GITHUB COPILOT: AGENTS, SKILLS, PLUGINS, AND HOOKS · PLAYBOOK

A primitiva certa para a tarefa decide se um workspace escala ou colapsa.

A maioria dos times adota o GitHub Copilot como uma única caixa de chat, e ela quebra no segundo caso de uso. Este playbook trata o roteamento como a primeira decisão de engenharia: seis primitivas, agentes, skills, plugins, servidores MCP, hooks e a hierarquia de configuração, cada uma com um limite que decide se o workspace escala. O retorno é mensurável, cerca de 30 por cento menos tokens por tarefa, agora cobrados em GitHub AI Credits, e uma saída que permanece consistente no time inteiro.

AUTORA

Paula Silva

CARGO

AI-Native Software Engineer

CONTATO

in/paulanunes

TEMPO DE LEITURA

~ 45 minutos, fim a fim

6	6	4
PRIMITIVAS CENTRAIS	CAPÍTULOS	PERGUNTAS DE ROTEAMENTO

~30%

CORTE NO GASTO DE TOKENS

CAPÍTULOS

Chapter 00

Roteamento é a primeira decisão de engenharia

Por que a caixa de chat única quebra no segundo caso de uso, a lacuna de capacidade do SWE-bench ao FeatureBench em números, quanto custa uma tarefa mal roteada em GitHub AI Credits, e as seis primitivas cujos limites decidem a escala.

O Problema →

Chapter 01

Quatro pontos de controle, um fluxo de tarefa

Como uma resposta do Copilot é montada a partir de uma pilha de arquivos: a pilha do workspace lida de baixo para cima, os quatro tipos de arquivo centrais, as seis camadas de prompt e a hierarquia de configuração do .github/ como uma Pedra de Roseta.

O Mapa →

Chapter 02

SKILL.md: anatomia, ativação e maturidade

As cinco seções e o teste SFA, os três modos de ativação e seu custo em tokens, o ciclo de vida da ideia à distribuição, e o modelo de maturidade de cinco níveis que trata a skill como código.

Skills →

Chapter 03

Dois tipos de agente e a árvore de decisão

Generalista versus custom agents, a proporção saudável de um agente para cada oito a doze skills, o contrato de seis seções do .agent.md, a árvore de roteamento de quatro perguntas e a escolha do modelo no picker do Copilot.

Agentes →

Chapter 04

Orquestração, handoffs e trilhos

O runtime de quatro fases com um orquestrador, a cadeia de handoffs Plan, Code, Test, Security que acumula contexto, os seis eventos de ciclo de vida de hooks e o modelo de governança em que o deny é absoluto.

Execução →

Chapter 05

Anti-padrões, diagnósticos e 90 dias

Os quatro anti-padrões que aparecem primeiro, seis perguntas diagnósticas, o playbook de adoção de 90 dias, e o que uma hora de design de roteamento devolve em GitHub AI Credits.

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

Roteamento é a primeira decisão de engenharia.

A maioria dos times adota o GitHub Copilot como uma única caixa de chat. Toda tarefa vai para o mesmo agente generalista, toda resposta é remontada a partir do mesmo context window, e o arranjo funciona até chegar o segundo caso de uso. Este capítulo defende que roteamento, decidir qual tarefa vai para qual primitiva, é a primeira decisão de engenharia, não uma otimização posterior. Ele coloca números na lacuna de capacidade, precifica o custo de uma tarefa mal roteada em GitHub AI Credits e nomeia as seis primitivas cujos limites decidem se um workspace escala.

A armadilha da caixa de chat única: um agente, um context window, um gargalo

O erro mais comum que vejo os times cometerem com o GitHub Copilot é tratá-lo como uma única interface de chat generalista. Toda tarefa vai para o mesmo agente em agent mode. Toda resposta reconstrói o contexto a partir da mesma janela. Em um workspace maduro, um orquestrador generalista dá conta de cerca de 70% a 80% das tarefas, e isso está ótimo. A armadilha é mandar para ele também os outros 20% a 30%, o trabalho que precisa de uma skill específica, de uma ferramenta exclusiva ou de uma identidade distinta. Um agente absorve tudo, e o context window vira o gargalo.

Esse gargalo cresce com o time, não com o indivíduo. Para um desenvolvedor, um context window um pouco inchado é ruído. Entre dezenas de desenvolvedores rodando o mesmo generalista em toda tarefa, vira um motor de custo mensurável. O context window tem orçamento fixo. Quando um único agente carrega regras de marca, padrões de segurança, vocabulário de domínio e a tarefa atual ao mesmo tempo, o sinal útil é empurrado para fora. A saída desvia, os revisores pegam mais

erros, e a produtividade prometida pela ferramenta empaca. A falha não é do modelo. É do roteamento.

A lacuna de capacidade em números: o SWE-bench parece resolvido, features reais não

A distância entre um benchmark e uma feature real é onde o argumento pelo roteamento fica concreto. No SWE-bench, um modelo de fronteira resolve 74,4% das correções de bug bem escopadas, as tarefas com requisitos apertados e isolados. Esse número parece dizer que o problema está resolvido. No FeatureBench, que representa trabalho de feature do mundo real em vez de patches localizados, o mesmo modelo sem contexto de domínio cai para 11% (arXiv:2602.10975). O benchmark media patches localizados, não desenvolvimento de features, então os dois números descrevem dois trabalhos diferentes.

A parte recuperável dessa lacuna é a parte interessante. Adicione contexto de skill, ou seja, padrões de arquitetura, convenções de código e vocabulário de domínio entregues por um SKILL.md, e o número no FeatureBench sobe para cerca de 38%. As skills fecham por volta de 27 pontos percentuais da distância. O resto não se fecha com um modelo maior. Fecha-se com custom agents com acesso dedicado a ferramentas, as primitivas para as quais este playbook constrói caminho. A lição é que contexto, entregue pela primitiva certa, move o número mais do que a capacidade bruta do modelo.

Quanto custa uma tarefa mal roteada, agora precificada em GitHub AI Credits

Uma tarefa mal roteada não é só de qualidade menor. Ela é mais cara, e agora o custo vem discriminado. A telemetria de produção do GitHub Copilot Enterprise no primeiro trimestre de 2026 mostra que um agente generalista tratando trabalho que uma skill especializada responderia direto consome cerca de três vezes mais tokens de contexto por tarefa. Desde 1 de junho de 2026, esse consumo é medido em GitHub AI Credits, onde um crédito equivale a um centavo de dólar. O desperdício deixou de se esconder dentro de uma assinatura fixa. Ele aparece na conta.

Multiplique três vezes os tokens por cada desenvolvedor que roda o generalista em trabalho com cara de skill, e o número deixa de ser ruído. A mesma telemetria aponta para o outro lado também: times que investem uma hora em desenho de roteamento, adotando uma árvore de decisão simples em vez de escolher no feeling, cortam o gasto médio de tokens em cerca de 30%. Esse é o argumento econômico central deste playbook. Roteamento não é um imposto sobre a velocidade. É a diferença entre GitHub AI Credits gastos raciocinando e GitHub AI Credits gastos reconstruindo contexto que o workspace já tinha.

As seis primitivas, e por que seus limites decidem a escala

O workspace moderno do GitHub Copilot não é um assistente só. É uma composição de seis primitivas, cada uma com seu formato de arquivo, seu ciclo de vida e seu contrato com as demais. Agentes escolhem o caminho. Skills empacotam o conhecimento de domínio em um SKILL.md versionado. Plugins alcançam o mundo. Servidores MCP expõem ferramentas tipadas pelo Model Context Protocol. Hooks mantêm os trilhos determinísticos, disparando em eventos fixos do ciclo de vida. E a hierarquia de configuração, ancorada em arquivos como o .github/copilot-instructions.md, decide quem vence quando duas delas discordam.

Os limites entre essas seis é o que importa. Uma skill é conhecimento, não uma ferramenta. Um agente é uma identidade e um escopo, não um depósito de conhecimento. Um hook é um gate determinístico, não um prompt. Quando esses limites são explícitos, um workspace compõe de forma limpa e escala de um desenvolvedor para a organização inteira. Quando eles borram, tudo colapsa de volta para a caixa de chat única, e o workspace quebra no segundo caso de uso. Acertar os limites é o trabalho inteiro. O resto deste playbook percorre um a um, primitiva por primitiva.

DEFINIÇÃO

Roteamento em uma frase: decida qual das seis primitivas é dona de uma tarefa antes de enviá-la, porque um generalista fazendo trabalho com cara de skill custa cerca de 3x os tokens de contexto, agora cobrados em GitHub AI Credits, e a lacuna de capacidade entre um benchmark resolvido e uma feature real se fecha com a primitiva certa, não com um modelo maior.

Referências

Fontes primárias para a lacuna de capacidade, a decisão de roteamento e as seis primitivas que este capítulo nomeia.

- FeatureBench, Benchmarking Agentic Coding for Complex Feature Development, a lacuna empírica: um modelo de fronteira com 74,4% no SWE-bench resolve 11% das tarefas de feature real sem contexto de domínio.
- Agent Skills in the Wild, An Empirical Study of Security Vulnerabilities at Scale, contexto para por que os limites e a governança de skills importam à medida que skills proliferam em um workspace.
- Customize AI in Visual Studio Code, a referência para skills, custom agents, instructions e prompt files no workspace.
- GitHub Copilot Documentation, a plataforma de referência para agent mode, MCP e a hierarquia 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

01 O MAPA

Quatro pontos de controle, um fluxo de tarefa: como a pilha do workspace monta uma resposta.

Quando você envia uma tarefa ao GitHub Copilot em agent mode, a resposta não é produzida só pelo modelo. Ela é produzida por uma pilha de arquivos que se montam em torno do seu pedido. Leia a pilha de baixo para cima e ela conta uma história só: a skill carrega o conhecimento de domínio, as ferramentas se conectam às capacidades externas, os hooks controlam o que pode rodar, e o agente responde com tudo isso em contexto. Este capítulo mapeia os quatro pontos de controle, as seis camadas de prompt que eles alimentam e onde cada primitivo vive no repositório.

A pilha do workspace, lida de baixo para cima

Eu leio um workspace do Copilot como leio uma linha de fábrica. Começo por baixo e sigo o material para cima. A skill carrega primeiro: um SKILL.md traz o conhecimento de domínio que a tarefa precisa, as convenções e o workflow. As ferramentas se conectam em seguida: servidores MCP, CLIs e APIs dão ao agente capacidades que ele não tem por padrão. Os hooks controlam a execução: um hook preToolUse pode negar um comando perigoso antes que ele rode. O agente responde por último, com o conhecimento de domínio, as ferramentas e os guardrails já no lugar. Quatro camadas, quatro funções, uma resposta.

O valor de cada camada fica visível quando ela é pulada. Pule a skill e o agente chuta as convenções que a base de código já codifica; a telemetria de produção do GitHub Copilot Enterprise no primeiro trimestre de 2026 mostrou que um agente generalista lidando com uma tarefa que uma skill especializada poderia responder diretamente consumiu 3 vezes mais tokens de contexto para o mesmo trabalho. Esse overhead não é abstrato. Ele chega na conta como GitHub AI Credits, onde um crédito é um centavo de dólar. Pule as ferramentas e o agente não alcança o banco

de dados ou a API de que precisa, então fabrica uma resposta plausível no lugar. Pule os hooks e nada fica entre um comando gerado e o seu repositório. Pule a identidade do agente e um único generalista carrega toda tarefa, sem escopo delimitado e sem ferramentas exclusivas.

A camada de skill é onde o maior gap se fecha

A medição mais clara da camada de skill vem de dois benchmarks. No SWE-bench, que pontua correções de bug precisas e isoladas, um modelo de fronteira resolve cerca de 74% das tarefas. No FeatureBench, que representa trabalho de feature real, o mesmo modelo sem contexto de domínio cai para 11%. Com contexto de skill, os padrões de arquitetura, as convenções de código, o vocabulário de domínio, o número sobe para aproximadamente 38%. A camada de skill fecha cerca de 27 pontos percentuais desse gap. A distância restante é para o que servem os custom agents com ferramentas dedicadas. É por isso que vale ler a pilha como pilha: cada camada recupera uma capacidade que a camada de baixo não alcança.

Os quatro tipos de arquivo centrais

Quatro tipos de arquivo sustentam todo o ecossistema, e cada um responde a uma pergunta diferente. O primeiro são as instruções, e vêm em dois escopos. O arquivo de workspace, .github/copilot-instructions.md, guarda as regras que valem para todo o repositório: a stack, as convenções de nomenclatura, os comandos. As instruções com escopo vivem em .github/instructions/ como arquivos .instructions.md, cada um com um glob applyTo que o limita a um caminho como src/components/**/*.tsx. As instruções respondem à pergunta do que o agente deve fazer, e se aplicam automaticamente, sem o usuário pedir. O agente não pode ignorá-las; elas funcionam como guardrails.

Os outros três tipos de arquivo respondem às perguntas restantes. As ferramentas, declaradas em .mcp.json, respondem o que o agente pode fazer: um servidor MCP o conecta ao GitHub, a um banco de dados ou a um serviço interno; uma CLI lhe dá git ou az; uma API lhe dá uma chamada HTTP. O arquivo .agent.md em .github/agents/ responde quem o agente é: seu nome e papel, as ferramentas que ele pode usar, as tarefas que aceita e recusa, e as regras de handoff para quando delega. O arquivo SKILL.md em .github/skills/ responde o que o agente sabe: os templates, o workflow

passo a passo e o checklist de validação para um tipo de saída. Regras restringem comportamento, ferramentas habilitam capacidade, identidade delimita escopo, conhecimento eleva qualidade. Os quatro são independentes e combináveis, e esse é o ponto.

As seis camadas de prompt que montam o contexto final

Esses arquivos não chegam ao modelo como arquivos. Eles são mesclados em um único contexto, montado em seis camadas. A base do modelo vem primeiro: o system prompt definido pelo GitHub e pelo modelo que você escolheu no seletor do Copilot, seja Claude Opus 4.8, Sonnet 5, Haiku 4.5, Claude Fable 5, Gemini 3.1 Pro ou um modelo GPT-5.x. As instruções de copilot-instructions.md são injetadas em seguida, depois as instruções com escopo que correspondem ao arquivo atual. A identidade do agente é carregada do .agent.md ativo. O conhecimento da skill é adicionado do SKILL.md acionado. Seu prompt é colocado por último, como a entrada de maior prioridade. Camadas posteriores refinam as anteriores, mas não as contradizem: uma skill não sobrepõe uma regra de segurança, e um prompt não burla um comando negado.

A razão para seis camadas separadas, alimentadas por arquivos separados, é a evolução. Cada tipo de arquivo mira uma camada para poder mudar por conta própria. Você pode versionar um SKILL.md e promovê-lo por um modelo de maturidade sem tocar no agente que o carrega. Você pode apertar um glob applyTo em uma instrução com escopo sem editar o arquivo de workspace que todo desenvolvedor lê. Você pode trocar o modelo no seletor sem reescrever uma única instrução. Separação de responsabilidades no nível do arquivo vira evolução independente em tempo de execução. Essa é a diferença entre um workspace que você consegue manter e um megaprompt único que ninguém ousa editar.

DEFINIÇÃO

As seis camadas, na ordem de montagem: base do modelo, depois instruções de workspace, depois instruções com escopo, depois identidade do agente, depois conhecimento da skill, depois o seu prompt por último como a entrada de maior prioridade. Cada camada refina a anterior, e nenhuma pode contradizê-la.

A hierarquia de configuração como Pedra de Roseta

No GitHub Copilot, cada primitivo tem um lugar fixo dentro de .github/, e a localização é o contrato. As instruções de workspace ficam em .github/copilot-instructions.md. As instruções com escopo ficam em .github/instructions/ como arquivos .instructions.md com globs applyTo. Os agentes ficam em .github/agents/ como arquivos .agent.md. As skills ficam em .github/skills/ como arquivos SKILL.md dentro de uma pasta nomeada. Os hooks ficam em .github/hooks/ como arquivos JSON que declaram handlers preToolUse, postToolUse, sessionStart e sessionEnd. Os templates de prompt reutilizáveis ficam em .github/prompts/ como arquivos .prompt.md. As conexões MCP ficam em .mcp.json na raiz. Os plugins empacotam vários desses juntos por um manifesto plugin.json, distribuídos pelo marketplace Awesome GitHub Copilot.

Chamo essa tabela de Pedra de Roseta porque o mesmo conceito existe no Claude Code e no GitHub Copilot com nomes e caminhos diferentes, e quando você consegue ler uma coluna, consegue ler a outra. O copilot-instructions.md é o equivalente do CLAUDE.md no Copilot. O .github/instructions/ é o equivalente do .claude/rules/. O .github/agents/ corresponde ao .claude/agents/, e o .github/skills/ corresponde ao .claude/skills/ com um formato SKILL.md idêntico. A única camada que você não escreve é a base do modelo; você a escolhe no seletor e paga por ela em AI Credits. Todo o resto é um arquivo que você escreve, coloca em um lugar previsível e deixa o orquestrador descobrir. Essa previsibilidade é o que permite a um workspace escalar além de uma única caixa de chat.

Referências

Fontes primárias para os quatro tipos de arquivo, as camadas de prompt e a hierarquia de configuração do GitHub Copilot.

- GitHub Copilot, Adding repository custom instructions, a referência para copilot-instructions.md e arquivos .instructions.md com escopo e globs applyTo.
- Customize AI in Visual Studio Code, a visão geral de agents, prompt files e skills como primitivos de customização separados.
- Prompt Files in VS Code, o formato para templates .prompt.md reutilizáveis com frontmatter de agent mode, modelo e ferramentas.
- GitHub Copilot Hooks Configuration Reference, o schema JSON e o modelo de eventos para hooks preToolUse, postToolUse, sessionStart e sessionEnd.
- Model Context Protocol Specification, o padrão aberto por trás das conexões de ferramentas em .mcp.json.
- GitHub Copilot: Instructions vs Prompts vs Custom Agents vs Skills, uma comparação lado a lado de cada primitivo de configuração do Copilot.

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 EMPACOTANDO CONHECIMENTO

SKILL.md: anatomia, ativação e maturidade.

Um SKILL.md não é um prompt, não é um arquivo de configuração e não é um script. É um artefato de conhecimento de domínio com uma estrutura fixa, três modos de ativação e um caminho de maturidade que vai de um rascunho de quarenta linhas a uma entrada de registro com versão fixada. Este capítulo desmonta o artefato. Ele cobre as cinco seções que toda skill carrega, o teste SFA que controla o uso em produção, como cada modo de ativação decide o custo em tokens, o ciclo de vida da ideia à distribuição, e o modelo de cinco níveis que permite a uma equipe tratar a skill como código.

Cinco seções, e o teste SFA que controla a produção

Todo SKILL.md carrega cinco seções, e pular uma delas é onde a maioria das skills falha na revisão. O arquivo abre com frontmatter YAML que declara name, description, globs e alwaysApply. A seção um enuncia a descrição e o propósito. A seção dois lista as regras de domínio: as restrições inegociáveis, as convenções de nomenclatura e os anti-padrões que guardam o conhecimento duro do domínio. A seção três é um fluxo de trabalho numerado, passo a passo, explícito o suficiente para que outro engenheiro ou um agente consiga reproduzi-lo, com referências às ferramentas e aos scripts que ele chama. A seção quatro traz os templates de saída, exemplos concretos de formato, estrutura e nomenclatura de arquivos, para que o agente siga um template em vez de inventar um. A seção cinco é um checklist de validação com critérios de aprovação ou reprovação aplicados antes da entrega. Essa última seção é um quality gate embutido na própria skill: se algum item falhar, o agente revisa antes de responder.

Antes de chegar à produção, uma skill precisa passar no teste SFA. SFA significa Scope, Format e Audience, e os três precisam estar explícitos no arquivo. Scope diz

o que a skill cobre e, tão importante quanto, o que ela recusa. Format fixa o formato da saída. Audience nomeia para quem a saída existe. Eu trato o teste SFA como o portão de entrada porque uma skill que não responde a essas três perguntas é uma skill que vai derivar. O retorno é mensurável. No FeatureBench, que representa trabalho real de feature em vez de correções isoladas de bug, um modelo de fronteira no picker do Copilot como o Claude Opus 4.8 resolve cerca de 11 por cento das tarefas sem contexto de domínio. Com contexto de skill, ou seja, padrões de arquitetura, convenções de código e vocabulário de domínio, esse número sobe para cerca de 38 por cento. As cinco seções são como esse contexto é codificado.

Três modos de ativação, e o custo em tokens de cada um

Uma skill tem três formas de entrar no contexto, e cada uma troca conveniência por custo. O glob match é automático: um arquivo entra no conjunto de trabalho, seu caminho corresponde ao padrão glob que a skill declara e a skill é carregada sem ação humana. É o padrão eficiente para especialização por tipo de arquivo. O alwaysApply definido como true carrega a skill em toda interação, independentemente do contexto. Serve para preocupações transversais como regras de marca, padrões de segurança e convenções de código compartilhadas, as coisas que deveriam se aplicar em todo lugar. A invocação explícita é o terceiro modo: o orquestrador ou o desenvolvedor referencia a skill pelo nome, sob demanda. É preciso e serve para tarefas compostas que reúnem várias skills de uma vez, mas depende de quem chama saber que a skill existe.

A escolha importa porque a ativação é onde o custo em tokens é decidido. A telemetria de produção do GitHub Copilot Enterprise no primeiro trimestre de 2026 mostra que um generalista lidando com uma tarefa que uma skill especializada poderia responder diretamente consome cerca de três vezes mais tokens de contexto por tarefa. Em um único desenvolvedor, isso é ruído. Numa equipe, é uma linha real de custo. O alwaysApply é o modo a vigiar: ele se aplica de forma ampla e cobra de forma ampla, então eu o reservo para as poucas preocupações que de fato pertencem a toda chamada. Para o resto, applyTo e padrões glob delimitam a skill aos arquivos e pastas onde ela paga seus tokens, e a nenhum outro. Uma hora gasta desenhando essa delimitação, em vez de carregar tudo em todo lugar, pode reduzir o gasto médio de tokens em cerca de 30 por cento.

O ciclo de vida: fases essencial, opcional e de qualidade

Construir uma skill passa por três fases, e só a primeira é obrigatória. A fase essencial tem três passos. Passo um, identificar e organizar: definir o propósito, decidir se a skill é reutilizável entre projetos ou específica de um, estabelecer uma convenção de nomenclatura e confirmar que ela passa no teste SFA. Passo dois, criar a estrutura: escrever o SKILL.md com seu frontmatter e suas cinco seções, e adicionar exemplos para maior clareza. Passo três, estimar a distribuição: decidir a visibilidade, no workspace inteiro, com escopo de pasta ou específica do projeto, e configurar applyTo para que os arquivos certos acionem a skill. Qualquer skill que completa esses três passos é utilizável.

A fase opcional adiciona poder. Você pode anexar scripts de automação em Python, Bash ou Node, com validação de argumentos e tratamento de erros adequados, e pode enriquecer a skill com documentos de referência, exemplos reais e links para skills relacionadas. A fase de qualidade é onde a skill conquista a confiança de produção. Você valida e testa de ponta a ponta com prompts reais, confirma que a saída corresponde aos templates e executa cada item do checklist. Depois distribui para o escopo dela e confirma que outros agentes conseguem descobrir e invocar a skill, e que a ativação por glob e applyTo dispara como esperado. Da ideia à distribuição não é uma linha reta. É o essencial primeiro, depois o poder, depois a prova.

O modelo de maturidade de cinco níveis: uma skill é código

As skills amadurecem em cinco níveis, e saber onde uma skill está diz o que ainda falta a ela. O nível um é um SKILL.md de cerca de quarenta linhas que cobre descrição e fluxo de trabalho. Funciona, e é frágil. O nível dois adiciona referências e completa os templates de saída. O nível três é o alvo para a maioria das skills enterprise: todas as cinco seções, scripts de automação, referências a portais, conteúdo enriquecido e padrões glob ajustados. O nível quatro adiciona testes automatizados e um gate de CI, para que uma mudança que quebra a skill falhe no build. O nível cinco é uma entrada com versão fixada em um registro cross-

workspace com promoção automatizada via CI. Nesse ponto a skill é um ativo de software de primeira classe, não uma anotação em um editor.

A maioria das equipes entrega skills de nível um que funcionam, e o custo aparece depois. Skills de nível um se proliferam sem promoção, e você acaba com saídas inconsistentes, conhecimento de domínio duplicado e nenhum quality gate em lugar algum. O modo de falha tem nome: skills editadas in place, nunca versionadas, em que a qualidade da saída muda silenciosamente, sem rollback e sem changelog. A correção é tratar todo SKILL.md como código. Versione, revise em um pull request, teste e promova pelos níveis. Uma skill não é documentação que por acaso mora em um repositório. É um ativo que entrega comportamento, e comportamento que é entregue sem controle de versão é comportamento em que você não pode confiar.

> DEFINIÇÃO
>
> Trate todo SKILL.md como código, não como uma anotação. Cinco seções que passam no teste SFA, um modo de ativação escolhido pelo seu custo em tokens e um nível de maturidade que você consegue nomear. Versione, revise, teste, promova. Uma skill que entrega comportamento sem controle de versão é comportamento em que você não pode confiar.

Referências

Fontes primárias para a anatomia da skill, a ativação, o ciclo de vida e o modelo de maturidade.

- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, o benchmark por trás do gap de contexto de skill: cerca de 11 por cento sem contexto de domínio, cerca de 38 por cento com ele.
- Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, o caso empírico para validar e versionar skills antes da distribuição.
- Customize AI in Visual Studio Code, referência para ativação por glob, applyTo e arquivos de instrução no GitHub Copilot.
- GitHub Copilot Documentation, a referência de plataforma para agent mode, skills e custom agents.

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 ROTEANDO O TRABALHO

Dois tipos de agente e a árvore de decisão: mande cada tarefa para o generalista ou para um especialista.

O GitHub Copilot te dá dois tipos de agente: um orquestrador generalista, que resolve a maior parte do trabalho, e custom agents, cada um dono de um domínio bem delimitado. Escolher entre eles a cada tarefa é uma decisão de roteamento, e rotear no feeling é onde custo e qualidade derivam sem ninguém perceber. Este capítulo define os dois tipos, o contrato de seis seções que torna um custom agent seguro e a árvore de quatro perguntas que resolve cada decisão de roteamento em menos de um minuto.

Dois tipos de agente, e a diferença que decide o roteamento

O runtime do GitHub Copilot opera com dois tipos de agente, e toda decisão de roteamento começa por distinguir um do outro. Um agente generalista é o orquestrador padrão. Ele resolve de 70% a 80% do trabalho em um workspace maduro, e fica mais afiado quando você carrega um SKILL.md que traz o conhecimento de domínio que falta a ele. Usa ferramentas compartilhadas no nível do workspace e não carrega arquivo de identidade próprio. É a escolha certa para a maioria das tarefas de desenvolvimento: qualquer coisa que se beneficie de conhecimento de domínio injetado sem exigir ferramentas exclusivas nem uma identidade operacional distinta.

Um custom agent é um componente de software, não um prompt. Ele é definido por um arquivo .agent.md que fixa a identidade, as conexões exclusivas de ferramentas, o escopo delimitado, os contratos de entrada e saída e as regras de escalonamento. Custom agents conectam a MCP servers, CLIs e APIs que não são compartilhados com outros agentes. Eles cuidam dos 20% a 30% das tarefas que exigem acesso exclusivo a ferramentas, um tom distinto (uma revisão de segurança rigorosa lê

diferente de uma documentação amigável) ou um lugar em um pipeline de múltiplos estágios com handoffs tipados.

A diferença de capacidade é real, e é por isso que a divisão importa. No SWE-bench, um modelo de fronteira resolve cerca de 74% das correções de bug com escopo bem apertado. No FeatureBench, que mede trabalho real de feature, o mesmo modelo sem contexto de domínio cai para cerca de 11%, e o contexto de skill o eleva de volta para perto de 38% (FeatureBench, arXiv 2026). As skills fecham a maior parte dessa lacuna. Custom agents com ferramentas dedicadas fecham o resto. A proporção que sinaliza um workspace saudável é de cerca de um custom agent para cada oito a doze skills.

> DEFINIÇÃO
>
> A proporção saudável é de aproximadamente um custom agent para cada oito a doze skills. Acima disso, você está superengenheirando especialistas onde uma skill bastaria. Abaixo disso, você está subescrevendo skills e o generalista fica chutando conhecimento que ninguém escreveu. Generalista para 70% a 80% do trabalho, apoiado por skills; custom agents para os 20% a 30% que exigem ferramentas exclusivas, uma identidade distinta ou handoffs de pipeline.

O contrato de seis seções do .agent.md

Um custom agent é definido inteiramente pelo seu arquivo .agent.md, e esse arquivo é um contrato de seis seções com o orquestrador. A seção um é identidade e persona: nome, papel, tom, domínio. A seção dois é o manifesto de ferramentas: os MCP servers, CLIs e APIs exclusivos que este agente pode chamar, e nada fora disso roda, mesmo que o corpo mencione. A seção três é escopo e fronteiras: o que o agente resolve e o que ele recusa de imediato. A seção quatro é o contrato de entrada e saída: o payload de handoff que ele aceita e o artefato tipado que devolve. A seção cinco lista as skills que ele carrega para conhecimento de domínio. A seção seis são as regras de escalonamento: as condições que devolvem a tarefa ao orquestrador sem conclusão.

Quatro agentes de produção

Quatro agentes de workspaces reais mostram o formato. Um security agent tem acesso exclusivo a um Vault MCP server, ao Semgrep CLI e à API do Snyk; devolve um relatório de segurança com scores CVSS; escala em um CVE crítico ou uma exposição de credencial. Um test agent tem um GitHub Actions MCP server, o Jest e uma ferramenta de cobertura; devolve uma suíte de testes mais os resultados de execução; escala em testes instáveis. Um modernization agent tem um Terraform MCP server e o Azure SDK; devolve um plano de migração e um diff; escala em qualquer risco de perda de dados. Um documentation agent tem um Confluence MCP server e o guia de estilo da marca; devolve documentação estruturada e publicada; escala no instante em que uma tarefa vira revisão de código.

Duas propriedades tornam esses agentes seguros. Cada um é dono de um único domínio delimitado, então o primeiro incidente fica contido a esse domínio, e não ao workspace inteiro. E cada um nomeia seu gatilho de escalonamento, então devolve o controle em vez de improvisar além da própria competência. O manifesto de ferramentas é onde mora o menor privilégio. A regra de escalonamento é onde mora a honestidade. Em um estudo de caso instrumentado de um sistema C# de 108.000 linhas construído ao longo de 283 sessões, os agentes especialistas tinham em média cerca de 700 linhas nos papéis de maior capacidade e cerca de 330 nos padrão, e mais da metade de cada spec era conhecimento de domínio do projeto, não instrução comportamental (Vasilopoulos, arXiv 2026).

> DEFINIÇÃO
>
> Seis seções, em ordem: identidade e persona, manifesto de ferramentas, escopo e fronteiras, contrato de entrada e saída, skills carregadas, regras de escalonamento. O manifesto de ferramentas impõe o menor privilégio. A regra de escalonamento impede que um agente improvise além do próprio escopo. Um agente sem lista explícita de ferramentas e sem gatilho explícito de escalonamento não é um contrato, é um passivo.

A árvore de decisão de quatro perguntas, resolvida em menos de um minuto

Roteamento não precisa de intuição. Precisa de quatro perguntas, feitas em ordem, uma vez por tarefa. Primeira: a tarefa exige acesso exclusivo a ferramentas, um MCP server, uma API ou um CLI dedicado? Se sim, roteie para um custom agent mais uma skill. Segunda: a tarefa exige uma identidade ou um tom distintos, do jeito que um revisor de segurança rigoroso difere de um redator de documentação amigável? Se sim, custom agent mais skill. Terceira: é um trabalho de múltiplos estágios com handoffs entre agentes? Se sim, um pipeline de custom agents. Quarta, só se as três anteriores forem não: já existe um SKILL.md que cobre este domínio? Se sim, generalista mais skill. Se não, generalista sozinho, ou escreva a skill primeiro.

A árvore chega a três resultados. Custom agent mais skill, para o trabalho que precisa do próprio .agent.md com identidade única, ferramentas exclusivas e escopo definido: um security agent com um Vault MCP server, um test agent com ferramentas de CI, um modernization agent com Terraform. Generalista mais skill, quando uma skill já carrega o conhecimento e nenhuma identidade especializada é necessária: escrever um DOCX, gerar um diagrama, analisar um CSV. Generalista sozinho, para o trabalho que não exige conhecimento especializado nenhum: uma pergunta factual, um cálculo rápido, um brainstorm. A avaliação leva bem menos de um minuto por tarefa. O ponto é consistência: o mesmo tipo de tarefa roteia do mesmo jeito toda vez, para cada pessoa do time.

> DEFINIÇÃO
>
> Quatro perguntas, três resultados, menos de um minuto por tarefa. Ferramentas exclusivas, uma identidade distinta ou handoffs de pipeline mandam a tarefa para um custom agent mais uma skill. Um domínio coberto a manda para o generalista mais uma skill. Nada de especial a manda para o generalista sozinho. Rotear por uma árvore, e não pelo feeling, é o que torna custo e qualidade repetíveis.

Custom agents no GitHub Copilot: .github/agents, agent mode e o picker

No GitHub Copilot, o contrato mora no repositório. Custom agents são arquivos Markdown com frontmatter YAML em .github/agents/, cada um nomeado com a extensão .agent.md. O frontmatter declara o nome do agente, sua descrição, seu array de ferramentas permitidas e o modelo em que ele roda. O corpo é o system prompt: identidade, a lista ordenada do que o agente procura e o formato de saída. Um agente fica ativo por vez, e quando você o invoca em agent mode ele substitui a persona de chat padrão e trabalha apenas com as ferramentas que você permitir. As regras de todo o repositório em .github/copilot-instructions.md continuam valendo por baixo, então um agente herda a base do workspace e acrescenta a especialização por cima.

Escolher o modelo, e pagar por ele

O array de ferramentas é onde o menor privilégio é imposto, e o campo de modelo é onde você escolhe o motor. O picker do Copilot expõe vários modelos, e a mentalidade de roteamento continua dentro do agente. Vá de Claude Opus 4.8 ou Claude Fable 5 no raciocínio mais difícil: o refactor de múltiplos arquivos, a revisão de segurança que precisa estar certa. Sonnet 5 é o padrão equilibrado para a maior parte do trabalho diário de agente. Haiku 4.5 é a opção rápida e de baixo custo para tarefas de alto volume e escopo apertado, em que a latência importa mais que a profundidade. Gemini 3.1 Pro e os modelos GPT-5.x estão ali quando a tarefa combina com as forças deles. Combine o modelo à tarefa do mesmo jeito que combinou o agente à tarefa: capacidade onde ela compensa, velocidade e custo onde não compensa.

A cobrança segue essa escolha. O GitHub Copilot mede o trabalho de agente em GitHub AI Credits, em que um crédito equivale a um centavo de dólar, em vigor desde 1º de junho de 2026. Um modelo mais pesado em uma tarefa mais pesada gasta mais créditos, então a árvore de roteamento e o picker de modelo são a mesma alavanca de controle de custo vista de dois ângulos. Uma hora gasta montando a árvore de quatro perguntas pode cortar o gasto médio de tokens em cerca de 30% e estabilizar a qualidade da saída no time inteiro, que é o trabalho de infraestrutura mais barato que a maioria dos times tem à mão.

DEFINIÇÃO

Custom agents moram em .github/agents/*.agent.md: frontmatter para nome, ferramentas e modelo; corpo para o system prompt. O agent mode troca para essa persona enquanto o .github/copilot-instructions.md permanece por baixo. Escolha o modelo por tarefa no picker do Copilot: Opus 4.8 ou Fable 5 para o raciocínio mais difícil, Sonnet 5 como padrão equilibrado, Haiku 4.5 para trabalho rápido de alto volume, Gemini 3.1 Pro e GPT-5.x onde encaixarem. O Copilot mede tudo em GitHub AI Credits, a um crédito por centavo de dólar.

Referências

Fontes primárias para os dois tipos de agente, a árvore de roteamento, o contrato .agent.md e a seleção de modelo no GitHub Copilot.

- GitHub Copilot documentation, a plataforma de referência para agent mode, custom agents e a superfície de configuração que este capítulo roteia.
- GitHub Docs, Choosing the right AI model for your task, a orientação oficial por trás de casar um modelo do picker do Copilot com o trabalho à sua frente.
- Customize AI in Visual Studio Code, como agentes, skills e prompt files são criados e invocados no editor.
- FeatureBench, Benchmarking Agentic Coding for Complex Feature Development, a diferença de capacidade entre correções de bug bem delimitadas e trabalho real de feature que motiva rotear para especialistas.
- Vasilopoulos, Codified Context Infrastructure, o estudo de caso instrumentado por trás da contagem de linhas dos agentes especialistas e da parcela de conhecimento de domínio de cada spec.

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 EXECUÇÃO E GOVERNANÇA

Orquestração, handoffs e trilhos.

Um workspace do GitHub Copilot em escala não é um agente respondendo a uma pergunta. É um orquestrador que recebe cada requisição, roteia e coordena uma cadeia de especialistas por trás de uma única conversa. Este capítulo descreve o runtime que faz isso funcionar: quatro fases da entrada à entrega, handoffs que acumulam contexto em vez de perdê-lo, hooks que aplicam política de forma determinística, e uma hierarquia de configuração que decide quem pode definir qual regra. Os agentes fazem o trabalho. Os trilhos decidem se esse trabalho é seguro para colocar em produção.

O runtime de quatro fases, e o único agente com quem o usuário fala

Cada requisição em um workspace do Copilot passa por quatro fases. Entrada: o orquestrador recebe o prompt, carrega a própria identidade em .agent.md e classifica o tipo de tarefa. Análise e roteamento: ele busca um SKILL.md correspondente e avalia a árvore de decisão, escolhendo entre um agente generalista com uma skill e um custom agent com ferramentas dedicadas. Execução: as skills carregam, o .github/copilot-instructions.md e qualquer .instructions.md com escopo se aplicam automaticamente, as ferramentas MCP conectam, e a tarefa roda com contexto completo. Validação e entrega: a saída é verificada contra o checklist da própria skill, um quality gate revisa formato, conteúdo e convenções, e só então a resposta chega ao usuário.

A decisão de roteamento acontece uma vez, na segunda fase, não a cada turno. O que importa mais do que as fases é o formato que elas impõem: o orquestrador é o único agente com quem o usuário fala. Todo sub-agente roda por trás dele. Todo handoff retorna ao orquestrador, que decide o próximo passo. Os agentes nunca falam diretamente entre si. Essa única regra é o que mantém um sistema multi-agente depurável. Quando um resultado sai errado, há um único lugar para olhar, um único contexto para inspecionar e um único responsável por montar a resposta final.

No model picker do Copilot, o orquestrador também pode rotear por custo, mandando varreduras baratas para o Haiku 4.5, geração para o Sonnet 5 e análise difícil para o Opus 4.8.

Handoffs que acumulam contexto, sem perdê-lo

O padrão multi-agente mais claro é um pipeline de quatro estágios: Plan, Code, Test, Security. O agente de planejamento recebe a requisição do usuário e devolve um plano estruturado com critérios de aceite. O agente de código recebe a requisição mais o plano e devolve a implementação com a lista de arquivos que tocou. O agente de teste recebe a requisição, o plano e o código, e devolve testes com seus resultados de execução. O agente de segurança recebe o quadro completo, requisição, plano, código e resultados de teste, e devolve um relatório de achados. Cada estágio enxerga tudo o que os estágios anteriores produziram.

Essa é a propriedade que faz o pipeline funcionar. O contexto se acumula em cada handoff. Um handoff passa não só a nova entrada, mas todos os artefatos anteriores, de modo que nenhum estágio parte do zero e nenhuma decisão anterior é descartada em silêncio. O agente de segurança só consegue fazer o seu trabalho se enxerga o plano que o código deveria implementar e os testes que deveriam cobri-lo. Tire esse histórico e a revisão de segurança vira adivinhação.

Artefatos tipados, ordem estrita

Duas restrições mantêm a cadeia honesta. Cada agente emite um artefato tipado, um plano, um diff, uma suíte de testes, um relatório, que vira a entrada do próximo estágio, de modo que um handoff é um contrato, não uma conversa. E a sequência é estritamente ordenada: cada agente termina antes que o próximo comece. O orquestrador guarda o contexto acumulado e entrega a fatia certa ao especialista certo. Os hooks envolvem cada handoff e registram o que cruzou a fronteira, de modo que a cadeia inteira pode ser auditada depois.

Hooks como gates determinísticos

Skills e agents moldam o que o modelo tende a fazer. Os hooks decidem o que de fato acontece, independentemente de como o modelo se comporta. Eles vivem em .github/hooks/*.json e disparam em seis eventos de ciclo de vida do GitHub Copilot: sessionStart quando uma sessão começa ou é retomada, userPromptSubmitted quando o usuário envia um prompt, preToolUse antes de o agente usar qualquer ferramenta, postToolUse depois que uma ferramenta termina, sessionEnd quando a sessão encerra, e errorOccurred quando algo falha. Só o preToolUse pode bloquear: ele pode negar uma chamada de ferramenta antes que ela rode. Os outros cinco observam, registram e reagem. Hooks de vários arquivos e plugins se combinam em vez de sobrescrever, e para o coding agent na nuvem os arquivos de hook precisam estar no branch padrão do repositório para carregar.

Quatro hooks pertencem a todo rollout no primeiro dia, porque o modelo não vai adicioná-los por você. Um estudo empírico de 2.303 arquivos de contexto de agente em 1.925 repositórios encontrou orientação de segurança explícita em apenas 14,5% deles (Chatlatanagulchai et al. 2025). Os hooks são o anteparo determinístico para os 85,5% que deixam isso de fora. Um secrets block no preToolUse nega qualquer escrita ou commit que carregue uma credencial. Um formatter no postToolUse roda o linter em cada arquivo que o agente edita, de modo que estilo nunca vire uma discussão de review. Um audit log, disparado por sessionStart e postToolUse, registra cada ação, versão de modelo e justificativa, de modo que uma mudança possa ser explicada e revertida. Um policy gate no preToolUse nega comandos destrutivos, o rm, o DROP TABLE, o kubectl delete contra produção, e chama um humano no lugar.

> DEFINIÇÃO
>
> Os quatro hooks do primeiro dia: um secrets block e um policy gate no preToolUse que negam antes do dano, um formatter no postToolUse que encerra as discussões de estilo, e um audit log em sessionStart e postToolUse que torna cada ação de agente explicável e reversível. Trilhos determinísticos, não boa vontade do modelo.

Governança de configuração: pisos, contratos e um deny absoluto

O último trilho é quem pode definir qual regra, e o modelo é unidirecional por desenho. A organização define os pisos por meio das políticas do GitHub Enterprise Copilot: exclusão de conteúdo, telemetria, acesso a modelos e aplicação de hooks que um único repositório não pode afrouxar. O workspace define o contrato do time em .github: copilot-instructions.md, .instructions.md com escopo, hooks, e o .vscode/settings.json que todo clone herda. O usuário personaliza por cima, preferências de editor e configurações locais que nunca vazam para um commit. Cada camada refina a de cima. Nenhuma pode contradizê-la.

Uma regra está acima de todas as outras: o deny é absoluto. Uma camada mais baixa pode adicionar permissão onde recebeu espaço, mas nunca pode sobrescrever um deny definido acima. Uma skill não pode furar uma política. Um prompt não pode reabilitar um comando que a organização bloqueou. Uma configuração de usuário não pode enfraquecer um guardrail de workspace que o time comitou. É isso que faz da hierarquia um modelo de governança, e não uma pilha de arquivos de config. Também mantém os agentes prestando contas quando agem sobre dinheiro: as execuções de agente são medidas em GitHub AI Credits, em que um crédito equivale a um centavo de dólar desde junho de 2026, então um teto por módulo interrompe um loop descontrolado antes que a fatura o faça, e o MCP expõe os sistemas corporativos por uma única interface auditada, em vez de credenciais espalhadas.

Referências

Fontes primárias para o runtime de orquestração, o ciclo de vida de hooks do Copilot, a evidência empírica da lacuna de segurança e o modelo de governança de configuração deste capítulo.

- GitHub Copilot, Hooks configuration reference, os seis eventos de ciclo de vida e o schema JSON que todo arquivo de hook segue.
- GitHub Copilot, Using hooks with the coding agent, por que os arquivos de hook precisam estar no branch padrão para carregar no agente da nuvem.

- GitHub Copilot in VS Code, Agent hooks, o modelo local de hooks para gates de preToolUse e postToolUse.
- GitHub Copilot, Repository custom instructions, o copilot-instructions.md e os arquivos de instrução com escopo que formam o contrato do time.
- Chatlatanagulchai et al., An Empirical Study of Agent Context Files, a constatação de que apenas 14,5% de 2.303 arquivos em 1.925 repositórios trazem orientação de segurança explícita.
- GitHub Copilot documentation, a plataforma de referência para o agent mode, o model picker e o MCP.
- Model Context Protocol Specification, o padrão aberto para expor sistemas corporativos aos agentes por uma única interface auditada.

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 ADOÇÃO E ENCERRAMENTO

Anti-padrões, diagnósticos e 90 dias: da teoria de roteamento a uma prática repetível.

Concordar com a teoria de roteamento é fácil. Mantê-la é difícil. Este capítulo é a camada de adoção: os quatro anti-padrões que aparecem antes de qualquer boa prática se consolidar, seis perguntas diagnósticas que revelam lacunas em qualquer workspace, e um plano de 90 dias que transforma a árvore de decisão de quatro perguntas em hábito. Ele fecha com o único número que paga por tudo isso.

Quatro anti-padrões que aparecem antes de qualquer boa prática

Todo workspace que vi escalar o GitHub Copilot passou primeiro pelos mesmos quatro anti-padrões. O primeiro é a skill como megaprompt: um SKILL.md cresce até duas mil linhas, cobre o domínio inteiro e ativa em cada glob match. O overhead de tokens por chamada passa a superar o valor do conhecimento que ele carrega, agora uma linha legível em GitHub AI Credits. A solução é dividir em três skills focadas, cada uma estreita o suficiente para passar no teste SFA de Scope, Format e Audience. O segundo é o agente para tudo: um único custom agent que detém todas as permissões e não recusa nenhuma tarefa. Funciona até o primeiro incidente, quando o raio de impacto é o workspace inteiro. Um custom agent precisa de um escopo delimitado no seu .agent.md e de um manifesto de ferramentas que ele não ultrapassa, porque em agent mode tudo o que aquele agente alcança via MCP é o que um atacante herda se ele for comprometido.

O terceiro é as skills nunca versionadas: arquivos SKILL.md editados in place, qualidade das saídas oscilando, sem changelog e sem rollback quando uma mudança piora os resultados. Trate todo SKILL.md como código: versione, revise em um pull request, teste e promova. O quarto é o roteamento por intuição, em que cada desenvolvedor decide por chamada, no feeling, se uma tarefa precisa de um custom

agent, de uma skill ou de nenhum. O resultado é saída inconsistente, gasto de AI Credits inconsistente e nenhuma baseline para medir. A árvore de decisão de quatro perguntas elimina o achismo, e leva cerca de uma hora para ser acordada pela equipe. Essas falhas são baratas de prevenir e caras de desfazer, e por isso pertencem ao início de qualquer plano de adoção.

> DEFINIÇÃO
>
> Os quatro anti-padrões, na ordem em que surgem: skill como megaprompt, agente para tudo, skills nunca versionadas e roteamento por intuição. O primeiro infla os AI Credits, o segundo o raio de impacto, o terceiro apaga a trilha de auditoria, e o quarto elimina a baseline.

Seis perguntas diagnósticas para qualquer workspace

Antes de planejar qualquer mudança, meça onde você está. Seis perguntas revelam as lacunas em qualquer workspace Copilot, e um não a qualquer uma delas vale ser endereçado antes da próxima sprint. Você conhece todas as skills ativas no seu workspace agora? Todo custom agent tem um .agent.md com lista explícita de ferramentas? Você consegue rastrear qual skill foi carregada para qualquer saída de tarefa? As três primeiras perguntas são sobre visibilidade. Se você não consegue respondê-las, está operando sem instrumentação, e nenhuma otimização sobrevive ao contato com um sistema que você não enxerga.

As três últimas são sobre disciplina. Existe uma árvore de decisão de roteamento que toda a equipe segue, ou cada membro decide de forma ad hoc? Todas as skills são versionadas e revisadas antes de serem distribuídas? E o único número que vale checar diretamente: a proporção de custom agents para skills está em 1:8 ou melhor? Mais custom agents que isso costuma significar um workspace sobre-engenheirado, pagando por identidades sob medida onde uma skill compartilhada no agente generalista faria o mesmo trabalho por uma fração dos AI Credits. Muito menos skills que agentes significa subespecialização, forçando custom agents a carregar conhecimento que pertence a um SKILL.md reutilizável. A proporção de 1:8 é a leitura mais rápida de qualquer um dos dois desvios.

O playbook de 90 dias: Fundação, Contrato, Medição

As equipes que escalam o valor do Copilot seguem a mesma progressão de três fases, e cada fase termina com um resultado concreto. A Fundação vai do dia 1 ao dia 30: faça um inventário completo de skills e pontue cada SKILL.md contra o teste SFA, adote a árvore de decisão de quatro perguntas como equipe, revise o .agent.md de todo custom agent e habilite o log de ativação de skills. Consolide as convenções do repositório em .github/copilot-instructions.md, para que o agente generalista comece toda tarefa com as mesmas regras de base. O único resultado da Fundação é uma baseline de roteamento: um registro escrito do que existe e de como o trabalho é atribuído.

O Contrato vai do dia 31 ao dia 60: promova suas cinco principais skills ao nível de maturidade 3, o nível que carrega as cinco seções, scripts e padrões glob ajustados, depois coloque o primeiro pipeline no ar em agent mode, com plan, code e test fazendo handoff por contratos tipados, e comece a medir os AI Credits por tarefa. O único resultado do Contrato é um primeiro pipeline entregando artefatos tipados, em que cada handoff acumula contexto em vez de perdê-lo. A Medição vai do dia 61 ao dia 90: publique uma biblioteca compartilhada de skills, aplique a árvore de decisão a pelo menos 80% das tarefas, acompanhe a eficiência de tokens semanalmente em AI Credits e documente um processo replicável para adicionar skills. O único resultado da Medição é uma biblioteca de skills operacional e qualidade mensurável. O objetivo dos 90 dias não é ter skills perfeitas. É ter um processo repetível para torná-las melhores.

O que uma hora de design de roteamento devolve

Este é o número que paga o exercício inteiro. Escrever a árvore de decisão de quatro perguntas leva cerca de uma hora. Nos workspaces que a adotam, o gasto médio de tokens por tarefa cai cerca de 30%, porque tarefas que uma skill no agente generalista consegue responder deixam de ser roteadas para um custom agent que puxa três vezes mais contexto. No GitHub Copilot essa economia é denominada em GitHub AI Credits, em que um crédito equivale a um centavo de dólar desde 1 de

junho de 2026. Uma redução de 30% no gasto de tokens é uma redução de 30% na linha de AI Credits, legível no painel de uso.

O segundo retorno é a consistência, mais difícil de reduzir a um número e mais fácil de sentir. Quando o roteamento é deliberado, o mesmo tipo de tarefa segue o mesmo caminho toda vez, carrega a mesma skill e passa pelo mesmo checklist de validação antes da entrega. A saída deixa de depender de qual desenvolvedor formulou o prompt, e a carga de revisão cai porque os revisores encontram menos surpresas. Uma hora de design compra as duas coisas: um corte mensurável em AI Credits e uma saída mais previsível em toda a equipe. Este é o encerramento deste playbook. O roteamento é a primeira decisão de engenharia que você toma com o GitHub Copilot, e a que continua pagando.

> DEFINIÇÃO
>
> Uma hora de design de roteamento devolve cerca de 30% menos gasto de tokens, medido em GitHub AI Credits, e uma saída mais consistente em toda a equipe. O roteamento é a primeira decisão de engenharia que você toma com o GitHub Copilot, e a mais barata de acertar.

Referências

Fontes primárias para o modelo de roteamento, o gap de capacidade em trabalho de feature real, a superfície de segurança das skills de agente e os primitivos de customização do GitHub Copilot sobre os quais este capítulo se apoia.

- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, o gap de capacidade em trabalho de feature real que skills de domínio e custom agents delimitados existem para fechar.
- Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, a base empírica de por que o anti-padrão do agente para tudo é perigoso e de por que todo SKILL.md precisa de controle de versão e revisão.
- Customize AI in Visual Studio Code, a referência para os primitivos SKILL.md, .agent.md e .instructions.md sobre os quais este playbook faz o roteamento.

- GitHub Copilot documentation, a plataforma de referência para agent mode, conexões MCP e o modelo de cobrança em GitHub AI Credits.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps