencia.

Referencias

Fuentes primarias para el patrón de falla, la evidencia empírica y los conceptos sobre los que se apoya SCL-AD.

- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, la evidencia empírica: desarrolladores experimentados tardaron 19% más con herramientas de IA en repositorios grandes y maduros.
- Moderne Prethink, Semantic Code Context for Coding Agents, articulación de referencia de la distinción entre determinismo e inferencia.
- Moderne, Lossless Semantic Tree, ejemplo concreto de modelado de código con precisión de compilador.

- Anthropic, Effective context engineering for AI agents, guía fundamental sobre el contexto como preocupación de primera clase.
- GitHub Copilot Documentation, la plataforma de referencia para los patrones de integración de agentes de este playbook.
- Model Context Protocol Specification, el estándar abierto para el intercambio de contexto entre agentes.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 SCL-AD, ARQUITECTURA DE REFERENCIA

Arquitectura e inteligencia de código: el sustrato de SCL-AD.

SCL-AD organiza el contexto semántico en cuatro capas con contratos explícitos. Este capítulo recorre la arquitectura de referencia, los pilares de cada capa y la inteligencia de código que fundamenta a GitHub Copilot: índices, símbolos y precómputo.

Cuatro capas, una dirección de flujo

SCL-AD se estructura en cuatro capas horizontales, cada una con responsabilidad única, interfaces bien definidas y evolución y gobernanza independientes. La Capa 1, Inteligencia de Código, produce un modelo con precisión de compilador y preservación de formato de cada repositorio en alcance, el sustrato de todo lo demás. La Capa 2, Recetas de Contexto, reúne extractores reutilizables que corren contra ese modelo y producen artefactos semánticos en Markdown, CSV, JSON y CALM. La Capa 3, el Registro de Contexto, almacena y distribuye esos artefactos. La Capa 4, Integración de Agentes, conecta el registro con GitHub Copilot, agentes personalizados, clientes MCP y extensiones de IDE.

La información fluye hacia arriba: el modelo alimenta las recetas, las recetas producen artefactos, el registro los sirve, los agentes los consumen en ejecución. Los flujos operativos corren al revés: disparadores de refresco, gobernanza y controles de acceso bajan como preocupaciones transversales. En la topología de referencia, repositorios en GitHub Enterprise o Azure DevOps alimentan un pipeline en GitHub Actions o Azure Pipelines, donde corren las Capas 1 y 2; el pipeline publica los artefactos en el registro; y los runtimes de agente consumen vía servidor MCP, archivos estáticos o API. Cada caja es una frontera de despliegue; cada flecha, un contrato estable.

> DEFINICIÓN
>
> Identidad, observabilidad, seguridad y ciclo de vida atraviesan todas las capas. Regla ancla: un agente que no puede leer el repositorio X no recibe contexto derivado de él. La propagación de identidad es obligatoria.

Los pilares de capacidad, capa por capa

La Capa 1 debe satisfacer cuatro propiedades: precisión de compilador, el modelo coincide con lo que el compilador produciría, tipos resueltos, no inferidos; preservación de formato, espacios, comentarios y trivia conservados para referenciar los rangos originales de código; enlace entre repositorios, una biblioteca usada por diez servicios genera diez referencias resueltas, no una cadena; y regeneración determinística por commit. La Capa 2 reconoce siete categorías de recetas: Inventario Estructural, Inteligencia de Dependencias, Topología de Servicios en CALM, Extracción de Convenciones, Mapeo de Intención de Pruebas, Superficie de Configuración e Instantánea de Postura de Seguridad; misma versión de receta más mismo commit produce salida idéntica byte a byte.

La Capa 3 reconoce tres patrones de almacenamiento: co-localizado, artefactos en un directorio .context/ versionado con el código, adecuado para pilotos; un registro centralizado, indexado para búsqueda, refresco y control de acceso a escala de plataforma; y registros federados bajo restricciones de soberanía. Contra la obsolescencia se exigen tres disparadores de refresco: por commit, programado y por evento. La Capa 4 soporta cuatro modos de integración: inyección de archivos estáticos, como el copilot-instructions.md que GitHub Copilot lee al iniciar la sesión; un servidor MCP que el agent mode de Copilot consulta bajo demanda; integración nativa; e híbrido, volumen estático más detalle vía MCP.

Dos contratos normativos

Dos contratos de interfaz son normativos. Todo artefacto lleva un manifiesto: id direccionado por contenido, categoría de la receta, formato, URL del repositorio y SHA del commit, nombre y versión de la receta, timestamp de generación, clasificación como público, interno o restringido, y firmas, la base de trazabilidad,

control de acceso e integridad. La API de descubrimiento exige cinco operaciones: listar repositorios con contexto, listar artefactos por repositorio, filtrar por tipo, obtener contenido por hash y buscar por atributo. El transporte lo define la implementación; la semántica es normativa.

Inteligencia de código: precisión de compilador o error compuesto

La mayoría de las herramientas de análisis de código no tienen precisión de compilador: parsean sintaxis y aplican heurísticas, rápidas, pero asumen donde el compilador resuelve hechos. Una llamada service.process(item) en Java depende del tipo estático de service, del tipo en runtime si la llamada es virtual, del genérico de item, de la sobrecarga de process y de la inyección de dependencias activa. Un parser heurístico ve una llamada a método; un modelo con precisión de compilador resuelve un método específico en un tipo específico. Cuando GitHub Copilot pregunta qué llama a esta función, esa brecha separa una respuesta útil de una engañosa, y las capas superiores componen la imprecisión.

Siete capacidades definen una implementación válida, verificables con fixtures. Resolución de símbolos liga cada referencia a su definición, respetando alcance, visibilidad, imports y genéricos. Resolución de tipos asigna un tipo estático a cada expresión y, bajo despacho polimórfico, el conjunto de tipos posibles en runtime. Grafos de herencia e implementación capturan cadenas de extends e implements. Grafos de llamadas incluyen los destinos posibles del despacho dinámico, no solo el declarado. Resolución de dependencias fija versiones, transitivas computadas. Mapeo de configuración liga application.yaml, appsettings.json y .env al código que los consume. Preservación de source range ancla cada hecho a archivo, líneas y columnas.

Estrategias y herramientas por lenguaje

Tres estrategias cubren el espacio. Los compiladores nativos son el estándar de oro: OpenRewrite LST o Eclipse JDT para lenguajes JVM, Roslyn Workspaces API para .NET, TypeScript Compiler API, Pyright con LibCST para Python, con resolución conservadora y el resto marcado como dinámico, go/packages y go/types para Go, rust-analyzer para Rust, cada uno con toolchain propio. Plataformas unificadas como

CodeQL, el motor de análisis semántico de GitHub, reducen la complejidad operativa pero agregan dependencia de proveedor. Tree-sitter con resolvedores de tipos no alcanza la barra por sí solo: es un fallback cuyos artefactos llevan etiqueta de menor fidelidad. Para COBOL y PL/I, plataformas comerciales de modernización son el camino práctico.

Índices entre repositorios, precómputo a escala

El análisis de un repositorio está resuelto; la resolución entre repositorios es donde la mayoría de las implementaciones se rompe. Una función en el repositorio A llama a una clase en el repositorio B, que depende de una interfaz en el repositorio C; resolver la cadena exige los tres con versiones consistentes. Tres enfoques: un workspace estilo monorepo con Nx, Bazel o Turborepo, el más simple y de mayor fidelidad; un registro interno de paquetes, como Artifactory, Azure Artifacts o GitHub Packages, con precisión limitada por la cadencia de publicación; y el análisis multi-repositorio coordinado, con inventario, planificación en orden de dependencias y paso final de enlace, el patrón donde las plataformas comerciales aportan más valor.

A escala empresarial, miles de repositorios y millones de líneas, tres patrones mantienen el precómputo viable. El análisis incremental reanaliza solo lo que cambió. La caché por hash de contenido explota el determinismo: el mismo commit produce el mismo modelo, reutilizable entre ejecuciones de recetas. La ejecución paralela distribuye el análisis en worker pools. El análisis con precisión de compilador cuesta más que el sintáctico, así que acote el alcance a repositorios activos, en producción y asistidos por agentes de IA. En build versus buy: herramientas nativas cuando dominan uno o dos lenguajes, plataforma comercial para amplitud políglota, o híbrido; las Capas 2 y superiores contratan solo la interfaz del modelo, y la elección sigue reversible.

Referencias

Fuentes primarias de la arquitectura y de la inteligencia de código de SCL-AD.

- Moderne Technology, Lossless Semantic Tree, ejemplo de capa de Inteligencia de Código.

- Model Context Protocol Specification, estándar de la capa de integración de agentes.
- FINOS Common Architecture Language Model (CALM), formato estándar para artefactos de Topología de Servicios.
- CodeQL Documentation, la plataforma de análisis semántico de GitHub.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

02 SCL-AD, RECETAS DE CONTEXTO

Recetas de contexto: ocho extractores que convierten la base de código en hechos confiables para GitHub Copilot.

Una receta es un extractor determinista, acotado y declarativo sobre el modelo de inteligencia de código. Este capítulo recorre las ocho categorías canónicas: qué responde cada una, el formato que emite y cuándo ejecutarla.

Qué es una receta

Una receta es la unidad de ingeniería de contexto en SCL-AD: una función determinista del modelo de inteligencia de código hacia uno o más artefactos semánticos, parametrizada, versionada, probada y reutilizable entre repositorios. Tres propiedades definen una receta válida. Es determinista: con la misma versión del modelo y los mismos parámetros, produce una salida idéntica byte a byte, lo que hace confiable al registro de artefactos. Es acotada: extrae una clase específica de hecho semántico, no un barrido amplio; el alcance estrecho es lo que las hace componibles. Y es declarativa respecto a sus salidas: declara los tipos de artefacto, el formato y el schema de cada salida, para que los consumidores, del registro al agent mode de GitHub Copilot, validen lo que reciben.

Toda receta comparte la misma anatomía, sin importar el lenguaje. Un manifiesto YAML o JSON declara identidad y contrato: nombre, versión, categoría, entradas como la versión requerida del modelo y patrones de inclusión y exclusión, y cada salida con su formato y schema. La implementación recorre el modelo y emite los artefactos: clase Recipe en Java o Kotlin en OpenRewrite, analizador C# en Roslyn o scripts Python y TypeScript sobre la API de consulta del modelo. Pruebas basadas en fixtures emparejan repositorios pequeños con salidas esperadas y cubren camino feliz, entradas vacías, casos límite como dependencias circulares y código

generado, y errores. La documentación cierra el contrato; sin ella, no es seguro depender de una receta.

> DEFINICIÓN
>
> El ancla de este capítulo: SCL-AD reconoce ocho categorías canónicas de recetas, cada una respondiendo una pregunta distinta sobre la base de código. Los artefactos que emiten son lo que GitHub Copilot realmente lee: el patrón de auto-wiring mantiene punteros hacia ellos en .github/copilot-instructions.md, y así agent mode, custom agents y herramientas MCP fundamentan cada tarea en hechos extraídos, no en exploración costosa de archivos crudos.

Recetas 1 a 4: qué es el código y cómo está escrito

Las cuatro primeras categorías describen la forma estática de la base de código: contenido, dependencias, fronteras de servicio y estilo. Implemente las ocho donde la fundamentación deba ser integral y un subconjunto en los casos más acotados.

Structural Inventory

Responde: ¿qué hay en este repositorio? Emite Markdown más CSV con módulos, paquetes y tipos de nivel superior, descripciones breves, puntos de entrada y conteos de superficie pública. Ejecútela primero en todo repositorio; es la de mayor apalancamiento por token de esfuerzo, porque reemplaza la exploración en la que el agent mode de Copilot abre archivo tras archivo solo para aprender el layout.

Dependency Intelligence

Responde: ¿de qué depende y qué depende de él? Produce CSV más un SBOM, SPDX como referencia, cubriendo dependencias directas y transitivas con versión, licencia y advisories. Úsela cuando Copilot maneja upgrades, correcciones de vulnerabilidades o migraciones; la fuente la clasifica como la más accionable para agentes de migración y seguridad.

Service Topology

Responde: ¿qué servicios existen y cómo se relacionan? La salida es JSON CALM, el schema de arquitectura de FINOS, con servicios, interfaces y relaciones. Úsela en entornos multi-servicio, para que un custom agent de Copilot que propone un cambio entre servicios vea el radio de impacto real, no el inferido por los nombres de las carpetas.

Convention Extraction

Responde: ¿cómo se escribe el código en esta base? Emite Markdown con las convenciones observadas: patrones de logging, manejo de excepciones, nomenclatura y el grado de consistencia de cada una. El framework la clasifica como la de mayor impacto en la calidad de la salida de los agentes; úsela cuando el objetivo es que Copilot genere código que parece pertenecer a la base y pasa la revisión a la primera.

Recetas 5 a 8: qué hace el código en runtime

Las cuatro categorías siguientes capturan comportamiento y operación: pruebas, configuración, postura de seguridad y deployment. Importan más cuando los agentes pasan de leer código a cambiarlo.

Test Intent Mapping

Responde: ¿qué se prueba, cómo, y qué nos dice la prueba? Un CSV enlaza cada símbolo de producción con sus símbolos de prueba mediante etiquetas de intención como camino feliz o validación. Úsela siempre que Copilot modifique comportamiento; muestra qué pruebas protegen el código tocado y dónde la intención de la cobertura tiene brechas.

Configuration Surface

Responde: ¿qué configuración afecta el comportamiento en runtime? Markdown más JSON mapean cada clave a tipo, default, override de entorno y los puntos exactos del código que la consumen. Úsela en tareas que tocan settings o feature flags; evita que un agente cambie un valor sin ver todos los caminos que lo leen.

Security Posture Snapshot

Responde: ¿qué hechos relevantes de seguridad están codificados en el código? El Markdown cubre mecanismos de autenticación, primitivas criptográficas en uso, referencias a secretos y superficies expuestas. Úsela antes de que los agentes trabajen cerca de auth o criptografía, y como fundamentación compacta cuando Copilot asiste una revisión de seguridad.

Deployment Context

Responde: ¿qué corre dónde y bajo qué restricciones? Agregada en v1.1.0 como octava categoría canónica, emite Markdown más JSON conectando Docker, Kubernetes e infraestructura como código con el código de aplicación que afectan: imágenes, límites de recursos, réplicas, endpoints de health, ConfigMaps y Secrets. Es crítica para agentes que proponen cambios con impacto operacional: sin ella, un agente puede proponer un cambio que compila sin errores pero falla en runtime por una entrada faltante en el ConfigMap.

Autoría, ciclo de vida y por dónde empezar

Ocho lineamientos gobiernan la autoría de recetas. Prefiera el alcance estrecho a la amplitud. Produzca salida para humanos y máquinas: un artefacto Markdown para revisión y, cuando aplique, CSV o JSON para consumo programático. Respete el determinismo: ordene las salidas de forma estable y mantenga los timestamps fuera del cuerpo de los artefactos. Incorpore trazabilidad: cada hecho referencia su rango de origen en el código. Falle explícitamente ante la ambigüedad, emitiendo entradas no resueltas en lugar de adivinar. Versione por separado receta, modelo y schema del artefacto. Pruebe con fixtures reales y sanitizadas. Y mantenga los artefactos compactos: un repositorio de 100.000 líneas no debe producir un inventario de 100.000 líneas, porque cada artefacto compite por la ventana de contexto de una sesión de agente de Copilot. El ciclo de vida es el de software: autoría en un repositorio de recetas, prueba obligatoria antes del release, publicación en un registro bajo versionado semántico, deployment por configuración y deprecación como la de una biblioteca.

Por dónde empezar: sin una plataforma comercial, empiece con tres recetas, Structural Inventory, Convention Extraction y Dependency Intelligence, y expanda a

las ocho durante las Fases 2 y 3 del roadmap de adopción. Como orientación, Moderne Prethink publica dos recetas iniciales: UpdatePrethinkContextNoAiStarter, análisis puramente determinista que produce Dependency Intelligence, Test Intent Mapping, Service Topology y Structural Inventory, y UpdatePrethinkContextStarter, que agrega un grafo de conocimiento y resúmenes de pruebas generados por IA vía Bring Your Own Model. Ambas actualizan automáticamente los archivos de configuración de agentes, incluido .github/copilot-instructions.md, conectando los artefactos directamente con GitHub Copilot. Las recetas personalizadas, desde mapeos regulatorios para PCI-DSS, HIPAA, LGPD y SOX hasta extracción de APIs internas y multi-tenancy, siguen la misma anatomía y ciclo de vida. La prueba es simple: si la pregunta que responde una receta es determinista y reutilizable, es una receta; si depende del prompt o del estado de la sesión, es una plantilla de prompt y pertenece a la capa de integración de agentes.

Referencias

Fuentes para las ocho categorías de recetas, la anatomía de las recetas y las implementaciones de referencia.

- OpenRewrite Recipe Documentation, el modelo de referencia para el diseño de recetas.
- Moderne Prethink Documentation, ejemplos concretos de salidas de recetas de contexto y las definiciones de las recetas iniciales.
- Moderne Blog: Changing the AI context, referencia para las categorías semánticas que informaron este framework.
- FINOS CALM Specification, el schema para los artefactos de Service Topology.
- SPDX Specification, el formato SBOM de referencia para las salidas de Dependency Intelligence.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

03 CAPÍTULO 03

Registry, distribución e integración con agentes: del artefacto al agente.

Los artefactos semánticos solo generan valor cuando un agente los lee en el momento correcto. Este capítulo cubre el Context Registry que los almacena y versiona, las políticas de refresh y acceso que los mantienen confiables, y los patrones de integración que los entregan a GitHub Copilot y al ecosistema de agentes.

El Context Registry: donde viven los artefactos semánticos

El Context Registry se sitúa entre dos relojes: las recetas se ejecutan en cadencia de build (un commit, un cronograma, un evento), mientras los agentes consultan contexto en cadencia de sesión, en cada interacción del desarrollador. Cinco responsabilidades lo definen. Almacena artefactos de forma durable: lo publicado sigue recuperable hasta que se archive. Descubre: los agentes encuentran artefactos por repositorio, tipo, versión o atributo de contenido sin conocer el layout de almacenamiento. Versiona: cada artefacto recibe un identificador estable, direccionado por contenido. Impone acceso: cada lectura se verifica contra las mismas políticas que gobiernan el código fuente. Y observa: cada lectura y escritura queda registrada para reconstruir qué contexto vio cualquier agente en cualquier momento.

El descubrimiento responde tres preguntas. Por repositorio: dado un repo, devolver todos sus artefactos, con filtro por tipo o commit. Por capacidad: dado un tipo de artefacto, como topología de servicios, devolver los repositorios que lo tienen. Por contenido: consultas transversales, como qué repositorios usan cierta biblioteca por debajo de cierta versión. Cada artefacto lleva además una URI estable, con registry, repositorio, commit, tipo y versión de la receta: la referencia canónica, inmutable durante la vida del artefacto.

Tres patrones de almacenamiento, una curva de madurez

El almacenamiento co-localizado mantiene los artefactos dentro del repositorio que describen, típicamente en .context/ en la raíz: la configuración más simple, acceso idéntico al del código, versionado por Git, ninguna configuración extra para los agentes, pero sin consultas entre repositorios ni gobernanza centralizada. El registry centralizado es un servicio dedicado con APIs de escritura para recetas y de lectura para agentes: consultas cross-repo nativas y ciclo de vida central, al costo de operar un servicio nuevo. El almacenamiento federado enlaza varios registries mediante una capa común de descubrimiento, por unidad de negocio, región o dominio de compliance: la respuesta para residencia de datos e industrias reguladas. Empiece co-localizado, centralice a escala de plataforma, federe solo cuando la soberanía lo exija.

Versionado, refresh y acceso: mantener el contexto confiable

Cada cuerpo de artefacto se hashea con SHA-256, el identificador inmutable: ejecuciones que producen bytes idénticos comparten el mismo hash, deduplicando el almacenamiento. El manifiesto lleva cuatro versiones explícitas, receta, schema, modelo y registry, así un cambio de schema nunca invalida artefactos antiguos; solo cambia contra qué schema validan los agentes. Sobre los hashes, los alias hacen usable el registry: latest resuelve al artefacto más reciente de un repositorio y tipo, stable al más reciente que pasó la validación, y commit más SHA a un build específico.

El contexto obsoleto es el modo de falla a impedir, así que el refresh es multimodal. Por commit: reejecuta solo las recetas afectadas por los archivos cambiados, típicamente en minutos. Programado: diario para repositorios activos, semanal para los estables, capturando el drift de dependencias transitivas. Por evento: un aviso de seguridad regenera la Dependency Intelligence, un upgrade de framework regenera la Convention Extraction. Cada tipo tiene una ventana de frescura: 24 horas para Dependency Intelligence, Configuration Surface y Security Posture Snapshot; 7 días para Structural Inventory, Service Topology y Test Intent Mapping; 30 días para Convention Extraction. Los obsoletos no se eliminan automáticamente; su estado se señala para que los consumidores decidan.

DEFINICIÓN

El control de acceso sigue dos principios sin excepción. Propagación de identidad: si un solicitante no puede leer el repositorio X en el control de versiones, no puede leer los artefactos derivados de él en el registry. Mínimo privilegio mediante etiquetas de clasificación que solo pueden elevarse, nunca rebajarse. Cada lectura genera auditoría con identidad, URI, hash de contenido, resultado y contexto de llamada, retenida de 12 a 36 meses en entornos regulados.

Patrones de integración: cómo llega el contexto a GitHub Copilot

Las plataformas de agentes difieren en cómo consumen contexto, y la capa de integración debe ser sin pérdidas, acotada y controlable: nada de truncamiento ni paráfrasis en la frontera, inyección estática dentro de la ventana de contexto, consultas dinámicas dentro del presupuesto de tool calls, y ajuste de lo que cada agente ve sin tocar las recetas. El primer patrón, inyección estática, escribe artefactos seleccionados en rutas conocidas: un directorio .context/ con inventario condensado, convenciones, topología y un manifiesto de URIs a los artefactos completos. El objetivo primario son las repository custom instructions de GitHub Copilot en .github/copilot-instructions.md, leídas al inicio de sesión en todas las superficies de Copilot (IDE, GitHub.com, Copilot Chat). El pipeline inserta un bloque gestionado con punteros a .context/, preservando todo lo demás; sin ese auto-wiring, la integración deriva en días tras la primera edición manual. Nunca inyecte la Dependency Intelligence ni la Configuration Surface completas: son grandes y rara vez se necesitan enteras.

MCP para consultas profundas, híbrido para producción

El segundo patrón pone un servidor Model Context Protocol delante del registry, exponiendo herramientas como list_repository_artifacts, get_artifact, search_dependencies, find_callers, describe_service y find_convention. Aplica cuando la plataforma soporta MCP nativamente y la economía de tokens exige que solo lleguen al modelo las porciones relevantes. Hay dos topologías: un servidor

local por desarrollador, bajo su identidad, o uno compartido con identidad propagada vía OAuth. En ambos, las consultas deben responderse en segundos de un dígito en el peor caso, con caché por URI invalidada por los eventos de refresh del registry; la recuperación lenta es peor que ninguna, porque el agente agota el tiempo y vuelve a inferir. Producción combina los patrones como distribución híbrida: inyección estática de convenciones e inventario resumido en cada repositorio, un servidor MCP para análisis de impacto y dependencias, y API directa para tooling por lotes.

Mapeo por agente, GitHub Copilot primero

GitHub Copilot ofrece la superficie más completa para SCL-AD. Las repository custom instructions en copilot-instructions.md llevan la capa estática. Los custom agents de Copilot reciben subconjuntos curados por rol: un agente de migración recibe Structural Inventory, Dependency Intelligence y Convention Extraction; un agente de review recibe convenciones, Test Intent Mapping y Security Posture Snapshot. El Copilot Agent Mode y el Coding Agent, que ejecutan tareas de múltiples pasos, se benefician sobre todo del contexto dinámico vía MCP, con el servidor registrado en la configuración del usuario o de la organización. Alrededor de Copilot, la convención neutral AGENTS.md es el objetivo estático primario recomendado, con archivos específicos de plataforma generados desde la misma fuente: herramientas del ecosistema como Claude Code con CLAUDE.md y Cursor con .cursorrules reciben los mismos bloques gestionados sin un segundo pipeline. Los frameworks agénticos custom llaman a la API del registry o al servidor MCP directamente, el camino de mayor fidelidad.

> DEFINICIÓN
>
> Seis anti-patrones quedan explícitamente desaconsejados: volcar el registry entero en archivos de instrucción, parsing en vivo dentro del loop del agente, snapshots sin pipeline de refresh, endpoints MCP que difuminan la identidad del desarrollador, prompts mágicos para compensar contexto ausente y stores de contexto duplicados dentro de la plataforma del agente. Saltarse el registry, aunque sea temporalmente, crea artefactos imposibles de gobernar, refrescar o auditar.

Referencias

Fuentes para el contrato del registry, los modos de distribución y los patrones de integración.

- GitHub Copilot Documentation: repository custom instructions, GitHub, el objetivo de inyección estática copilot-instructions.md.
- Model Context Protocol Specification, el estándar para distribución dinámica de contexto orientada a consultas.
- AGENTS.md Convention, la convención emergente y neutral para archivos de instrucción de agentes.
- OCI Distribution Specification, fundación para implementaciones de registry basadas en OCI.
- SLSA Provenance Specification, referencia para procedencia y firma de artefactos.
- Moderne Blog: Changing the AI context, referencia para el patrón de auto-wiring de archivos de configuración de agentes.
- Anthropic Engineering: Effective context engineering for AI agents, guía de la industria sobre estrategia de contexto.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

04 PARTE IV, GOBERNANZA Y ADOPCIÓN

Gobernanza, adopción y métricas.

Controles que hacen seguro un registro de contexto en entornos regulados, la ruta del piloto a la plataforma y las métricas que separan un GitHub Copilot más barato de uno mejor.

El contexto no es dato neutro

El contexto semántico derivado del código fuente codifica decisiones de arquitectura, posturas de seguridad, dependencias y patrones propietarios. El registro que lo almacena describe cómo funciona realmente el software de la organización, y cada sesión de GitHub Copilot que lo consume hereda ese conocimiento. Mal gobernado, genera tres riesgos. Fuga: contexto de repositorio sensible llega a agentes sin autorización, y leer el contexto del servicio de pagos es, en la práctica, leer su arquitectura. Integridad: artefactos adulterados hacen a Copilot razonar sobre base corrompida, y la confianza es irrecuperable una vez rota. Compliance: datos regulados almacenados o transmitidos contra las reglas. El framework trata la gobernanza como par de la arquitectura, no como capa por encima.

Cinco principios, cinco roles

Cinco principios innegociables rigen la operación. Identity-first: toda lectura, escritura y actualización vinculada a una identidad autenticada; cuentas de servicio son identidades, 'el sistema' no lo es. Trace-by-default: todo artefacto trazable hasta la recipe, el código fuente y el operador. Least exposure: agentes y humanos ven el mínimo que exige la tarea, denegación por defecto. Integridad verificable: todo artefacto lleva firma o hash de contenido. Propósito documentado: toda recipe e integración declara por qué existe. Cinco roles operan el modelo: dueño de plataforma, stewards de recipes, dueños de repositorio, que deciden qué se expone y no pueden ser evadidos, revisores de seguridad y compliance y auditores, que leen logs, nunca artefactos.

Seis dominios de política

Seis dominios exigen decisiones documentadas: alcance, qué repositorios entran, con el código que impacta producción como default; clasificación, etiquetas como público, interno, confidencial y restringido; acceso, con propagación de identidad como base; actualización, ventanas de staleness y disparadores; retención, alineada con el código fuente y la regulación; e integración, qué agentes consumen el registro, revisados antes de llamarlo.

Doce controles y el rastro de auditoría

Doce controles implementan las políticas en cuatro familias. Identidad y acceso: identidad autenticada en cada lectura, espejo de autorización de los permisos del código fuente y acceso privilegiado just-in-time. Integridad: firma de artefactos, direccionamiento por contenido con detección automática de adulteración y atestación de recipe que liga cada artefacto a su versión de recipe. Ciclo de vida: artefactos críticos vencidos bloquean el consumo hasta refrescarse, los de repositorios archivados se archivan en 7 días y el borrado se registra, autoriza y notifica. Observabilidad: logs de lectura y escritura más detección de anomalías. El rastro de auditoría es append-only, retenido 12 meses por defecto y de 36 a 84 meses en entornos regulados. Incidentes tienen tres plantillas, integridad, divulgación y disponibilidad, todas con postmortem en 30 días.

> DEFINICIÓN
>
> La regla ancla es el espejo de autorización: leer un artefacto exige lectura en el repositorio de origen. GitHub Copilot ve lo que su usuario ya podía ver, nunca más.

Cinco fases, de la línea base a la plataforma

La fase 0, descubrimiento y línea base, toma de 2 a 4 semanas: inventarie agentes y herramientas, GitHub Copilot en qué tier y adopción por equipo; capture las métricas base; seleccione de 1 a 3 repositorios piloto; y decida build, buy o híbrido para la inteligencia de código. La fase 1, el piloto, toma de 4 a 8 semanas: implemente dos o tres de las siete recipes canónicas, empezando por inventario estructural, extracción

de convenciones e inteligencia de dependencias, almacene los artefactos en el .context/ del repositorio e inyéctelos en copilot-instructions.md para que Copilot ancle cada solicitud en ellos. Barra de salida: gasto de tokens por tarea cae 30% o más, calidad mejora y equipos reportan mejor salida.

La fase 2, expansión, toma de 3 a 6 meses: 10 a 50 repositorios diversos, registro centralizado, las recipes canónicas restantes, integración MCP para que el modo agente de Copilot consulte el registro bajo demanda y los primeros cinco controles vivos en producción. La fase 3, plataforma, toma de 6 a 12 meses: todos los repositorios en alcance incorporados, los doce controles operativos, la capa en el IDE, el code review y los portales internos, y ROI demostrado contra la línea base de la fase 0. La fase 4, optimización, es continua: recipes sin consumo se deprecan, integraciones evolucionan con las plataformas y costos se ajustan.

Cinco riesgos recurrentes de adopción

Éxito de piloto que no escala, mitigado con diversidad realista. Retrofit de gobernanza, el error más caro, mitigado habilitando identidad, auditoría y clasificación desde la fase 1, aun en baja fidelidad. Proliferación de recipes, mitigada nombrando stewards temprano y deprecando con agresividad. Lock-in de proveedor, mitigado imponiendo los contratos de interfaz incluso con un solo proveedor. Y ahorro de tokens sin calidad, un Copilot más barato que no es mejor, mitigado tratando calidad como criterio primario.

Seis métricas que prueban el valor

El gasto de tokens por tarea mide lo que un agente consume en una categoría de tarea, con meta de reducción de 40 a 70% sobre la base al final de la fase 2. La tasa de alucinación mide la fracción de salidas con error factual por contexto incorrecto, como Copilot citando una función inexistente o una API deprecada, con reducción medible al final de la fase 1. La adherencia a convenciones mide cuánto del código generado cumple las convenciones documentadas a la primera, con meta de al menos 30 puntos porcentuales al final de la fase 2.

El tiempo hasta la primera sugerencia útil mide del prompt a la primera salida de Copilot aceptada. La satisfacción del desarrollador es una encuesta periódica estilo NPS sobre confianza y utilidad; la disposición de uso suele ya ser alta. El throughput

de pull requests es terciario, atribuido con cautela. Dos métricas operativas completan el conjunto: frescura del registro en 95% o más y cobertura de auditoría en 100%.

El modelo de ROI

El modelo tiene tres componentes. Ahorro directo: 500 desarrolladores, 100 tareas asistidas por desarrollador por mes, base de 50.000 tokens por tarea, reducción de 50% y costo de $0,005 por 1.000 tokens rinden cerca de $62.500 por mes, y ese es el piso. Calidad: si la alucinación cae 50% y un error de agente con impacto en producción cuesta 8 horas de remediación, el ahorro de retrabajo domina al de tokens por un orden de magnitud. El apalancamiento de plataforma es el upside después de la fase 3, no la justificación de las fases 0 a 2. Costos: 2 a 4 ingenieros de plataforma hasta la fase 2, licencias, infraestructura y gobernanza. ROI positivo típico entre la salida de la fase 2 y el primer trimestre de la fase 3.

Referencias

Fuentes del modelo de gobernanza, la hoja de ruta de adopción y las métricas.

- NIST Cybersecurity Framework, NIST, la base de la organización del catálogo de controles.
- SLSA Specification, el modelo de proveniencia y atestación de la firma de artefactos.
- ISO/IEC 42001, la norma de sistemas de gestión de IA para despliegues agénticos.
- DORA State of DevOps Reports, la referencia para medir impacto de plataforma de desarrollo.
- GitHub Octoverse 2024, GitHub, datos de adopción de herramientas de IA en empresas.
- Effective context engineering for AI agents, Anthropic, el valor del contexto estructurado.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 BUILD VS BUY Y REFERENCIA

Build vs buy, y el material de referencia.

SCL-AD es vendor-neutral por diseño: el framework define qué es el contexto semántico y cómo debe comportarse, nunca qué comprar. Este capítulo compara los tres caminos de implementación bajo el lente de GitHub Copilot, condensa el glosario esencial y consolida las referencias.

Los tres caminos hasta producción

Toda organización que adopta el framework enfrenta la misma decisión: construir las cuatro capas internamente, comprar una plataforma comercial que ya las implementa o combinar ambas. El Camino A, build, implementa todo sobre las inversiones existentes, típicamente GitHub Enterprise más Microsoft Azure. Roslyn cubre .NET, OpenRewrite cubre la JVM, la TypeScript Compiler API y Pyright cubren web y Python, y CodeQL agrega grafos de llamadas entre lenguajes. Las recipes corren como workflows reutilizables de GitHub Actions, el Context Registry vive en Azure Blob Storage más Cosmos DB detrás de Azure API Management, y un job de GitHub Actions hace el auto-wiring de copilot-instructions.md y AGENTS.md en cada commit. La recompensa es soberanía, integración nativa con GitHub Copilot y personalización sin límite; el precio es un equipo dedicado de tres a cinco ingenieros de plataforma durante nueve a quince meses, con cada recipe escrita y mantenida internamente.

El Camino B, buy, adopta una plataforma comercial que entrega la capa más difícil en lugar de pedirte construirla. Moderne es el ejemplo más prominente: la Lossless Semantic Tree cubre JVM, JavaScript, TypeScript y COBOL, dos recipes iniciales de Prethink acompañan más de 5,000 recipes de OpenRewrite, y Moddy expone todo como un servidor MCP que GitHub Copilot, Claude Code y Cursor consumen directamente. Un piloto produce artefactos reales en semanas, con un equipo de dos a tres ingenieros enfocado en integración. Los costos: suscripción enterprise, dependencia del proveedor y personalización acotada a lo que la plataforma expone.

El Camino C, híbrido, es el patrón de producción más común: compra lo que escala entre lenguajes, inteligencia de código y catálogo de recipes, y construye lo singular de la organización: recipes de dominio, registry, servidor MCP y gobernanza sobre Entra ID y Sentinel.

Cómo decidir, y cuánto cuesta

El framework de decisión hace seis preguntas: cuántos lenguajes dominan producción, uno o dos inclinan a build, seis o más a buy; cuánta capacidad de ingeniería de plataforma está dedicada; qué tan urgente es el time-to-value; qué tan estrictos son soberanía y personalización; si la migración es el caso de uso primario; y el nivel de comodidad con la dependencia de un proveedor. El conteo de lenguajes y la capacidad de ingeniería dominan en la práctica. El cuadro de costos: build cuesta de USD 1.5M a 3M en nueve a quince meses, luego 1.5 a 2 FTEs continuos; buy cuesta de USD 400K a 800K en tres a seis meses más la suscripción, luego 0.5 a 1 FTE; el híbrido queda entre USD 800K y 1.5M en seis a nueve meses más suscripción, luego 1 a 1.5 FTEs. En cinco años los caminos suelen converger en costo total; lo que cambia es el flujo de caja, capex en build, opex en buy, y la capacidad que queda en la organización.

El proceso recomendado es corto y guiado por evidencia. Discovery, de una a dos semanas, documenta lenguajes, repositorios, inversiones de plataforma y restricciones de soberanía. Evaluación, de tres a seis semanas, corre un piloto pagado del proveedor en paralelo con un spike de factibilidad de build, uno o dos ingenieros durante dos semanas produciendo una rebanada funcional de Code Intelligence para el lenguaje dominante. Los artefactos son concretos: una línea base medida de consumo de tokens y tasa de alucinación, un prototipo de un pilar, típicamente Precompute, y una proyección de costos fundamentada. La decisión sigue de inmediato, con evidencia en lugar de intuición, y el camino elegido entra en la Fase 1 del roadmap de adopción.

> DEFINICIÓN
>
> La decisión es reversible. Quien empieza en build puede pivotar a buy si la capacidad de ingeniería resulta insuficiente, y quien empieza en buy puede pivotar a híbrido si emergen requisitos de personalización que la plataforma no cubre. Los contratos de interfaz del framework hacen esos pivotes manejables, y es exactamente por eso que SCL-AD se mantiene vendor-neutral por diseño.

El glosario esencial

Un puñado de términos sostiene el framework, cada uno con un único significado en los documentos de SCL-AD. Contexto semántico es una representación legible por máquina de propiedades derivadas de forma determinista del código fuente, distinta del contexto conversacional o probabilístico. Un modelo de inteligencia de código es una representación consultable y fiel al compilador de un repositorio; la Lossless Semantic Tree, originada por Moderne, es una implementación posible, no un requisito. Una recipe es un extractor reutilizable y parametrizado que corre contra ese modelo y produce artefactos, la unidad de ingeniería de contexto; el framework reconoce ocho categorías canónicas, de Structural Inventory a Deployment Context. El Context Registry almacena artefactos versionados y direccionados por contenido y los expone mediante una API de descubrimiento, con ventanas de staleness y refresh por commits, calendarios o eventos.

Del lado del agente, cuatro pilares de capacidad nombran lo que hace la capa: Precompute deriva conocimiento estructurado por adelantado, Target encuentra el código correcto mediante índices precomputados, Execute aplica cambios de forma determinista en muchos repositorios y Coordinate comparte contexto de cambio entre agentes, reduciendo duplicación y gasto de tokens. El auto-wiring mantiene los archivos de configuración de agentes, copilot-instructions.md, AGENTS.md y CLAUDE.md, apuntando a los artefactos actuales como parte del pipeline de recipes, para que GitHub Copilot nunca lea un puntero obsoleto. La inyección de archivos estáticos y el servidor MCP son los dos patrones principales de integración. BYOM permite llamar a un modelo aprobado para enriquecer artefactos con resúmenes o grafos de conocimiento, opcional y aislado del núcleo determinístico. Y una alucinación es una salida que afirma un hecho ausente del contexto real; el

determinismo, misma entrada, misma salida, es la propiedad que todo el framework despliega en su contra.

Referencias

Las fuentes fundacionales del posicionamiento, del glosario y del propio framework.

- Moderne Prethink, Semantic Code Context for Coding Agents, la articulación original del contexto semántico determinista.
- Moderne Agent Capabilities, la fuente de los cuatro pilares de capacidad.
- METR, Measuring AI Effects on Experienced Developer Productivity, evidencia empírica de que el problema de contexto es medible.
- Effective context engineering for AI agents, Anthropic, el contexto como preocupación de primera clase.
- The architect's guide to the AI-native developer platform, GitHub, la perspectiva enterprise detrás del camino de build.
- Model Context Protocol Specification, el estándar abierto detrás de la integración vía MCP.
- OpenRewrite Documentation, motor open-source de LST y recipes, disponible para los tres caminos.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps