CONFIGURATION CONTEXT ARCHITECTURE · PLAYBOOK

Um arquivo cobre 10%. Dez camadas em três tiers tornam o Copilot auditável.

A maioria dos times ativa o GitHub Copilot em agent mode, escreve um único .github/copilot-instructions.md e para. Esse arquivo cobre cerca de 10% da superfície de configuração que o VS Code expõe para o Copilot. Este playbook mapeia os outros 90%: dez camadas de configuração agrupadas em três tiers de memória, hot, warm e cold, para que cada regra carregue quando é relevante, nada seja truncado, e o comportamento do Copilot se torne auditável, versionado e com dono, em vez de tribal.

AUTORA

Paula Silva

CARGO

AI-Native Software Engineer

CONTATO

in/paulanunes

TEMPO DE LEITURA

~ 45 minutos, fim a fim

10
CAMADAS DE CONFIGURAÇÃO

3
TIERS DE MEMÓRIA

6
CAPÍTULOS

10%

SUPERFÍCIE QUE UM ARQUIVO COBRE

CAPÍTULOS

Chapter 00

Dez camadas, não um arquivo

A armadilha dos 10%, o estudo empírico de 2.303 arquivos e o ponto cego de segurança de 14,5%, e por que manifestos de arquivo único colapsam além de 3.000 tokens em uma base de 108.000 linhas e 283 sessões.

Problema →

Chapter 01

O mapa: três tiers em dez superfícies

As dez camadas de configuração na ordem em que o VS Code as carrega, o modelo de memória hot, warm e cold de Vasilopoulos 2026, e a regra única para decidir onde cada regra sua pertence.

Mapa →

Chapter 02

A camada de workspace: o que é sempre verdade

O copilot-instructions.md mantido abaixo de 1.000 tokens, as seções de Segurança e Performance que a maioria dos times pula, e o glob applyTo que mantém o arquivo hot curto em quatro padrões de escopo.

Workspace →

Chapter 03

A camada de autoria: o que você invoca

Agents, chat modes, skills e prompt files, escolhidos pelo modo como cada um é acionado e não pelo conteúdo, com escopo de ferramentas de menor privilégio e seleção de modelo por agent.

Autoria →

Chapter 04

Governança de runtime: o que roda ao redor do agente

Hooks como guardrails determinísticos, conexões MCP e plugins para dados frescos e bundles versionados, precedência de settings, e a ordem de montagem de nove passos que diz por que uma regra foi ignorada.

Runtime →

Chapter 05

Do zero à baseline em uma semana

Um commit por dia, do copilot-instructions.md a um plugin compartilhado, expectativas honestas sobre o que a hierarquia fará e não fará, e governança em vez de prompt engineering.

Adoção →

DICAS DE LEITURA

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

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 O PROBLEMA

Dez camadas, não um arquivo.

A maioria dos times ativa o GitHub Copilot em agent mode, escreve um único .github/copilot-instructions.md e considera a configuração pronta. Esse único arquivo cobre cerca de 10% da superfície de configuração que o VS Code expõe para o Copilot. Os outros 90% ficam vazios. Em um repositório pequeno o gap é invisível. Em um parque enterprise de milhares de repositórios, ele vira o motivo pelo qual o comportamento do Copilot permanece tribal em vez de auditável. Este capítulo nomeia a falha, apoia-a em dois estudos empíricos e mostra por que um único arquivo não sustenta uma base de código enterprise. A correção não é um arquivo maior. É uma superfície em camadas.

A armadilha dos 10%

O padrão se repete entre os times. Alguém ativa o GitHub Copilot em agent mode, escreve um único .github/copilot-instructions.md e para. O arquivo lista a stack, um punhado de convenções, talvez o comando de build. Lê bem em uma demo. Na prática, cobre cerca de 10% da superfície de configuração que o VS Code expõe para o Copilot. Os 90% restantes ficam sem configuração, e superfície não configurada é onde o comportamento drifta. Um repositório pequeno esconde o custo. Um monorepo de dezenas de pacotes, replicado em centenas de repositórios, não.

Os 90% ausentes não são uma coisa só. São mais nove superfícies: instruções com escopo usando globs applyTo, agents, chat modes, skills, prompt files, hooks, conexões MCP, plugins e settings do VS Code. Cada uma responde a uma pergunta que o arquivo único não responde. Quais regras se aplicam apenas à pasta de pagamentos. Qual workflow o time roda a cada release. Qual comando o Copilot nunca pode executar contra produção. Quando essas perguntas não têm resposta escrita, o modelo preenche o vazio com um chute, e o chute permanece invisível até um revisor pegá-lo em um pull request.

O gap empírico: 2.303 arquivos de contexto e o ponto cego de 14,5%

O gap é mensurável. Um estudo de 2025 analisou 2.303 arquivos de contexto reais em 1.925 repositórios (Chatlatanagulchai et al., arXiv 2511.12884). O padrão é consistente. Os times escrevem o que torna o agente funcional. Comandos de build e execução aparecem em 62,3% dos arquivos. Detalhes de implementação aparecem em 69,9%. Arquitetura aparece em 67,7%. São os fatos que colocam um agente para produzir código já na primeira tentativa, então são escritos primeiro.

O que os times pulam é o conjunto de regras que mantém esse código seguro. Guardrails de segurança aparecem em apenas 14,5% dos arquivos. Regras de performance aparecem em outros 14,5%. Tratamento de erros e comportamento degradado são raros. O resultado é um agente que escreve código produtivo, mas não previsível. O problema não é o modelo. O Copilot deixa você escolher Claude Fable 5, Opus 4.8 ou GPT-5.x no mesmo model picker, e nenhum deles consegue aplicar uma regra que nunca foi escrita. O ponto cego de segurança de 14,5% é um gap de configuração, não um gap de modelo. Trocar de modelo não fecha esse gap. Escrever as regras que faltam, sim.

Por que manifestos de arquivo único colapsam

O argumento de escala tem um estudo de caso por trás. Um projeto documentado construiu 108.000 linhas de C# ao longo de 70 dias e 283 sessões do Copilot (Vasilopoulos, arXiv 2602.20478). Ele concluiu que manifestos de arquivo único colapsam cedo. Um .github/copilot-instructions.md plano que ultrapassa cerca de 3.000 tokens é truncado pelo modelo. Tudo depois do corte é descartado sem aviso. O time continua adicionando regras. O agente continua lendo menos delas. As instruções parecem completas no editor e já estão incompletas quando o modelo as vê.

A correção nesse projeto foi uma estrutura em tiers: hot memory carregada em cada requisição, warm memory carregada por trigger, cold memory consultada via MCP. Essa estrutura manteve uma razão conhecimento-código de 24,2% em um monorepo de 14 pacotes ao longo das 283 sessões. Organizar em tiers também protege a conta. O Copilot em agent mode mede o uso em GitHub AI Credits, com

um crédito equivalente a um centavo de dólar. Uma instrução truncada ainda consome tokens de entrada no caminho até o modelo. Pagar para enviar contexto que o modelo depois descarta é o pior dos dois mundos: você é cobrado pela regra e o agente nunca a aplica. Organizar em tiers transforma esse desperdício em reuso: contexto computado uma vez, carregado só onde ele se justifica.

> DEFINIÇÃO
>
> Um arquivo é ferramenta de protótipo. Uma base de código enterprise precisa de dez camadas de configuração em três tiers, hot, warm e cold, para que cada regra carregue quando é relevante e nenhuma regra seja truncada por falta de espaço.

Três sintomas de uma superfície subutilizada

Você não precisa de um estudo para detectar o problema. Três sintomas aparecem no trabalho do dia a dia. Instruções duplicadas: a mesma regra vive em três arquivos, e o Copilot segue aquela que por acaso corresponde ao arquivo aberto. Drift silencioso: o modelo gera código útil que ignora uma convenção do time, porque a convenção existe apenas na cabeça de alguém e nunca em um arquivo que o modelo possa ler. Conhecimento tribal: um prompt forte para construir um endpoint de API funciona, mas apenas uma pessoa no time sabe que ele existe.

Cada sintoma aponta para uma camada específica. Instruções duplicadas indicam que a regra pertence a um arquivo com escopo em .github/instructions com um glob applyTo, não copiada para o manifesto raiz. Instruções com escopo se mergeiam com o arquivo de workspace, não o substituem, então a raiz permanece curta enquanto a regra carrega apenas onde se aplica. Drift silencioso indica que uma convenção nunca foi codificada no tier hot. Conhecimento tribal indica que um workflow deveria ser uma skill ou um prompt file, endereçável por todo o time. A correção é estrutural, não conversacional. Mapeie cada regra para a camada que a governa. O objetivo não é mais configuração. O objetivo é configuração que seja auditável, versionada e inequívoca sobre quem é responsável por cada camada.

Referências

Fontes primárias para a superfície de dez camadas, o gap empírico e o estudo de caso por trás da falha do arquivo único.

- Agent READMEs: An Empirical Study of Context Files for Agentic Coding, o gap empírico: 2.303 arquivos de contexto, com guardrails de segurança presentes em apenas 14,5% deles.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, o estudo de caso de 108.000 linhas e 283 sessões sobre por que manifestos de arquivo único colapsam.
- Customize AI in Visual Studio Code, a referência para as dez superfícies de configuração mapeadas neste capítulo.
- GitHub Copilot Documentation, a plataforma por trás do agent mode, do copilot-instructions.md e do model picker.
- Model Context Protocol, o padrão aberto que leva o tier cold de contexto até o agente.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 O MAPA

Três tiers em dez superfícies: o mapa de cada camada de configuração do GitHub Copilot.

Cada resposta do GitHub Copilot no VS Code é montada a partir de dez superfícies de configuração, não de um único arquivo de instruções. Essas dez superfícies se agrupam em três tiers de memória, e os três tiers respondem a três perguntas sobre a base de código. Este capítulo é o mapa. Ele nomeia as dez camadas na ordem em que o Copilot as carrega, agrupa-as nos tiers hot, warm e cold documentados por Vasilopoulos 2026, e entrega a regra única que decide onde cada regra sua pertence.

As dez camadas de configuração, na ordem em que o Copilot as carrega

Cada resposta do GitHub Copilot é construída a partir de uma janela de contexto em camadas, e há dez superfícies que a alimentam. O arquivo de instruções do workspace em .github/copilot-instructions.md carrega sempre. As instruções com escopo em .github/instructions/*.instructions.md carregam por match de caminho contra um glob applyTo. Agents carregam quando invocados por nome, chat modes quando selecionados, skills por match de descrição e prompt files por slash command. Hooks disparam em eventos de ciclo de vida. Conexões MCP se anexam por sessão. Plugins carregam quando instalados. As settings do VS Code se aplicam sempre. A maioria dos times escreve uma dessas, o copilot-instructions.md, e para por aí, o que cobre cerca de dez por cento da superfície. Os outros noventa por cento ficam sem configuração, e é por isso que o comportamento do Copilot permanece tribal em vez de auditável.

A ordem não é cosmética, porque o Copilot monta a janela de contexto de forma determinística. O VS Code resolve as settings primeiro, default depois usuário depois workspace, com o escopo mais específico vencendo. Ele carrega o copilot-

instructions.md incondicionalmente. Ele avalia os globs applyTo contra os arquivos abertos e faz merge de cada conjunto de instruções que corresponde. Ele anexa o primitivo de autoria ativo, seja um agent, um chat mode, uma skill ou um prompt file. Ele executa os hooks on-prompt-submit, que podem negar, mutar ou anotar a requisição. Ele descobre as ferramentas expostas pelos servidores MCP conectados. Então despacha o system prompt montado, o catálogo de ferramentas e a mensagem do usuário para o modelo selecionado no picker do Copilot. As camadas posteriores refinam as anteriores; não podem contradizê-las. Uma skill não ignora uma política de organização, e um prompt não sobrescreve uma setting de administrador enterprise.

Depure de cima para baixo

Essa ordem também é uma ferramenta de depuração. Quando o Copilot ignora uma regra, a falha está quase sempre em um de três lugares: o escopo de settings errado, um glob applyTo que não corresponde ao arquivo que você tem aberto, ou um hook que bloqueou ou mutou a instrução antes de ela chegar ao modelo. Percorra a ordem de montagem de cima para baixo antes de concluir que o modelo está com problema. A sequência é reproduzível, então o bug é localizável.

Hot, warm, cold: o modelo de memória em três tiers de Vasilopoulos 2026

As dez superfícies se agrupam em três tiers de memória, uma taxonomia documentada por Vasilopoulos (2026, arXiv 2602.20478). Aquele estudo de caso construiu e instrumentou um sistema em C# de 108.000 linhas ao longo de 70 dias e 283 sessões em um monorepo de 14 pacotes, e manteve uma razão conhecimento-código de 24,2 por cento. O achado que importa aqui é que um manifesto de arquivo único colapsa bem antes dessa escala. Um copilot-instructions.md plano que ultrapassa cerca de 3.000 tokens é truncado pelo modelo, e tudo após o corte vai driftando silenciosamente. A correção não é um arquivo mais longo. A correção são tiers.

A memória hot são as camadas 01 e 02 mais as settings do VS Code relacionadas ao Copilot: sempre carregadas, em cada requisição. A memória warm são as camadas 03 a 07, as instruções com escopo, agents, chat modes, skills, prompt files e hooks,

carregadas sob demanda quando um gatilho dispara por nome, seleção, glob, match de descrição, slash command ou evento de ciclo de vida. A memória cold são as camadas 08 e 09, as conexões MCP e plugins, consultadas explicitamente quando o agente busca dados que vivem fora do repositório. A memória hot viaja em cada turno, então cada token nela é pago em cada requisição. Desde 1 de junho de 2026, o GitHub Copilot cobra o uso premium em GitHub AI Credits, um credit equivalente a um centavo de dólar, o que transforma um tier hot acima do peso em uma despesa recorrente, não em um custo único. As memórias warm e cold são baratas porque só carregam quando o trabalho as exige.

> DEFINIÇÃO
>
> A memória hot é cara porque carrega em cada requisição e seus tokens são cobrados a cada turno. As memórias warm e cold são baratas porque só carregam quando acionadas. O erro que a maioria dos times comete é colocar tudo no hot.

Workspace, autoria, runtime: as três responsabilidades que cada tier serve

Os três tiers também se mapeiam em três responsabilidades, e nomeá-las torna a propriedade óbvia. A camada de workspace responde à pergunta "o que é sempre verdade sobre esta base de código." É o copilot-instructions.md mais os arquivos .github/instructions/*.instructions.md com escopo, guiados por globs applyTo. Juntos são o contrato do time, e carregam em cada requisição. Um subconjunto com escopo de caminho refina as regras globais sem sobrescrevê-las; quando vários arquivos correspondem ao mesmo caminho, eles fazem merge em vez de substituir. É assim que uma regra de React com escopo em src/components e as regras globais do workspace carregam lado a lado.

A camada de autoria responde à pergunta "o que eu invoco." Agents, chat modes, skills e prompt files são conhecimento reutilizável empacotado como artefatos e vinculado a uma tarefa, não a um arquivo. A regra de seleção é simples: escolha o primitivo pelo mecanismo de invocação, não pelo conteúdo. Um agent é invocado por nome para uma classe inteira de tarefas, como revisão ou teste. Um chat mode

troca a persona pela duração de uma conversa. Uma skill carrega automaticamente quando a requisição do usuário corresponde à sua descrição. Um prompt file executa por trás de um slash command que o usuário digita de propósito.

A camada de runtime responde à pergunta "o que roda ao redor do agente." Hooks interceptam cada chamada de ferramenta, antes e depois, para bloquear, registrar ou reescrever. Conexões MCP estendem o alcance do agente para sistemas externos, bancos de dados, issue trackers, observabilidade, APIs internas, por meio de um único protocolo descoberto no início da sessão. Plugins empacotam capacidades de terceiros por trás de um único manifesto. As settings do VS Code ajustam o comportamento determinístico do modelo. Todas as quatro superfícies são de responsabilidade do platform team, configuradas uma vez e mudadas raramente. Editar a configuração de runtime por feature ou por sprint é o antipadrão que esta camada existe para prevenir.

A regra: promova para hot apenas o que é verdade em cada linha do workspace

Uma regra decide onde uma regra sua pertence. Reserve o tier hot para o que é verdade em cada linha de código do workspace: convenções de nomenclatura, política de segurança, baselines de performance. Se uma regra se aplica a apenas uma tarefa, um tipo de arquivo ou um diretório, ela não pertence ao hot; pertence ao tier warm como uma instrução com escopo, um agent, uma skill ou um prompt file. Se ela configura o ambiente ao redor do agente em vez do conteúdo do prompt, pertence ao tier cold. O corolário é operacional: aplique a política no escopo externo e não a policie por usuário. Trave o contrato no nível da organização ou do workspace, e deixe os escopos internos adicionar preferências ergonômicas que não podem comprometê-lo.

O gap empírico é onde essa regra prova seu valor. Um estudo de 2.303 arquivos de contexto reais em 1.925 repositórios (Chatlatanagulchai et al., 2025, arXiv 2511.12884) constatou que os times escrevem o que torna o agente funcional e pulam o que o torna seguro. Comandos de build e execução aparecem em 62,3 por cento dos arquivos, detalhes de implementação em 69,9 por cento e arquitetura em 67,7 por cento, mas as guardas de segurança aparecem em apenas 14,5 por cento e as regras de performance em outros 14,5 por cento. O resultado é um agente que

produz código produtivo, mas não necessariamente seguro ou previsível. Segurança e performance são exatamente as regras que valem em cada linha do workspace, então pertencem ao tier hot pela regra acima, e a ausência delas ali é o gap mais comum e mais caro que um rollout de Copilot deixa aberto.

> DEFINIÇÃO
>
> Promova para hot apenas o que é verdade em cada linha do workspace: convenções, política de segurança, baselines de performance. Tudo que é específico de tarefa fica warm. Tudo que configura o runtime fica cold. Aplique no escopo externo; não policie por usuário.

Referências

Fontes primárias para o modelo de memória em três tiers, o gap empírico nos arquivos de contexto reais e as superfícies de configuração que este capítulo mapeia.

