SEMANTIC CONTEXT LAYER FOR AGENTIC DEVOPS · PLAYBOOK

Contexto determinístico decide se os agentes entregam ou alucinam.

Agentes de codificação de IA falham por contexto, não por capacidade. O SCL-AD trata contexto semântico como artefato de build versionado: derivado do código com precisão de compilador, armazenado em um registry governado e entregue ao GitHub Copilot via copilot-instructions.md e MCP. A meta é 40 a 70 por cento menos tokens por tarefa, menos alucinações e código que passa na revisão de primeira.

AUTORA

Paula Silva

CARGO

Software Global Black Belt

CONTATO

paulasilva@microsoft.com

TEMPO DE LEITURA

~ 45 minutos, fim a fim

4	6	8
CAMADAS DE ARQUITETURA	CAPÍTULOS	RECEITAS DE CONTEXTO

40-70%

META DE REDUÇÃO DE TOKENS

CAPÍTULOS

Chapter 00

O problema e o conceito

Por que agentes falham por contexto, quanto essa falha custa em tokens, alucinações e confiança, por que RAG e prompt engineering não bastam, e o que é o SCL-AD em uma frase.

Conceito →

Chapter 01

Arquitetura e inteligência de código

A arquitetura de referência em quatro camadas, os pilares de capacidade de cada camada e o modelo de código com precisão de compilador que ancora o GitHub Copilot em fatos.

Arquitetura →

Chapter 02

Receitas de contexto

Os oito extratores canônicos, de Structural Inventory a Deployment Context: a pergunta que cada um responde, o formato que emite e quando executá-lo.

Receitas →

Chapter 03

Registry, distribuição e integração

O Context Registry que armazena e versiona artefatos, as políticas de refresh e acesso, e os padrões que entregam contexto ao GitHub Copilot.

Distribuição →

Chapter 04

Governança, adoção e métricas

Doze controles para ambientes regulados, o roteiro em cinco fases do piloto à plataforma e as métricas que provam que o Copilot ficou melhor, não só mais barato.

Governança →

Chapter 05

Build vs buy, e referência

Os três caminhos de implementação na stack GitHub e Microsoft, o framework de decisão de seis perguntas, o glossário essencial e as referências consolidadas.

Decisão →

DICAS DE LEITURA

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

Paula Silva

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, O PROBLEMA E O CONCEITO

O problema e o conceito: por que agentes falham, e o que é o SCL-AD.

Agentes de codificação de IA não falham porque o modelo é fraco. Falham porque operam sobre contexto incompleto, reconstruído ou fabricado. Em um repositório pequeno essa falha é invisível; em um parque enterprise de milhares de repositórios, ela vira o principal motor de custo. Este capítulo nomeia o padrão de falha, mede seu custo e define a correção em uma frase.

O padrão de falha: agentes falham por contexto, não por capacidade

Implantações enterprise de agentes de codificação de IA, com o GitHub Copilot em agent mode liderando o campo, compartilham um padrão de falha comum. O agente reconstrói o contexto a cada interação: relê arquivos, reanalisa imports e volta a inferir fronteiras arquiteturais em cada tarefa. Depois gera código que viola convenções que a base de código já codifica. A causa raiz não é o agente, nem o modelo. É a ausência de uma representação autoritativa e legível por máquina de como a base de código realmente funciona.

Três padrões de falha se repetem entre implantações. O imposto de reconstrução de tokens é pago toda vez que um agente abre uma sessão: sem contexto semântico persistente, tudo o que a sessão anterior aprendeu é descartado e recomputado, computação desperdiçada e cobrada por token. A lacuna de inferência aparece quando o agente chuta fatos que a base de código já codifica: resolução de símbolos, versões de dependências transitivas e fronteiras de serviços não são opiniões, são propriedades determinísticas do código, e quando o agente as infere a partir de texto bruto, introduz alucinações em um raciocínio que de resto seria sólido. O desvio de convenções se manifesta na qualidade do código: cada repositório tem regras implícitas para tratamento de exceções, logging e

organização de testes, e um agente que não enxerga essas regras produz código que compila, mas não pertence.

O custo: tokens, alucinações e confiança erodida

Esses padrões se compõem, e a conta chega em três moedas. A primeira são os tokens: conforme a adoção escala de times-piloto para implantação em toda a plataforma, o custo de tokens cresce de forma superlinear com o número de repositórios, porque cada sessão em cada repositório paga o imposto de reconstrução de novo. A segunda são as alucinações: fatos inferidos que nunca foram fatos, erros de tipo, uso de APIs depreciadas e violações de convenção escorregando para dentro de código de resto plausível. A terceira é a confiança: cada mudança alucinada que um revisor pega torna a próxima sugestão do agente mais difícil de aceitar, e o desvio de convenções cria uma carga de revisão e retrabalho que anula os ganhos de produtividade que o agente deveria entregar.

A evidência externa coloca um número no problema. Um estudo de 2025 da METR constatou que desenvolvedores experientes levaram 19% mais tempo para concluir tarefas usando ferramentas de IA em repositórios grandes e maduros, com menos da metade das saídas de IA aceitas como estavam. A causa não era a capacidade de raciocínio do modelo; era a ausência de contexto estruturado sobre como a base de código realmente funciona. Esse resultado é a formulação mais afiada da tese deste playbook: em bases enterprise, o contexto, não a qualidade do modelo, é a restrição que limita a codificação agêntica.

Por que as abordagens existentes não bastam

Quatro abordagens comuns atacam partes do problema, e nenhuma o resolve estruturalmente. A geração aumentada por recuperação (RAG) recupera trechos de código-fonte por similaridade com o prompt; o resultado é parcial e pesado em tokens, o agente ainda infere as relações entre os trechos, e a própria recuperação é probabilística. RAG serve bem para documentação e perguntas e respostas, não para ancorar modificações de código. A busca por embeddings mapeia código para um espaço vetorial e recupera por proximidade, o que produz correspondências aproximadas sem garantias semânticas: duas funções com embeddings parecidos

podem ter comportamentos de execução completamente diferentes, e embeddings não distinguem um método depreciado do seu substituto.

O prompt engineering codifica contexto à mão em system prompts, arquivos de instrução do repositório ou templates de chat. Funciona em escala pequena e quebra em escala enterprise: a curadoria manual é frágil, inconsistente entre times e cega aos fatos estruturais que mudam conforme o código evolui. Integrações customizadas, chamar language servers, grafos de dependência ou analisadores estáticos de dentro do loop do agente, dão ao agente acesso vivo aos fatos, mas cada chamada é uma ida e volta em latência e tokens, e o agente ainda monta o quadro a partir de muitas consultas pequenas. A limitação comum é que as quatro produzem contexto sob demanda, por sessão, por tarefa. Tratam o contexto como efêmero. A correção estrutural é tratá-lo como um artefato durável, versionado e determinístico, que existe independentemente de qualquer sessão de agente.

O que é o SCL-AD, em uma frase

A palavra contexto é sobrecarregada, então o SCL-AD usa uma definição precisa. Contexto semântico é uma representação legível por máquina de propriedades que podem ser derivadas deterministicamente do código-fonte: informação de tipos resolvida, referências de símbolos com seus locais de definição e uso, grafos de dependência diretos e transitivos, grafos de chamada entre funções e serviços, interfaces de serviço e pontos de integração, configuração que afeta o comportamento em execução, cobertura de testes mapeada ao código de produção, fronteiras arquiteturais e as convenções que a base de código codifica. É distinto do contexto sintático (o texto bruto dos arquivos), do contexto conversacional (o chat atual) e do contexto probabilístico (recuperação por similaridade). É o que um engenheiro sênior competente diria a um recém-chegado que perguntasse como a base de código realmente funciona, e sua propriedade definidora é o determinismo: o mesmo código no mesmo commit produz o mesmo contexto toda vez que é computado.

Sobre essa definição, a Semantic Context Layer for Agentic DevOps é uma capacidade de plataforma que deriva, armazena, distribui e atualiza contexto semântico determinístico para agentes de codificação de IA, tratando contexto como um artefato de build versionado, não como um insumo efêmero de sessão. O

framework prescreve o que derivar do código, como derivá-lo por análise com precisão de compilador em vez de inferência, onde armazená-lo, como expô-lo por interfaces padrão como o Model Context Protocol, e como governá-lo em milhares de repositórios. Ele operacionaliza esse comportamento em quatro pilares de capacidade: Precompute, Target, Execute e Coordinate. E, deliberadamente, não é um produto: é uma metodologia neutra em relação a fornecedores, que uma organização pode implementar com ferramentas nativas do GitHub em torno do Copilot, com analisadores open-source, com plataformas comerciais, ou qualquer combinação, sem mudar como os agentes se comportam.

> DEFINIÇÃO
>
> SCL-AD em uma frase: uma capacidade de plataforma que deriva, armazena, distribui e atualiza contexto semântico determinístico para agentes de codificação de IA, tratando contexto como um artefato de build versionado, não como um insumo efêmero de sessão. Determinismo sobre inferência, reutilização sobre recomputação, neutralidade de fornecedor sobre lock-in, auditabilidade sobre mágica, atualização sobre obsolescência.

Referências

Fontes primárias para o padrão de falha, a evidência empírica e os conceitos sobre os quais o SCL-AD se apoia.

- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, a evidência empírica: desenvolvedores experientes levaram 19% mais tempo com ferramentas de IA em repositórios grandes e maduros.
- Moderne Prethink, Semantic Code Context for Coding Agents, articulação de referência da distinção entre determinismo e inferência.
- Moderne, Lossless Semantic Tree, exemplo concreto de modelagem de código com precisão de compilador.
- Anthropic, Effective context engineering for AI agents, orientação fundamental sobre contexto como preocupação de primeira classe.

- GitHub Copilot Documentation, a plataforma de referência para os padrões de integração de agentes deste playbook.
- Model Context Protocol Specification, o padrão aberto para troca 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, ARQUITETURA DE REFERÊNCIA

Arquitetura e inteligência de código: o substrato do SCL-AD.

O SCL-AD organiza o contexto semântico em quatro camadas com contratos explícitos. Este capítulo percorre a arquitetura de referência, os pilares de cada camada e a inteligência de código que fundamenta o GitHub Copilot: índices, símbolos e precompute.

Quatro camadas, uma direção de fluxo

O SCL-AD é estruturado em quatro camadas horizontais, cada uma com responsabilidade única, interfaces bem definidas e evolução e governança independentes. A Camada 1, Inteligência de Código, produz um modelo com precisão de compilador e preservação de formato de cada repositório no escopo, o substrato de todo o resto. A Camada 2, Receitas de Contexto, reúne extratores reutilizáveis que rodam contra esse modelo e produzem artefatos semânticos em Markdown, CSV, JSON e CALM. A Camada 3, o Registro de Contexto, armazena e distribui esses artefatos. A Camada 4, Integração de Agentes, conecta o registro ao GitHub Copilot, agentes customizados, clientes MCP e extensões de IDE.

A informação flui para cima: o modelo alimenta as receitas, as receitas produzem artefatos, o registro os serve, os agentes os consomem na execução. Os fluxos operacionais correm ao contrário: gatilhos de refresh, governança e controles de acesso descem como preocupações transversais. Na topologia de referência, repositórios no GitHub Enterprise ou Azure DevOps alimentam um pipeline no GitHub Actions ou Azure Pipelines, onde rodam as Camadas 1 e 2; o pipeline publica os artefatos no registro; e os runtimes de agente consomem via servidor MCP, arquivos estáticos ou API. Cada caixa é uma fronteira de deployment; cada seta, um contrato estável.

> DEFINIÇÃO
>
> Identidade, observabilidade, segurança e ciclo de vida atravessam todas as camadas. Regra âncora: um agente que não pode ler o repositório X não recebe contexto derivado dele. Propagação de identidade é obrigatória.

Os pilares de capacidade, camada por camada

A Camada 1 deve satisfazer quatro propriedades: precisão de compilador, o modelo concorda com o que o compilador produziria, tipos resolvidos, não inferidos; preservação de formato, espaços, comentários e trivia mantidos para referenciar os intervalos originais de código; linkage entre repositórios, uma biblioteca usada por dez serviços gera dez referências resolvidas, não uma string; e regeneração determinística por commit. A Camada 2 reconhece sete categorias de receitas: Inventário Estrutural, Inteligência de Dependências, Topologia de Serviços em CALM, Extração de Convenções, Mapeamento de Intenção de Testes, Superfície de Configuração e Retrato de Postura de Segurança; mesma versão de receita mais mesmo commit produz saída idêntica byte a byte.

A Camada 3 reconhece três padrões de armazenamento: co-localizado, artefatos em um diretório .context/ versionado com o código, adequado a pilotos; um registro centralizado, indexado para busca, refresh e controle de acesso em escala de plataforma; e registros federados sob restrições de soberania. Contra staleness, três gatilhos de refresh são obrigatórios: por commit, agendado e por evento. A Camada 4 suporta quatro modos de integração: injeção de arquivos estáticos, como o copilot-instructions.md que o GitHub Copilot lê ao abrir a sessão; um servidor MCP que o agent mode do Copilot consulta sob demanda; integração nativa; e híbrido, volume estático mais detalhe via MCP.

Dois contratos normativos

Dois contratos de interface são normativos. Todo artefato carrega um manifesto: id endereçado por conteúdo, categoria da receita, formato, URL do repositório e SHA do commit, nome e versão da receita, timestamp de geração, classificação como público, interno ou restrito, e assinaturas, a base de rastreabilidade, controle de

acesso e integridade. A API de descoberta exige cinco operações: listar repositórios com contexto, listar artefatos por repositório, filtrar por tipo, buscar por hash e pesquisar por atributo. O transporte é definido pela implementação; a semântica é normativa.

Inteligência de código: precisão de compilador ou erro composto

A maioria das ferramentas de análise de código não tem precisão de compilador: parseiam sintaxe e aplicam heurísticas, rápidas, mas assumem onde o compilador resolve fatos. Uma chamada service.process(item) em Java depende do tipo estático de service, do tipo em runtime se a chamada é virtual, do genérico de item, da sobrecarga de process e da injeção de dependência ativa. Um parser heurístico vê uma chamada de método; um modelo com precisão de compilador resolve um método específico em um tipo específico. Quando o GitHub Copilot pergunta o que chama esta função, essa lacuna separa uma resposta útil de uma enganosa, e as camadas superiores compõem a imprecisão.

Sete capacidades definem uma implementação válida, testáveis com fixtures. Resolução de símbolos liga cada referência à sua definição, respeitando escopo, visibilidade, imports e genéricos. Resolução de tipos atribui um tipo estático a cada expressão e, sob despacho polimórfico, o conjunto de tipos possíveis em runtime. Grafos de herança e implementação capturam cadeias de extends e implements. Grafos de chamada incluem os alvos possíveis do despacho dinâmico, não só o declarado. Resolução de dependências fixa versões, transitivas computadas. Mapeamento de configuração liga application.yaml, appsettings.json e .env ao código que os consome. Preservação de source range ancora cada fato em arquivo, linhas e colunas.

Estratégias e ferramentas por linguagem

Três estratégias cobrem o espaço. Compiladores nativos são o padrão-ouro: OpenRewrite LST ou Eclipse JDT para linguagens JVM, Roslyn Workspaces API para .NET, TypeScript Compiler API, Pyright com LibCST para Python, com resolução conservadora e o resto marcado como dinâmico, go/packages e go/types para Go, rust-analyzer para Rust, cada um com toolchain próprio. Plataformas unificadas

como o CodeQL, motor de análise semântica do GitHub, reduzem a complexidade operacional mas adicionam dependência de fornecedor. Tree-sitter com resolvedores de tipos não atinge a barra sozinho: é um fallback cujos artefatos carregam rótulo de fidelidade menor. Para COBOL e PL/I, plataformas comerciais de modernização são o caminho prático.

Índices entre repositórios, precompute em escala

A análise de um repositório é problema resolvido; a resolução entre repositórios é onde a maioria das implementações quebra. Uma função no repositório A chama uma classe no repositório B, que depende de uma interface no repositório C; resolver a cadeia exige os três com versões consistentes. Três abordagens: um workspace estilo monorepo com Nx, Bazel ou Turborepo, o mais simples e de maior fidelidade; um registro interno de pacotes, como Artifactory, Azure Artifacts ou GitHub Packages, com precisão limitada pela cadência de publicação; e a análise multi-repositório coordenada, com inventário, agendamento na ordem das dependências e etapa final de linkage, o padrão onde plataformas comerciais entregam mais valor.

Em escala empresarial, milhares de repositórios e milhões de linhas, três padrões mantêm o precompute viável. Análise incremental reanalisa só o que mudou. Cache por hash de conteúdo explora o determinismo: o mesmo commit produz o mesmo modelo, reutilizável entre execuções de receitas. Execução paralela distribui a análise em worker pools. Análise com precisão de compilador custa mais que a sintática, então restrinja o escopo a repositórios ativos, em produção e assistidos por agentes de IA. Em build versus buy: ferramentas nativas quando uma ou duas linguagens dominam, plataforma comercial para amplitude poliglota, ou híbrido; as Camadas 2 e acima contratam só a interface do modelo, e a escolha permanece reversível.

Referências

Fontes primárias da arquitetura e da inteligência de código do SCL-AD.

- Moderne Technology, Lossless Semantic Tree, exemplo de camada de Inteligência de Código.
- Model Context Protocol Specification, padrão da camada de integração de agentes.
- FINOS Common Architecture Language Model (CALM), formato padrão para artefatos de Topologia de Serviços.
- CodeQL Documentation, a plataforma de análise semântica do 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, RECEITAS DE CONTEXTO

Receitas de contexto: oito extratores que transformam a base de código em fatos confiáveis para o GitHub Copilot.

Uma receita é um extrator determinístico, escopado e declarativo sobre o modelo de inteligência de código. Este capítulo percorre as oito categorias canônicas: o que cada uma responde, o formato que emite e quando executá-la.

O que é uma receita

Uma receita é a unidade de engenharia de contexto no SCL-AD: uma função determinística do modelo de inteligência de código para um ou mais artefatos semânticos, parametrizada, versionada, testada e reutilizável entre repositórios. Três propriedades definem uma receita válida. Ela é determinística: com a mesma versão do modelo e os mesmos parâmetros, produz saída byte a byte idêntica, o que torna o registro de artefatos confiável. Ela é escopada: extrai uma classe específica de fato semântico, não uma varredura ampla; o escopo estreito é o que as torna componíveis. E ela é declarativa quanto às saídas: declara os tipos de artefato, o formato e o schema de cada saída, para que os consumidores, do registro ao agent mode do GitHub Copilot, validem o que recebem.

Toda receita compartilha a mesma anatomia, independentemente da linguagem. Um manifesto YAML ou JSON declara identidade e contrato: nome, versão, categoria, entradas como a versão exigida do modelo e padrões de inclusão e exclusão, e cada saída com formato e schema. A implementação percorre o modelo e emite os artefatos, seja como classe Recipe em Java ou Kotlin no OpenRewrite, analisador C# no Roslyn ou scripts Python e TypeScript sobre a API de consulta do modelo. Testes baseados em fixtures pareiam repositórios pequenos com saídas esperadas e cobrem caminho feliz, entradas vazias, casos de borda como dependências

circulares e código gerado, e erros. A documentação fecha o contrato; sem ela, não é seguro depender de uma receita.

> DEFINIÇÃO
>
> A âncora deste capítulo: o SCL-AD reconhece oito categorias canônicas de receitas, cada uma respondendo a uma pergunta distinta sobre a base de código. Os artefatos que elas emitem são o que o GitHub Copilot de fato lê: o padrão de auto-wiring mantém ponteiros para eles em .github/copilot-instructions.md, e assim agent mode, custom agents e ferramentas MCP fundamentam cada tarefa em fatos extraídos, não em exploração cara de arquivos brutos.

Receitas 1 a 4: o que o código é e como ele é escrito

As quatro primeiras categorias descrevem a forma estática da base de código: conteúdo, dependências, fronteiras de serviço e estilo. Implemente as oito onde a fundamentação deve ser abrangente e um subconjunto nos casos mais estreitos.

Structural Inventory

Responde: o que existe neste repositório? Emite Markdown mais CSV com módulos, pacotes e tipos de nível superior, descrições curtas, pontos de entrada e contagens de superfície pública. Execute-a primeiro em todo repositório; é a de maior alavancagem por token de esforço, porque substitui a exploração em que o agent mode do Copilot abre arquivo após arquivo só para aprender o layout.

Dependency Intelligence

Responde: de que ele depende e o que depende dele? Produz CSV mais um SBOM, SPDX como referência, cobrindo dependências diretas e transitivas com versão, licença e advisories. Use quando o Copilot cuida de upgrades, correções de vulnerabilidade ou migrações; a fonte a classifica como a mais acionável para agentes de migração e segurança.

Service Topology

Responde: quais serviços existem e como se relacionam? A saída é JSON no formato CALM, o schema de arquitetura da FINOS, descrevendo serviços, interfaces e relacionamentos. Use em parques multi-serviço, para que um custom agent do Copilot que propõe mudança entre serviços veja o raio de impacto real, não o inferido pelos nomes das pastas.

Convention Extraction

Responde: como o código é escrito nesta base? Emite Markdown com as convenções observadas: padrões de logging, tratamento de exceções, nomenclatura e o grau de consistência de cada uma. O framework a classifica como a de maior impacto na qualidade da saída dos agentes; use quando o objetivo é o Copilot gerar código que parece pertencer à base e passa na revisão de primeira.

Receitas 5 a 8: o que o código faz em runtime

As quatro categorias seguintes capturam comportamento e operação: testes, configuração, postura de segurança e deployment. Elas importam mais quando os agentes deixam de ler código e passam a mudá-lo.

Test Intent Mapping

Responde: o que é testado, como, e o que o teste está nos dizendo? Um CSV liga cada símbolo de produção aos símbolos de teste com rótulos de intenção como caminho feliz ou validação. Use sempre que o Copilot modificar comportamento; mostra quais testes protegem o código tocado e onde a intenção da cobertura tem lacunas.

Configuration Surface

Responde: qual configuração afeta o comportamento em runtime? Markdown mais JSON mapeiam cada chave para tipo, default, override de ambiente e os pontos exatos de código que a consomem. Use em tarefas que tocam settings ou feature flags; impede que um agente altere um valor sem ver todos os caminhos que o leem.

Security Posture Snapshot

Responde: quais fatos relevantes de segurança estão codificados no código? O Markdown cobre mecanismos de autenticação, primitivas criptográficas em uso, referências a segredos e superfícies expostas. Use antes de deixar agentes mexerem perto de auth ou criptografia, e como fundamentação compacta quando o Copilot apoia uma revisão de segurança.

Deployment Context

Responde: o que roda onde e sob quais restrições? Adicionada na v1.1.0 como oitava categoria canônica, emite Markdown mais JSON conectando Docker, Kubernetes e infraestrutura como código ao código de aplicação que afetam: imagens, limites de recursos, réplicas, endpoints de health, ConfigMaps e Secrets. É crítica para agentes que propõem mudanças com impacto operacional: sem ela, um agente pode propor uma mudança de configuração que compila sem erros mas falha em runtime por uma entrada ausente no ConfigMap.

Autoria, ciclo de vida e por onde começar

Oito diretrizes governam a autoria de receitas. Prefira escopo estreito a abrangência. Produza saída para humanos e máquinas: um artefato Markdown para revisão e, quando aplicável, CSV ou JSON para consumo programático. Respeite o determinismo: ordene as saídas de forma estável e mantenha timestamps fora do corpo dos artefatos. Embuta rastreabilidade: cada fato referencia seu trecho de origem no código. Falhe explicitamente diante de ambiguidade, emitindo entradas não resolvidas em vez de adivinhar. Versione separadamente receita, modelo e schema do artefato. Teste com fixtures reais e sanitizadas. E mantenha os artefatos compactos: um repositório de 100.000 linhas não deve produzir um inventário de 100.000 linhas, porque cada artefato disputa a janela de contexto de uma sessão de agente do Copilot. O ciclo de vida é o de software: autoria em repositório de receitas, teste obrigatório antes do release, publicação em registro sob versionamento semântico, deployment via configuração e depreciação como a de uma biblioteca.

Por onde começar: sem uma plataforma comercial, comece com três receitas, Structural Inventory, Convention Extraction e Dependency Intelligence, e expanda

para as oito nas Fases 2 e 3 do roadmap de adoção. Como orientação, o Moderne Prethink publica duas receitas iniciais: UpdatePrethinkContextNoAiStarter, análise puramente determinística que produz Dependency Intelligence, Test Intent Mapping, Service Topology e Structural Inventory, e UpdatePrethinkContextStarter, que adiciona grafo de conhecimento e resumos de teste gerados por IA via Bring Your Own Model. Ambas atualizam automaticamente os arquivos de configuração de agentes, incluindo .github/copilot-instructions.md, ligando os artefatos diretamente ao GitHub Copilot. Receitas customizadas, de mapeamentos regulatórios para PCI-DSS, HIPAA, LGPD e SOX a extração de APIs internas e multi-tenancy, seguem a mesma anatomia e ciclo de vida. O teste é simples: se a pergunta que a receita responde é determinística e reutilizável, é uma receita; se depende da formulação do prompt ou do estado da sessão, é um template de prompt e pertence à camada de integração de agentes.

Referências

Fontes para as oito categorias de receitas, a anatomia das receitas e as implementações de referência.

- OpenRewrite Recipe Documentation, o modelo de referência para o design de receitas.
- Moderne Prethink Documentation, exemplos concretos de saídas de receitas de contexto e as definições das receitas iniciais.
- Moderne Blog: Changing the AI context, referência para as categorias semânticas que informaram este framework.
- FINOS CALM Specification, o schema para artefatos de Service Topology.
- SPDX Specification, o formato SBOM de referência para as saídas 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, distribuição e integração com agentes: do artefato ao agente.

Artefatos semânticos só geram valor quando um agente os lê no momento certo. Este capítulo cobre o Context Registry que os armazena e versiona, as políticas de refresh e acesso que os mantêm confiáveis, e os padrões de integração que os entregam ao GitHub Copilot e ao ecossistema de agentes.

O Context Registry: onde vivem os artefatos semânticos

O Context Registry fica entre dois relógios: as receitas executam em cadência de build (um commit, um agendamento, um evento), enquanto os agentes consultam contexto em cadência de sessão, a cada interação do desenvolvedor. Cinco responsabilidades o definem. Armazena artefatos de forma durável: o que foi publicado permanece recuperável até ser arquivado. Descobre: agentes encontram artefatos por repositório, tipo, versão ou atributo de conteúdo sem conhecer o layout de armazenamento. Versiona: cada artefato recebe um identificador estável, endereçado por conteúdo. Impõe acesso: cada leitura é verificada contra as mesmas políticas que governam o código-fonte. E observa: cada leitura e escrita é registrada para reconstruir que contexto qualquer agente viu em qualquer momento.

A descoberta responde três perguntas. Por repositório: dado um repo, retornar todos os seus artefatos, com filtro por tipo ou commit. Por capacidade: dado um tipo de artefato, como topologia de serviços, retornar os repositórios que o possuem. Por conteúdo: consultas transversais, como quais repositórios usam certa biblioteca abaixo de certa versão. Cada artefato carrega ainda uma URI estável, com registry, repositório, commit, tipo e versão da receita: a referência canônica para citações, imutável durante a vida do artefato.

Três padrões de armazenamento, uma curva de maturidade

O armazenamento co-localizado mantém os artefatos dentro do repositório que descrevem, tipicamente em .context/ na raiz: a configuração mais simples, acesso idêntico ao do código, versionamento pelo Git, nenhuma configuração extra para os agentes, mas sem consultas entre repositórios nem governança centralizada. O registry centralizado é um serviço dedicado com APIs de escrita para receitas e de leitura para agentes: consultas cross-repo nativas e ciclo de vida central, ao custo de operar um novo serviço. O armazenamento federado liga vários registries por uma camada comum de descoberta, por unidade de negócio, região ou domínio de compliance: a resposta para residência de dados e indústrias reguladas. Comece co-localizado no piloto, centralize em escala de plataforma, federe apenas quando a soberania exigir.

Versionamento, refresh e acesso: mantendo o contexto confiável

Cada corpo de artefato recebe hash SHA-256, o identificador imutável: execuções que produzem bytes idênticos compartilham o mesmo hash, deduplicando o armazenamento. O manifesto carrega quatro versões explícitas, receita, schema, modelo e registry, de modo que uma mudança de schema nunca invalida artefatos antigos; muda apenas contra qual schema os agentes validam. Sobre os hashes, aliases tornam o registry utilizável: latest resolve para o artefato mais recente de um repositório e tipo, stable para o mais recente aprovado na validação, e commit mais SHA para um build específico.

Contexto obsoleto é o modo de falha que o registry deve impedir, então o refresh é multimodal. Por commit: reexecuta apenas as receitas afetadas pelos arquivos alterados, tipicamente em minutos. Agendado: diário para repositórios ativos, semanal para os estáveis, capturando o drift de dependências transitivas. Por evento: um aviso de segurança regenera a Dependency Intelligence, um upgrade de framework regenera a Convention Extraction. Cada tipo tem janela de frescor: 24 horas para Dependency Intelligence, Configuration Surface e Security Posture Snapshot; 7 dias para Structural Inventory, Service Topology e Test Intent Mapping; 30 dias para Convention Extraction. Artefatos obsoletos não são removidos automaticamente; a obsolescência é sinalizada para que os consumidores decidam.

DEFINIÇÃO

O controle de acesso segue dois princípios sem exceção. Propagação de identidade: se um solicitante não pode ler o repositório X no controle de versão, não pode ler os artefatos derivados dele no registry. Menor privilégio via rótulos de classificação que só podem ser elevados, nunca rebaixados. Cada leitura gera auditoria com identidade, URI, hash de conteúdo, resultado e contexto de chamada, retida por 12 a 36 meses em ambientes regulados.

Padrões de integração: como o contexto chega ao GitHub Copilot

Plataformas de agentes diferem em como consomem contexto, e a camada de integração precisa ser sem perdas, limitada e controlável: nada de truncamento ou paráfrase na fronteira, injeção estática dentro da janela de contexto, consultas dinâmicas dentro do orçamento de tool calls, e ajuste do que cada agente vê sem tocar nas receitas. O primeiro padrão, injeção estática, escreve artefatos selecionados em caminhos conhecidos: um diretório .context/ com inventário condensado, convenções, topologia e um manifesto de URIs para os artefatos completos. O alvo primário são as repository custom instructions do GitHub Copilot em .github/copilot-instructions.md, lidas no início da sessão em todas as superfícies do Copilot (IDE, GitHub.com, Copilot Chat). O pipeline insere um bloco gerenciado com ponteiros para .context/, preservando tudo fora do bloco; sem esse auto-wiring, a integração deriva em dias após a primeira edição manual. Nunca injete a Dependency Intelligence nem a Configuration Surface completas: são grandes e raramente necessárias por inteiro.

MCP para consultas profundas, híbrido para produção

O segundo padrão coloca um servidor Model Context Protocol na frente do registry, expondo ferramentas como list_repository_artifacts, get_artifact, search_dependencies, find_callers, describe_service e find_convention. Aplica-se quando a plataforma suporta MCP nativamente e a economia de tokens exige que só fatias relevantes cheguem ao modelo. Há duas topologias: um servidor local por desenvolvedor, sob sua identidade, ou um compartilhado com identidade propagada

via OAuth. Em ambos, as consultas devem responder em segundos de um dígito no pior caso, com cache por URI invalidado pelos eventos de refresh do registry; recuperação lenta é pior que nenhuma, porque o agente estoura o tempo e volta a inferir. Produção combina os padrões como distribuição híbrida: injeção estática de convenções e inventário resumido em cada repositório, blocos gerenciados, um servidor MCP para análise de impacto e consultas de dependências, e API direta para tooling em lote.

Mapeamento por agente, GitHub Copilot primeiro

O GitHub Copilot oferece a superfície mais completa para o SCL-AD. As repository custom instructions em copilot-instructions.md carregam a camada estática. Custom agents do Copilot recebem subconjuntos curados por papel: um agente de migração recebe Structural Inventory, Dependency Intelligence e Convention Extraction; um agente de review recebe convenções, Test Intent Mapping e Security Posture Snapshot. O Copilot Agent Mode e o Coding Agent, que executam tarefas de múltiplos passos, são os que mais se beneficiam do contexto dinâmico via MCP, com o servidor registrado na configuração do usuário ou da organização. Ao redor do Copilot, a convenção neutra AGENTS.md é o alvo estático primário recomendado, com arquivos específicos de plataforma gerados da mesma fonte: ferramentas do ecossistema como Claude Code com CLAUDE.md e Cursor com .cursorrules recebem os mesmos blocos gerenciados sem um segundo pipeline. Frameworks agênticos custom chamam a API do registry ou o servidor MCP diretamente, o caminho de maior fidelidade.

> DEFINIÇÃO
>
> Seis anti-padrões são explicitamente desencorajados: despejar o registry inteiro em arquivos de instrução, parsing ao vivo dentro do loop do agente, snapshots sem pipeline de refresh, endpoints MCP que borram a identidade do desenvolvedor, prompts mágicos para compensar contexto ausente e stores de contexto duplicados dentro da plataforma do agente. Contornar o registry, mesmo temporariamente, cria artefatos que ele não pode governar, atualizar nem auditar.

Referências

Fontes para o contrato do registry, os modos de distribuição e os padrões de integração.

- GitHub Copilot Documentation: repository custom instructions, GitHub, o alvo de injeção estática copilot-instructions.md.
- Model Context Protocol Specification, o padrão para distribuição dinâmica de contexto orientada a consultas.
- AGENTS.md Convention, a convenção emergente e neutra para arquivos de instrução de agentes.
- OCI Distribution Specification, fundação para implementações de registry baseadas em OCI.
- SLSA Provenance Specification, referência para proveniência e assinatura de artefatos.
- Moderne Blog: Changing the AI context, referência para o padrão de auto-wiring de arquivos de configuração de agentes.
- Anthropic Engineering: Effective context engineering for AI agents, orientação da indústria sobre estratégia 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, GOVERNANÇA E ADOÇÃO

Governança, adoção e métricas.

Os controles que tornam um registro de contexto seguro em ambientes regulados, o roteiro do piloto à plataforma e as métricas que separam um GitHub Copilot mais barato de um melhor.

Contexto não é dado neutro

Contexto semântico derivado de código-fonte codifica decisões de arquitetura, posturas de segurança, dependências e padrões proprietários. O registro que o armazena descreve como o software da organização realmente funciona, e cada sessão do GitHub Copilot que o consome herda esse conhecimento. Mal governado, gera três riscos. Vazamento: contexto de repositório sensível chega a agentes sem autorização, e ler o contexto do serviço de pagamentos é, na prática, ler a sua arquitetura. Integridade: artefatos adulterados fazem o Copilot raciocinar sobre base corrompida, e a confiança é irrecuperável depois de quebrada. Compliance: dados regulados armazenados ou transmitidos contra as regras. O framework trata governança como par da arquitetura, não camada por cima.

Cinco princípios, cinco papéis

Cinco princípios inegociáveis regem a operação. Identity-first: toda leitura, escrita e atualização vinculada a uma identidade autenticada; contas de serviço são identidades, 'o sistema' não é. Trace-by-default: todo artefato rastreável até a recipe, o código-fonte e o operador. Least exposure: agentes e humanos veem o mínimo que a tarefa exige, negação por padrão. Integridade verificável: todo artefato carrega assinatura ou hash de conteúdo. Propósito documentado: toda recipe e integração declara por que existe. Cinco papéis operam o modelo: dono da plataforma, stewards de recipes, donos de repositório, que decidem o que é exposto e não podem ser contornados, revisores de segurança e compliance e auditores, que leem logs, nunca artefatos.

Seis domínios de política

Seis domínios exigem decisões documentadas: escopo, quais repositórios entram, com código com impacto em produção como padrão; classificação, rótulos como público, interno, confidencial e restrito; acesso, com propagação de identidade como base; atualização, janelas de staleness e gatilhos; retenção, alinhada ao código-fonte e à regulação; e integração, quais agentes consomem o registro, revisados antes de chamá-lo.

Doze controles e a trilha de auditoria

Doze controles implementam as políticas em quatro famílias. Identidade e acesso: identidade autenticada em cada leitura, espelho de autorização das permissões do código-fonte e acesso privilegiado just-in-time. Integridade: assinatura de artefatos, endereçamento por conteúdo com detecção automática de adulteração e atestação de recipe ligando cada artefato à sua versão de recipe. Ciclo de vida: artefatos críticos vencidos bloqueiam o consumo até a atualização, os de repositórios arquivados são arquivados em até 7 dias e a exclusão é registrada, autorizada e notificada. Observabilidade: logs de leitura e escrita mais detecção de anomalias. A trilha de auditoria é append-only, retida 12 meses por padrão e de 36 a 84 meses em ambientes regulados. Incidentes têm três templates, integridade, divulgação e disponibilidade, todos com postmortem em 30 dias.

> DEFINIÇÃO
>
> A regra âncora é o espelho de autorização: ler um artefato exige leitura no repositório de origem. O GitHub Copilot vê o que o usuário já podia ver, nunca mais.

Cinco fases, da linha de base à plataforma

A fase 0, descoberta e linha de base, leva de 2 a 4 semanas: inventarie agentes, quais ferramentas, o GitHub Copilot em qual tier de licença e adoção por time; capture as métricas de base; selecione de 1 a 3 repositórios-piloto; e decida build, buy ou híbrido para a inteligência de código. A fase 1, o piloto, leva de 4 a 8 semanas: implemente duas ou três das sete recipes canônicas, começando por

inventário estrutural, extração de convenções e inteligência de dependências, armazene os artefatos no .context/ do repositório e injete-os no copilot-instructions.md para o Copilot ancorar cada requisição neles. Barra de saída: gasto de tokens por tarefa cai 30% ou mais, qualidade melhora e times relatam saída melhor.

A fase 2, expansão, leva de 3 a 6 meses: 10 a 50 repositórios diversos, registro centralizado, as recipes canônicas restantes, integração MCP para o modo agente do Copilot consultar o registro sob demanda e os cinco primeiros controles vivos em produção. A fase 3, plataforma, leva de 6 a 12 meses: todos os repositórios em escopo integrados, os doze controles operacionais, a camada na IDE, no code review e nos portais internos, e ROI demonstrado contra a linha de base da fase 0. A fase 4, otimização, é contínua: recipes sem consumo são descontinuadas, integrações evoluem com as plataformas e custos são ajustados.

Cinco riscos recorrentes de adoção

Sucesso de piloto que não escala, mitigado com diversidade realista. Retrofit de governança, o erro mais caro, mitigado habilitando identidade, auditoria e classificação desde a fase 1, mesmo em baixa fidelidade. Proliferação de recipes, mitigada nomeando stewards cedo e descontinuando com agressividade. Lock-in de fornecedor, mitigado impondo os contratos de interface mesmo com um único fornecedor. E economia de tokens sem qualidade, um Copilot mais barato que não é melhor, mitigada tratando qualidade como critério primário.

Seis métricas que provam o valor

O gasto de tokens por tarefa mede o que um agente consome numa categoria de tarefa, com meta de redução de 40 a 70% sobre a base até o fim da fase 2. A taxa de alucinação mede a fração de saídas com erro factual vindo de contexto incorreto, como o Copilot citar uma função inexistente ou uma API descontinuada, com redução mensurável até o fim da fase 1. A aderência a convenções mede quanto do código gerado cumpre as convenções documentadas na primeira geração, com meta de ao menos 30 pontos percentuais até o fim da fase 2.

O tempo até a primeira sugestão útil mede do prompt à primeira saída do Copilot aceita. A satisfação do desenvolvedor é uma pesquisa periódica no estilo NPS sobre

confiança e utilidade, em que a disposição de uso costuma já ser alta. O throughput de pull requests é terciário, atribuído com cautela. Duas métricas operacionais completam o conjunto: frescor do registro em 95% ou mais e cobertura de auditoria em 100%.

O modelo de ROI

O modelo tem três componentes. Economia direta: 500 desenvolvedores, 100 tarefas assistidas por desenvolvedor por mês, base de 50.000 tokens por tarefa, redução de 50% e custo de $0,005 por 1.000 tokens rendem cerca de $62.500 por mês, e esse é o piso. Qualidade: se a alucinação cai 50% e um erro de agente com impacto em produção custa 8 horas de remediação, a economia de retrabalho domina a de tokens por uma ordem de grandeza. A alavancagem de plataforma é o upside depois da fase 3, não a justificativa das fases 0 a 2. Custos: 2 a 4 engenheiros de plataforma até a fase 2, licenças, infraestrutura e governança. ROI positivo típico entre a saída da fase 2 e o primeiro trimestre da fase 3.

Referências

Fontes do modelo de governança, do roteiro de adoção e das métricas.

- NIST Cybersecurity Framework, NIST, a base da organização do catálogo de controles.
- SLSA Specification, o modelo de proveniência e atestação da assinatura de artefatos.
- ISO/IEC 42001, a norma de sistemas de gestão de IA para implantações agênticas.
- DORA State of DevOps Reports, a referência para medir impacto de plataforma de desenvolvimento.
- GitHub Octoverse 2024, GitHub, dados de adoção de ferramentas de IA em empresas.
- Effective context engineering for AI agents, Anthropic, o valor do contexto estruturado.

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 E REFERÊNCIA

Build vs buy, e o material de referência.

O SCL-AD é vendor-neutral por design: o framework define o que é contexto semântico e como deve se comportar, nunca o que comprar. Este capítulo compara os três caminhos de implementação sob a lente do GitHub Copilot, condensa o glossário essencial e consolida as referências.

Os três caminhos até a produção

Toda organização que adota o framework enfrenta a mesma decisão: construir as quatro camadas internamente, comprar uma plataforma comercial que já as implementa ou combinar as duas. O Caminho A, build, implementa tudo sobre os investimentos existentes, tipicamente GitHub Enterprise mais Microsoft Azure. Roslyn cobre .NET, OpenRewrite cobre a JVM, TypeScript Compiler API e Pyright cobrem web e Python, e o CodeQL adiciona grafos de chamadas entre linguagens. As recipes rodam como workflows reutilizáveis do GitHub Actions, o Context Registry vive em Azure Blob Storage mais Cosmos DB atrás do Azure API Management, e um job do GitHub Actions faz o auto-wiring de copilot-instructions.md e AGENTS.md a cada commit. A recompensa é soberania, integração nativa com o GitHub Copilot e customização sem limite; o preço é um time dedicado de três a cinco engenheiros de plataforma por nove a quinze meses, com cada recipe escrita e mantida internamente.

O Caminho B, buy, adota uma plataforma comercial que entrega a camada mais difícil em vez de pedir que você a construa. A Moderne é o exemplo mais proeminente: a Lossless Semantic Tree cobre JVM, JavaScript, TypeScript e COBOL, duas recipes iniciais do Prethink acompanham mais de 5.000 recipes do OpenRewrite, e o Moddy expõe tudo como um servidor MCP que GitHub Copilot, Claude Code e Cursor consomem diretamente. Um piloto produz artefatos reais em semanas, com um time de dois a três engenheiros focado em integração. Os custos: assinatura enterprise, dependência de fornecedor e customização limitada ao que a

plataforma expõe. O Caminho C, híbrido, é o padrão de produção mais comum: compre o que escala entre linguagens, inteligência de código e catálogo de recipes, e construa o que é singular da organização: recipes de domínio, registry, servidor MCP e governança sobre Entra ID e Sentinel.

Como decidir, e quanto custa

O framework de decisão faz seis perguntas: quantas linguagens dominam a produção, uma ou duas inclinam para build, seis ou mais para buy; quanta capacidade de engenharia de plataforma está dedicada; quão urgente é o time-to-value; quão estritos são os requisitos de soberania e customização; se migração é o caso de uso primário; e o nível de conforto com dependência de fornecedor. Contagem de linguagens e capacidade de engenharia dominam na prática. O quadro de custos: build custa de USD 1.5M a 3M em nove a quinze meses, depois 1.5 a 2 FTEs contínuos; buy custa de USD 400K a 800K em três a seis meses mais a assinatura, depois 0.5 a 1 FTE; o híbrido fica entre USD 800K e 1.5M em seis a nove meses mais assinatura, depois 1 a 1.5 FTEs. Em cinco anos os caminhos costumam convergir em custo total; o que muda é o fluxo de caixa, capex no build, opex no buy, e a capacidade que fica na organização.

O processo recomendado é curto e guiado por evidência. Discovery, de uma a duas semanas, documenta linguagens, repositórios, investimentos de plataforma e restrições de soberania. Avaliação, de três a seis semanas, roda um piloto pago do fornecedor em paralelo com um spike de viabilidade de build, um ou dois engenheiros por duas semanas produzindo uma fatia funcional de Code Intelligence para a linguagem dominante. Os artefatos são concretos: uma linha de base medida de consumo de tokens e taxa de alucinação, um protótipo de um pilar, tipicamente o Precompute, e uma projeção de custo fundamentada. A decisão vem em seguida, com evidência em vez de intuição, e o caminho escolhido entra na Fase 1 do roadmap de adoção.

> DEFINIÇÃO
>
> A decisão é reversível. Quem começa no build pode pivotar para buy se a capacidade de engenharia se mostrar insuficiente, e quem começa no buy pode pivotar para o híbrido se surgirem requisitos de customização que a plataforma não atende. Os contratos de interface do framework tornam esses pivôs administráveis, e é exatamente por isso que o SCL-AD permanece vendor-neutral por design.

O glossário essencial

Um punhado de termos sustenta o framework, cada um com um único significado nos documentos do SCL-AD. Contexto semântico é uma representação legível por máquina de propriedades derivadas deterministicamente do código-fonte, distinta de contexto conversacional ou probabilístico. Um modelo de inteligência de código é uma representação consultável e fiel ao compilador de um repositório; a Lossless Semantic Tree, originada pela Moderne, é uma implementação possível, não um requisito. Uma recipe é um extrator reutilizável e parametrizado que roda contra esse modelo e produz artefatos, a unidade de engenharia de contexto; o framework reconhece oito categorias canônicas, de Structural Inventory a Deployment Context. O Context Registry armazena artefatos versionados e endereçados por conteúdo e os expõe por uma API de descoberta, com janelas de staleness e refresh por commits, agendas ou eventos.

Do lado do agente, quatro pilares de capacidade nomeiam o que a camada faz: Precompute deriva conhecimento estruturado com antecedência, Target encontra o código certo por índices pré-computados, Execute aplica mudanças deterministicamente em muitos repositórios e Coordinate compartilha contexto de mudança entre agentes, derrubando duplicação e gasto de tokens. O auto-wiring mantém os arquivos de configuração de agente, copilot-instructions.md, AGENTS.md e CLAUDE.md, apontados para os artefatos atuais como parte do pipeline de recipes, para que o GitHub Copilot nunca leia um ponteiro obsoleto. Injeção de arquivo estático e servidor MCP são os dois padrões principais de integração. BYOM permite chamar um modelo aprovado para enriquecer artefatos com resumos ou grafos de conhecimento, opcional e isolado do núcleo

determinístico. E alucinação é uma saída que afirma um fato ausente do contexto real; determinismo, mesma entrada, mesma saída, é a propriedade que o framework mobiliza contra ela.

Referências

As fontes fundacionais do posicionamento, do glossário e do próprio framework.

- Moderne Prethink, Semantic Code Context for Coding Agents, a articulação original do contexto semântico determinístico.
- Moderne Agent Capabilities, a fonte dos quatro pilares de capacidade.
- METR, Measuring AI Effects on Experienced Developer Productivity, evidência empírica de que o problema de contexto é mensurável.
- Effective context engineering for AI agents, Anthropic, contexto como preocupação de primeira classe.
- The architect's guide to the AI-native developer platform, GitHub, a perspectiva enterprise por trás do caminho de build.
- Model Context Protocol Specification, o padrão aberto por trás da integração via MCP.
- OpenRewrite Documentation, motor open-source de LST e recipes, disponível aos três caminhos.

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