- Vasilopoulos, Codified Context: Infrastructure for AI Agents in a Complex Codebase, a fonte do modelo de memória hot, warm, cold: um sistema de 108.000 linhas ao longo de 283 sessões a uma razão conhecimento-código de 24,2 por cento.
- Chatlatanagulchai et al., Agent READMEs: An Empirical Study of Context Files for Agentic Coding, o gap empírico em 2.303 arquivos de contexto: 62,3 por cento cobrem build e execução, apenas 14,5 por cento cobrem segurança.
- Customize AI in Visual Studio Code, a referência de como as superfícies de configuração carregam e como os escopos de settings se resolvem.
- GitHub Copilot Documentation, a plataforma de referência para o comportamento de agent mode, arquivo de instruções e MCP deste capítulo.
- Model Context Protocol Specification, o padrão aberto por trás das conexões MCP do tier cold.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

02 CAMADA DE WORKSPACE

O que é sempre verdade: os dois arquivos que carregam em cada requisição.

A camada de workspace responde a uma pergunta: o que é sempre verdade sobre esta base de código. É a fundação que todo projeto com GitHub Copilot precisa, e são dois arquivos com um trabalho cada. O primeiro é o copilot-instructions.md, carregado em cada requisição. O segundo é um conjunto de arquivos de instrução com escopo que carregam apenas onde se aplicam. Acerte esta camada e o Copilot faz o onboarding do próximo colaborador por você. Erre e cada revisão pega os mesmos erros evitáveis.

copilot-instructions.md: o arquivo que o modelo lê toda vez

O copilot-instructions.md fica em .github/copilot-instructions.md. Ele carrega em cada requisição do GitHub Copilot: chat, edits e agent mode. É Markdown puro, sem frontmatter, e existe exatamente um arquivo por workspace. Sem cascata, sem uma segunda cópia que o sobrescreva. É o arquivo de maior impacto que você pode escrever para o Copilot, porque é o único conteúdo que chega ao modelo em cada linha de código.

A restrição é o tamanho. Mantenha-o abaixo de 1.000 tokens. Passado esse orçamento, o Copilot resume o arquivo e você perde o determinismo: o modelo lê uma versão comprimida e você não consegue mais prever quais regras sobrevivem. Um arquivo plano que ultrapassa 3.000 tokens é truncado de imediato, e tudo após o corte drifta em silêncio. Reserve este arquivo para o que é verdade em todo lugar: o stack, as convenções, a política de segurança, a baseline de performance. Se uma regra se aplica a apenas uma tarefa ou uma pasta, ela não pertence aqui.

Por que um único arquivo para de escalar

Um projeto documentado em C# com 108.000 linhas, construído ao longo de 70 dias e 283 sessões em um monorepo de 14 pacotes, concluiu que manifestos de arquivo único colapsam cedo (Vasilopoulos, 2026). O projeto manteve uma razão conhecimento-código de 24,2% apenas depois de migrar para um modelo de memória em tiers. A lição não é que o copilot-instructions.md seja fraco. A lição é que ele tem um trabalho e um orçamento, e tudo além desse orçamento pertence às camadas abaixo dele.

Um exemplo real, com as seções de Segurança e Performance que a maioria dos times pula

É assim que o arquivo fica quando é curto e específico. Ele nomeia o stack em uma linha: Next.js 14 App Router, TypeScript em modo estrito, Tailwind, Drizzle ORM sobre PostgreSQL, Vitest. Ele declara convenções como regras que o modelo pode seguir: arquivos em kebab-case, componentes React em PascalCase, testes co-localizados ao lado do código que cobrem, sem imports diretos de node_modules. Ele lista os comandos-chave para dev, build, lint e teste. Nada disso é incomum.

O incomum são as duas seções que a maioria dos times deixa de fora. Uma seção de Segurança: nunca registrar corpos de requisição ou headers de autenticação, validar cada entrada externa antes de usá-la e verificar a sessão antes de qualquer escrita no banco. Uma seção de Performance: prefira server components, mantenha payloads pequenos e pagine listas longas. São essas as regras que tornam a saída do Copilot segura e previsível, não apenas produtiva. Custam quatro linhas cada e fecham a lacuna que produz a maioria das sugestões rejeitadas.

Uma pesquisa com 2.303 arquivos de contexto reais em 1.925 repositórios mediu exatamente essa lacuna (Chatlatanagulchai et al., 2025). Comandos de build e execução apareceram em 62,3% dos arquivos, detalhes de implementação em 69,9%, arquitetura em 67,7%. Regras de segurança apareceram em apenas 14,5%. Regras de performance em outros 14,5%. Tratamento de erros era ainda mais raro. O modo de falha padrão de uma adoção do Copilot não é a ausência de um arquivo de contexto. É um arquivo de contexto que torna o agente produtivo e o deixa inseguro.

DEFINIÇÃO

As duas seções que fecham a lacuna são as duas que a maioria dos times nunca escreve. Apenas 14,5% dos arquivos de contexto reais incluem regras de segurança, e apenas 14,5% incluem regras de performance. Adicionar um bloco de Segurança e um bloco de Performance ao copilot-instructions.md é a correção mais barata disponível.

Instruções com escopo: o glob applyTo que mantém o arquivo hot curto

O arquivo de workspace fica dentro do orçamento ao mover a orientação especializada para fora dele. As instruções com escopo ficam em .github/instructions/*.instructions.md. Cada arquivo carrega um glob applyTo no seu frontmatter, e o Copilot carrega o arquivo apenas quando um arquivo aberto corresponde a esse glob. Um arquivo com escopo applyTo: "src/components/**/*.tsx" chega ao modelo quando alguém edita um componente e permanece em silêncio no resto do tempo. As regras detalhadas existem, mas só custam tokens quando são relevantes.

A regra que torna isso seguro é que as instruções se somam, elas não se sobrescrevem. Vários arquivos de instrução podem corresponder ao mesmo caminho, e todos carregam junto com o arquivo de workspace. Nada substitui o contrato global; os arquivos com escopo o refinam. É assim que você mantém o copilot-instructions.md pequeno sem perder o detalhe que um stack específico exige. As regras de React ficam em um arquivo, as de SQL em outro, e o arquivo hot só carrega o que é verdade em todo lugar.

Quando uma regra com escopo não carrega

Quando o Copilot ignora uma regra com escopo, a causa quase sempre é o glob. Verifique se o padrão applyTo realmente corresponde ao arquivo que você tem aberto antes de concluir que o modelo está com problema. Um padrão que aponta para src/components não dispara em um arquivo dentro de app/components, e uma

regra escrita para a extensão errada nunca chega ao modelo. Percorra o glob antes de discutir com o agente.

Quatro padrões de escopo, incluindo o padrão de superfície de risco que a maioria ignora

Quatro padrões cobrem a maioria dos repositórios. O primeiro dá escopo por camada de framework: um arquivo para src/components, um para src/server, um para src/db, cada um carregando as regras que pertencem àquela camada. O segundo dá escopo por tipo de arquivo: convenções de teste em **/*.test.ts, estilo de documentação em **/*.md, regras de revisão de query em **/*.sql. O terceiro dá escopo por domínio ou bounded context: um arquivo por pacote em um monorepo, de modo que packages/billing e packages/auth ganham cada um o seu contrato.

O quarto padrão é o que a maioria dos times nunca escreve, e é o de maior valor. Dê escopo por superfície de risco. Coloque as regras de PCI em **/payments/**. Coloque as regras de endpoint privilegiado em **/api/admin/**. Coloque as regras de mudança destrutiva em migrations/**. Esses são os caminhos onde uma sugestão errada custa mais caro, e eles mapeiam exatamente na lacuna de segurança de 14,5% que a pesquisa mediu. O arquivo de workspace não consegue carregar esse nível de detalhe sem estourar o orçamento. Um arquivo de instrução por superfície de risco consegue, e ele só carrega quando alguém toca no código que o justifica.

> DEFINIÇÃO
>
> O padrão de superfície de risco é a decisão de escopo que a maioria dos times deixa passar. Dê escopo aos guardrails detalhados nos caminhos onde uma sugestão errada é cara: pagamentos, endpoints de admin, migrations. É a mesma lacuna de segurança que a pesquisa mediu, resolvida no nível do arquivo em vez do nível do prompt.

Referências

Fontes primárias para a camada de workspace, a evidência empírica e as superfícies de configuração sobre as quais este capítulo se apoia.

- Customize AI in Visual Studio Code, a documentação de referência para o copilot-instructions.md, os arquivos de instrução com escopo e o glob applyTo.
- An Empirical Study of Context Files for Agentic Coding, a fonte da lacuna de segurança e performance de 14,5% em 2.303 arquivos de contexto reais.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, o estudo de caso de 108.000 linhas sobre por que manifestos de arquivo único colapsam e por que a memória em tiers é necessária.
- GitHub Copilot Documentation, a plataforma de referência para o agent mode e os padrões de configuração deste playbook.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

03 CAMADA DE AUTORIA

O que você invoca: agents, chat modes, skills e prompt files, e como escolher entre eles.

A camada de autoria é onde você empacota comportamento reutilizável como artefatos. Ela contém quatro primitivos: agents, chat modes, skills e prompt files. Os quatro ficam no tier warm, o que significa que o GitHub Copilot os carrega apenas quando algo os aciona, nunca em toda requisição. O erro que os times cometem é escolher o primitivo errado porque comparam o que cada um contém. A pergunta certa é como cada um é acionado. Este capítulo percorre a decisão e depois cada primitivo, um a um.

A árvore de decisão: escolha pelo modo como é acionado, não pelo conteúdo

Os quatro primitivos de autoria parecem semelhantes no papel. Cada um é um arquivo Markdown com frontmatter e instruções, cada um molda como o Copilot responde, e cada um vive sob .github/. A tentação é escolher entre eles lendo o conteúdo, mas o conteúdo se sobrepõe tanto que a comparação trava. A regra confiável é outra: escolha pelo mecanismo de invocação. Um agent é invocado por nome. Um chat mode é selecionado no picker. Uma skill é comparada automaticamente com o seu pedido. Um prompt file roda por slash command. O gatilho, não o conteúdo, é o eixo que os separa com clareza.

O mecanismo mapeia a intenção. Se você quer um especialista nomeado para uma classe inteira de tarefas, isso é um agent. Se você quer mudar o estilo da conversa ou a profundidade da explicação, isso é um chat mode. Se você quer que o Copilot busque conhecimento de domínio empacotado por conta própria, isso é uma skill. Se você quer uma receita de uso único que invoca deliberadamente, isso é um prompt file. Na ordem de montagem, apenas um primitivo de autoria é anexado por turno, no passo seis, depois que as instruções do tier hot e as instruções com escopo já foram

carregadas. Então o primitivo que você invoca refina o contrato do workspace. Ele não o substitui.

> DEFINIÇÃO
>
> A regra de seleção em uma linha: escolha o primitivo de autoria pelo modo como é acionado, não pelo que ele contém. Invocado por nome é um agent. Selecionado no picker é um chat mode. Comparado por descrição é uma skill. Executado por slash command é um prompt file.

Agents: personas nomeadas com suas próprias ferramentas e menor privilégio

Os arquivos de agent vivem em .github/agents/ com a extensão .agent.md. Cada um define uma persona especializada com suas próprias instruções, seu próprio escopo de comportamento e, o mais importante, sua própria lista de ferramentas permitidas. Um agent fica ativo por vez. Quando você o invoca por nome, ele substitui a persona de chat padrão para aquela classe de trabalho. O campo tools é o que distingue agents de todos os outros primitivos. Você lista exatamente as ferramentas que o agent pode chamar, e nada além disso fica disponível para ele. Um agent revisor que apenas lê e comenta não recebe acesso a shell ou escrita. Isso é menor privilégio expresso como configuração.

O frontmatter também carrega um campo model, que roteia o agent para um modelo específico no picker do Copilot. Um agent de classificação barato pode rodar no Haiku 4.5. Um agent de refatoração difícil pode rodar no Opus 4.8 ou no Claude Fable 5. Como o faturamento roda sobre GitHub AI Credits, onde um crédito é um centavo, rotear agents rotineiros para um modelo menor é uma alavanca direta de custo, não uma preferência. O estudo de caso empírico sobre contexto codificado encontrou dezenove agents especialistas com média de 711 linhas para agents de maior capacidade e 327 para os padrão, com mais da metade de cada especificação sendo conhecimento de domínio do projeto, e não instrução de comportamento. Um agent é sobretudo o que ele sabe, não como você diz para ele agir.

Chat modes e skills: estilo de conversa versus conhecimento carregado automaticamente

Os chat modes vivem em .github/chatmodes/ e diferem dos agents de três formas. Não carregam lista própria de ferramentas. São selecionados em vez de invocados. E permanecem ativos entre turnos até você trocar. Um chat mode muda o estilo e a autoridade da explicação, não a capacidade. O modo teach anota cada sugestão linha a linha. O modo summarize comprime. O modo rubber-duck devolve perguntas. Quando o comportamento que você quer é sobre tom, profundidade ou persona para uma conversa inteira, e não sobre quais ferramentas são permitidas, um chat mode é o primitivo certo e um agent é exagero.

Quando recorrer a uma skill

As skills vivem em .github/skills/{nome}/SKILL.md, cada uma na própria pasta, para carregar arquivos de referência, scripts e templates ao lado das instruções. A distinção que importa é o gatilho. O Copilot carrega uma skill automaticamente quando o seu pedido corresponde à descrição da skill no frontmatter. Você não a nomeia e não a seleciona. Isso torna as skills a resposta quando o mesmo fluxo de várias etapas é reexplicado toda hora. Escreva uma vez, descreva bem, e o Copilot a encontra por conta própria. O registro comunitário Awesome GitHub Copilot distribui skills e prompts dessa forma, então uma boa descrição é o que torna uma skill descobrível e reutilizável, no seu time e além dele.

Prompt files: slash commands de uso único para receitas conhecidas

Os prompt files vivem em .github/prompts/ com a extensão .prompt.md. Ao contrário das skills, nunca são acionados automaticamente. O usuário os chama explicitamente com um slash command, então /release-notes ou /pr-summary roda uma receita específica e previsível sob demanda. Use um prompt file quando o fluxo é discreto, os passos são conhecidos, e quem o executa já sabe que quer executá-lo. Um prompt file pode declarar suas próprias ferramentas e seu próprio modelo no frontmatter, igual a um agent, mas é uma única invocação, não uma persona persistente. Ele faz um trabalho e retorna.

A fronteira entre uma skill e um prompt file é, de novo, o gatilho. Uma skill é para conhecimento que o Copilot deve buscar por conta própria quando o momento pede. Um prompt file é para uma receita que um humano executa deliberadamente. Se você se pega desejando que o Copilot soubesse fazer algo, isso é uma skill. Se você fica digitando a mesma instrução longa para fazê-lo executar uma tarefa conhecida, isso é um prompt file. Ambos mantêm o tier hot enxuto por viverem no tier warm e carregarem apenas quando acionados, o que mantém o contexto sempre carregado pequeno e os AI Credits gastos por turno previsíveis.

Referências

Fontes primárias para os primitivos de autoria, seus mecanismos de invocação e a evidência empírica por trás do design de agents e skills.

- Customize AI in Visual Studio Code, a documentação de referência para agents, chat modes, skills e prompt files no VS Code.
- Agent READMEs: An Empirical Study of Context Files for Agentic Coding, a fonte para os 14,5% dos arquivos de contexto que trazem orientação explícita de segurança.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, a fonte para o benchmark de 711 e 327 linhas por agent em dezenove agents especialistas.
- GitHub Copilot documentation, a plataforma de referência para o agent mode e os primitivos de autoria deste capítulo.
- Awesome GitHub Copilot, o registro comunitário que distribui skills, prompts e agents como artefatos compartilháveis.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

04 GOVERNANÇA DE RUNTIME

O que roda ao redor do agente: a camada de runtime que governa cada resposta do Copilot.

As primeiras partes deste playbook descrevem o que o GitHub Copilot sabe: as instruções de workspace que ele sempre lê e os artefatos de autoria que carrega sob demanda. Este capítulo descreve o que roda ao redor do agente enquanto ele trabalha. Quatro superfícies de runtime (hooks, conexões MCP, plugins e settings do VS Code) decidem o que o agente pode fazer, onde ele obtém dados frescos e como seu comportamento é travado em toda a organização. Nenhuma delas carrega conteúdo de prompt. Elas governam o loop. De responsabilidade do platform team, configuradas uma vez e mudadas raramente, são a diferença entre um agent mode produtivo e um agent mode seguro.

Hooks: guardrails determinísticos em cada chamada de ferramenta

Hooks vivem em .github/hooks/*.json. Disparam em eventos de ciclo de vida (preToolUse, postToolUse, onSessionStart, onPromptSubmit, onFileChange) e podem permitir, negar, mutar ou registrar um evento antes ou depois de o Copilot agir. Um hook é código que roda ao redor de uma ação do agente, não dentro dela. Ele não pede ao modelo que se comporte. Ele aplica a regra de forma determinística, do mesmo jeito que o CI aplica um build. Um hook preToolUse que casa com um comando de shell destrutivo e retorna deny impede que "limpe o cluster" vire kubectl delete namespace prod. O modelo nunca tem a chance de ser esperto.

Hooks importam sobretudo por causa de um gap medido. Nos 2.303 arquivos reais de contexto de agente estudados por Chatlatanagulchai e colegas em 2025, orientação explícita de segurança aparece em apenas 14,5% dos repositórios. Isso deixa 85,5% sem nenhuma regra de segurança escrita que o modelo pudesse

seguir. Hooks são o backstop determinístico para esses 85,5%. Onde um arquivo de instrução diz "por favor, não faça", um hook simplesmente recusa. Cinco linhas de JSON em um evento preToolUse evitam o incidente que o time ainda não teve, e, ao contrário de um prompt, a garantia não depende de qual modelo está no Copilot picker nem de como a requisição foi formulada.

Conexões MCP e plugins: dados frescos e bundles versionados

MCP, o Model Context Protocol, é o padrão aberto que o Copilot usa para alcançar sistemas externos em runtime: bancos de dados, rastreadores de issues, plataformas de observabilidade, APIs internas. Servidores são declarados em .mcp.json para o workspace ou em settings.json para o usuário, descobertos uma vez por sessão, e cada um expõe tools, resources e prompts que o agente pode chamar. A regra de quando adicionar um é simples. Se o dado vive fora do repositório e muda entre sessões (issues abertas, linhas de log, registros em uma tabela), ele pertence atrás de um servidor MCP, não colado no chat. Se a ação que um servidor expõe é destrutiva, envolva-a com um hook preToolUse. Cada chamada MCP gasta latência e GitHub AI Credits, então credenciais de menor privilégio e uma superfície de ferramentas estreita fazem parte da configuração, não um detalhe posterior.

Plugins resolvem a distribuição. Um plugin empacota agents, chat modes, skills, prompt files, hooks e declarações MCP atrás de um único manifesto plugin.json, fixado em uma versão. É como um padrão viaja por muitos repositórios sem copy-paste. No momento em que um time se pega copiando o conteúdo de .github/ de um repositório para outro, o drift já começou: quarenta repositórios com quarenta arquivos de instrução editados à mão divergem em um trimestre. Um plugin, fixado por repositório, transforma isso em um único pull request contra o plugin. O registro da comunidade, Awesome GitHub Copilot, distribui o trabalho compartilhado do mesmo jeito. Plugins são configuração cold: instalados uma vez, versionados e mudados por revisão, nunca editados por feature.

Precedência de settings: o contrato do time desce, o gosto pessoal sobe

Todo comportamento do Copilot que não é um arquivo é um setting, e settings se resolvem em ordem estrita, do menos específico ao mais específico: default (embutido no produto), user (~/.../User/settings.json, nunca commitado), workspace (.vscode/settings.json, commitado no repositório) e workspace folder (para um pacote em um setup multi-root). O escopo mais específico vence, com uma exceção que importa. No topo está a política do GitHub Enterprise, que define floors: exclusão de conteúdo, telemetria e uma trava de modelo que nenhum setting de workspace ou de usuário pode sobrescrever. Uma organização que trava a escolha de modelo no Copilot picker, por exemplo entre Claude Fable 5, Opus 4.8, Sonnet 5, Gemini 3.1 Pro ou um modelo GPT-5.x, faz isso aqui, e a trava vale até o nível mais interno.

A herança é unidirecional por design. A organização define floors, o workspace define o contrato do time, o usuário personaliza, e a sessão (arquivos fixados, abas abertas, a seleção atual) é a camada mais interna e mais efêmera. O escopo externo não pode ser sobrescrito de dentro, e settings internos nunca vazam para fora, então uma preferência pessoal nunca é commitada no repositório por acidente. O princípio operacional decorre do formato: aplique no escopo externo, não policie por usuário. Trave a política no nível da organização ou do workspace, coloque useInstructionFiles e instructionsFilesLocations em .vscode/settings.json para que o contrato do time seja real e não aspiracional, e deixe cada desenvolvedor adicionar preferências ergonômicas que não podem comprometer o contrato.

Ordem de montagem: os nove passos que dizem por que uma regra foi ignorada

Quando um desenvolvedor submete um prompt, o Copilot monta o contexto efetivo em uma sequência determinística de nove passos. Os passos um a três resolvem os settings em ordem de precedência, default depois user depois workspace. O passo quatro carrega a hot memory, .github/copilot-instructions.md, incondicionalmente. O passo cinco casa as instruções com escopo, avaliando cada glob applyTo contra os arquivos abertos e mergeando os conjuntos que correspondem. O passo seis anexa o primitivo de autoria ativo, seja o agent, chat mode, skill ou prompt file em jogo. O

passo sete roda os hooks onPromptSubmit, que podem negar, mutar ou anotar a requisição. O passo oito descobre as tools MCP, listando os servidores conectados e expondo seu catálogo. O passo nove envia ao modelo o system prompt montado, o catálogo de ferramentas e a mensagem do usuário.

O valor de conhecer a ordem é diagnóstico. Quando o Copilot parece ignorar uma regra, o modelo quase nunca é a causa. O bug está no passo três (o setting resolvido no escopo errado), no passo cinco (o glob applyTo não casa com o arquivo que está de fato aberto) ou no passo sete (um hook mutou ou bloqueou a instrução em silêncio). Percorra os nove passos de cima para baixo antes de discutir com o modelo. O mesmo determinismo que torna a sequência depurável também a torna auditável: o mesmo repositório no mesmo commit monta o mesmo contexto toda vez, então uma regra que funcionou ontem e não funciona hoje aponta para uma mudança em um de nove lugares nomeados, não para um humor do modelo.

> DEFINIÇÃO
>
> Governança de runtime em uma linha: hooks decidem o que o agente pode fazer, MCP decide onde ele obtém dados frescos, plugins decidem como o padrão viaja, e settings decidem quem pode mudar qualquer um deles. Quando uma regra é ignorada, percorra os nove passos de montagem de cima para baixo. A resposta está quase sempre no passo três, cinco ou sete, não no modelo.

Referências

Fontes primárias para as superfícies de runtime: o gap de segurança que justifica os hooks, o padrão MCP e seu modelo de ameaças, o registro de plugins, a precedência de settings e a evidência de herança.

- Chatlatanagulchai et al., An Empirical Study of Agent Context Files, o gap medido: orientação explícita de segurança aparece em apenas 14,5% de 2.303 arquivos reais de contexto, os 85,5% que os hooks amparam.
- Model Context Protocol Specification, o padrão aberto que o Copilot usa para alcançar sistemas externos em runtime.

- MCP Security, Threat Model and Mitigations, a base para envolver qualquer ação destrutiva de MCP em um hook preToolUse.
- Awesome GitHub Copilot, o registro da comunidade que distribui plugins, agents, chat modes e prompts.
- VS Code, Settings Precedence, a referência de como os escopos default, user, workspace e folder se resolvem.
- Vasilopoulos, Codified Context Infrastructure, o estudo de caso de 108.000 linhas que manteve uma única constituição de workspace ao longo de 283 sessões com razão conhecimento-código de 24,2%.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 SEGUNDA DE MANHÃ

Do zero à baseline em uma semana.

A superfície de dez camadas não vale nada se ninguém a tocar. Este capítulo é a menor sequência de commits que leva um time do zero a uma baseline de configuração real em uma semana. Um commit por dia, cada um fechando uma lacuna específica que a pesquisa documenta. Na sexta-feira, as camadas de workspace, autoria e runtime já têm um primeiro inquilino, e o comportamento do GitHub Copilot no repositório passa a ser auditável em vez de tribal.

A sequência de commits, dia a dia

Segunda-feira: crie o .github/copilot-instructions.md com quatro seções: Stack, Convenções, Segurança e Performance. Mantenha o arquivo abaixo de cerca de 1.000 tokens. As seções de Segurança e Performance são as que mais importam, porque o estudo de 2.303 arquivos de contexto reais encontrou orientação de segurança em apenas 14,5% deles e regras de performance em outros 14,5%. Escrever essas duas seções no primeiro dia fecha a lacuna que produz código produtivo, mas inseguro.

Terça-feira: adicione o .vscode/settings.json com useInstructionFiles ativado e instructionsFilesLocations apontando para .github/instructions. É o passo que torna o contrato do time real em vez de aspiracional. Commite as settings de workspace para que cada colaborador carregue o mesmo comportamento. Deixe as preferências pessoais nas settings de usuário, que nunca chegam ao repositório.

De quarta a sexta: escopo, guardrails, reúso

Quarta-feira: escreva um arquivo de instrução com escopo em .github/instructions/ com um glob applyTo mirando o diretório mais arriscado do repositório, migrations, pagamentos ou uma superfície de API administrativa. É a passagem de um único arquivo monolítico, que deixa de escalar por volta de 3.000 tokens, para o modelo de três tiers. Quinta-feira: escreva um hook em .github/hooks/ que negue comandos

de shell destrutivos antes de eles rodarem. Cinco linhas de JSON são o backstop determinístico para os caminhos de código que o modelo nem sempre protege sozinho. Sexta-feira: extraia um fluxo recorrente de vários passos para uma skill em .github/skills/ com um campo de descrição forte, para que o agent mode a encontre por match de descrição em vez de um colega reexplicá-la a cada sprint.

Na semana seguinte, empacote os cinco artefatos em um plugin e instale-o nos três principais repositórios. Um plugin reúne agents, chat modes, skills, prompt files, hooks e declarações MCP atrás de um único manifesto, fixado a uma versão. As mudanças passam então por um único pull request no plugin, em vez de quarenta edições de copiar e colar em quarenta repositórios.

> DEFINIÇÃO
>
> Um commit por dia: segunda o copilot-instructions.md, terça o settings.json, quarta uma instrução com escopo, quinta um hook, sexta uma skill, na semana seguinte um plugin. Cada commit dá a uma camada seu primeiro inquilino e fecha uma lacuna que a pesquisa nomeia.

Expectativas honestas: o que a hierarquia fará e o que não fará

A baseline não vai transformar um time júnior em um time sênior. O GitHub Copilot amplifica o que está escrito. Se uma convenção vive apenas na cabeça de alguém, o modelo não consegue segui-la, e a hierarquia não muda nada até que a convenção seja codificada. As camadas são um lugar para colocar conhecimento, não um substituto para tê-lo.

O que a baseline vai fazer é tornar o comportamento reproduzível. A avaliação da Zoominfo sobre um rollout com mais de 400 desenvolvedores relatou uma taxa de aceitação de sugestões sustentada perto de 30% depois que as settings de nível de workspace foram padronizadas, em vez de resultados que variavam de uma máquina para a outra. Uma baseline reproduzível de 30% é um número que um time consegue defender em um code review e melhorar de forma deliberada. A

abordagem de arquivo único não oferece isso, porque drifta em silêncio no momento em que o arquivo cresce além do orçamento do modelo.

> DEFINIÇÃO
>
> O que não fará: fazer aparecer convenções não documentadas. O que fará: dar ao time uma baseline de aceitação reproduzível perto de 30% (Zoominfo 2025) em vez de histórias de funciona-na-minha-máquina.

Governança, não prompt engineering: tornar cada camada auditável e com dono

A hierarquia não é uma checklist de features. É um modelo de governança para IA em engenharia de software: quem declara contexto, quem autora instruções e o que roda a cada prompt. Cada uma das dez camadas tem um dono. Instruções de workspace e instruções com escopo pertencem ao time como um contrato, commitadas no repositório e revisadas como qualquer outro código. Agents, chat modes, skills e prompt files pertencem a quem é dono da tarefa que eles codificam. Hooks, conexões MCP, plugins e settings pertencem ao platform team, porque configuram o runtime ao redor do agente, não uma feature isolada.

A auditabilidade decorre dessa titularidade. Cada regra vive em um arquivo versionado, revisado em um pull request, atribuído a um commit. Quando o Copilot ignora uma regra, a correção é percorrer a ordem de montagem de cima para baixo, verificando o escopo das settings, depois o glob applyTo, depois qualquer hook que mute a requisição, em vez de discutir com o modelo. Aplique a política no escopo externo e não policie por usuário: trave os floors da organização e do workspace, e deixe os escopos internos adicionar preferências ergonômicas sem quebrar o contrato. Times que tratam a hierarquia como governança entregam. Times que a tratam como prompt engineering estagnam.

Referências e leitura adicional

Fontes primárias para a baseline dia a dia, as lacunas empíricas que ela fecha e os números de aceitação que ela mira.

- Agent READMEs: An Empirical Study of Context Files for Agentic Coding (Chatlatanagulchai et al., 2025), a fonte para a lacuna de 14,5% em segurança e performance que o commit de segunda-feira fecha, em 2.303 arquivos de contexto reais.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase (Vasilopoulos, 2026), o estudo de caso de 108.000 linhas e 283 sessões por trás do modelo de três tiers e do colapso do arquivo único acima de 3.000 tokens.
- A Four-Phase Evaluation of GitHub Copilot Across 400+ Developers (Zoominfo, 2025), a fonte para a taxa de aceitação reproduzível perto de 30% depois que as settings de workspace são padronizadas.
- Customize AI in Visual Studio Code, a referência para arquivos de instrução, instruções com escopo e as settings que tornam o contrato do time real.
- GitHub Copilot Documentation, a referência de plataforma para agent mode, copilot-instructions.md e as camadas de runtime que este capítulo configura.
- Model Context Protocol Specification, o padrão aberto para as conexões MCP que um plugin pode empacotar junto de hooks e skills.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps