La primitiva correcta para la tarea decide si un workspace escala o colapsa.

La mayoría de los equipos adopta GitHub Copilot como una única caja de chat, y se rompe en el segundo caso de uso. Este playbook trata el enrutamiento como la primera decisión de ingeniería: seis primitivas, agentes, skills, plugins, servidores MCP, hooks y la jerarquía de configuración, cada una con un límite que decide si el workspace escala. El retorno es medible, cerca de 30 por ciento menos tokens por tarea, ahora cobrados en GitHub AI Credits, y una salida que se mantiene consistente en todo el equipo.

AUTORA
Paula Silva

CARGO
AI-Native Software Engineer

CONTACTO
in/paulanunes

TIEMPO DE LECTURA
~ 45 minutos, de punta a punta

6
PRIMITIVAS CENTRALES

6
CAPÍTULOS

4
PREGUNTAS DE ENRUTAMIENTO

~30%

RECORTE DE GASTO
EN TOKENS

CAPÍTULOS

Chapter 00

El enrutamiento es la primera decisión de ingeniería

Por qué la caja de chat única se rompe en el segundo caso de uso, la brecha de capacidad de SWE-bench a FeatureBench en números, cuánto cuesta una tarea mal enrutada en GitHub AI Credits, y las seis primitivas cuyos límites deciden la escala.

El Problema →

Chapter 01

Cuatro puntos de control, un flujo de tarea

Cómo se ensambla una respuesta de Copilot a partir de una pila de archivos: la pila del workspace leída de abajo hacia arriba, los cuatro tipos de archivo centrales, las seis capas de prompt y la jerarquía de configuración de .github/ como una Piedra de Rosetta.

El Mapa →

Chapter 02

SKILL.md: anatomía, activación y madurez

Las cinco secciones y la prueba SFA, los tres modos de activación y su costo en tokens, el ciclo de vida de la idea a la distribución, y el modelo de madurez de cinco niveles que trata la skill como código.

Skills →

Chapter 03

Dos tipos de agente y el árbol de decisión

Generalista versus custom agents, la proporción saludable de un agente por cada ocho a doce skills, el contrato de seis secciones del .agent.md, el árbol de enrutamiento de cuatro preguntas y la elección del modelo en el picker de Copilot.

Agentes →

Chapter 04

Orquestración, handoffs y rieles

El runtime de cuatro fases con un orquestador, la cadena de handoffs Plan, Code, Test, Security que acumula contexto, los seis eventos de ciclo de vida de hooks y el modelo de gobernanza en el que el deny es absoluto.

Ejecución →

Chapter 05

Anti-patrones, diagnósticos y 90 días

Los cuatro anti-patrones que aparecen primero, seis preguntas diagnósticas, el playbook de adopción de 90 días, y lo que devuelve una hora de diseño de enrutamiento en GitHub AI Credits.

Adopción →

CONSEJOS DE LECTURA

> **ATAJOS DE TECLADO**
>
> Presiona `/` para buscar. Usa `j` y `k` para moverte entre capítulos.
> Presiona `t` para cambiar tema. Presiona `g` para ir al inicio.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 EL PROBLEMA

El enrutamiento es la primera decisión de ingeniería.

La mayoría de los equipos adopta GitHub Copilot como una única caja de chat. Cada tarea va al mismo agente generalista, cada respuesta se rearma desde el mismo context window, y el arreglo funciona hasta que llega el segundo caso de uso. Este capítulo sostiene que el enrutamiento, decidir qué tarea va a qué primitiva, es la primera decisión de ingeniería, no una optimización posterior. Le pone números a la brecha de capacidad, cotiza el costo de una tarea mal enrutada en GitHub AI Credits y nombra las seis primitivas cuyos límites deciden si un workspace escala.

La trampa de la caja de chat única: un agente, un context window, un cuello de botella

El error más común que veo cometer a los equipos con GitHub Copilot es tratarlo como una única interfaz de chat generalista. Cada tarea va al mismo agente en agent mode. Cada respuesta reconstruye el contexto desde la misma ventana. En un workspace maduro, un orquestador generalista resuelve cerca del 70% al 80% de las tareas, y eso está bien. La trampa es mandarle también el otro 20% al 30%, el trabajo que necesita una skill específica, una herramienta exclusiva o una identidad distinta. Un agente absorbe todo, y el context window se vuelve el cuello de botella.

Ese cuello de botella crece con el equipo, no con el individuo. Para un desarrollador, un context window algo inflado es ruido. Entre docenas de desarrolladores corriendo el mismo generalista en cada tarea, se vuelve un motor de costo medible. El context window tiene un presupuesto fijo. Cuando un solo agente carga reglas de marca, estándares de seguridad, vocabulario de dominio y la tarea actual al mismo tiempo, la señal útil queda desplazada. La salida se desvía, los revisores atrapan más errores, y la productividad que prometió la herramienta se estanca. La falla no es del modelo. Es del enrutamiento.

La brecha de capacidad en números: SWE-bench parece resuelto, las features reales no

La distancia entre un benchmark y una feature real es donde el argumento a favor del enrutamiento se vuelve concreto. En SWE-bench, un modelo de frontera resuelve el 74,4% de las correcciones de bug bien acotadas, las tareas con requisitos ajustados y aislados. Ese número pareciera decir que el problema está resuelto. En FeatureBench, que representa trabajo de feature del mundo real en lugar de parches localizados, el mismo modelo sin contexto de dominio cae al 11% (arXiv:2602.10975). El benchmark medía parches localizados, no desarrollo de features, así que los dos números describen dos trabajos distintos.

La parte recuperable de esa brecha es la parte interesante. Agrega contexto de skill, es decir, patrones de arquitectura, convenciones de código y vocabulario de dominio entregados por un SKILL.md, y el número en FeatureBench sube a cerca del 38%. Las skills cierran alrededor de 27 puntos porcentuales de la distancia. El resto no se cierra con un modelo más grande. Se cierra con custom agents con acceso dedicado a herramientas, las primitivas hacia las que este playbook construye. La lección es que el contexto, entregado por la primitiva correcta, mueve el número más que la capacidad bruta del modelo.

Cuánto cuesta una tarea mal enrutada, ahora cotizada en GitHub AI Credits

Una tarea mal enrutada no es solo de menor calidad. Es más cara, y ahora el costo viene detallado. La telemetría de producción de GitHub Copilot Enterprise en el primer trimestre de 2026 muestra que un agente generalista que atiende trabajo que una skill especializada respondería directo consume cerca de tres veces más tokens de contexto por tarea. Desde el 1 de junio de 2026, ese consumo se mide en GitHub AI Credits, donde un crédito equivale a un centavo de dólar. El desperdicio dejó de esconderse dentro de una suscripción fija. Aparece en la cuenta.

Multiplica tres veces los tokens por cada desarrollador que corre el generalista en trabajo con forma de skill, y el número deja de ser ruido. La misma telemetría apunta también hacia el otro lado: los equipos que invierten una hora en diseño de enrutamiento, adoptando un árbol de decisión simple en lugar de elegir por intuición,

recortan el gasto promedio de tokens en cerca del 30%. Ese es el argumento económico central de este playbook. El enrutamiento no es un impuesto sobre la velocidad. Es la diferencia entre GitHub AI Credits gastados razonando y GitHub AI Credits gastados reconstruyendo contexto que el workspace ya tenía.

Las seis primitivas, y por qué sus límites deciden la escala

El workspace moderno de GitHub Copilot no es un solo asistente. Es una composición de seis primitivas, cada una con su formato de archivo, su ciclo de vida y su contrato con las demás. Los agentes eligen el camino. Las skills empaquetan el conocimiento de dominio en un SKILL.md versionado. Los plugins alcanzan el mundo. Los servidores MCP exponen herramientas tipadas mediante el Model Context Protocol. Los hooks mantienen los rieles deterministas, disparando en eventos fijos del ciclo de vida. Y la jerarquía de configuración, anclada en archivos como .github/copilot-instructions.md, decide quién gana cuando dos de ellas no coinciden.

Los límites entre esas seis son lo que importa. Una skill es conocimiento, no una herramienta. Un agente es una identidad y un alcance, no un depósito de conocimiento. Un hook es un gate determinista, no un prompt. Cuando esos límites son explícitos, un workspace compone de forma limpia y escala de un desarrollador a la organización entera. Cuando se difuminan, todo colapsa de vuelta a la caja de chat única, y el workspace se rompe en el segundo caso de uso. Acertar los límites es todo el trabajo. El resto de este playbook los recorre uno por uno, primitiva por primitiva.

> DEFINICIÓN
>
> El enrutamiento en una frase: decide cuál de las seis primitivas es dueña de una tarea antes de enviarla, porque un generalista haciendo trabajo con forma de skill cuesta cerca de 3x los tokens de contexto, ahora cobrados en GitHub AI Credits, y la brecha de capacidad entre un benchmark resuelto y una feature real se cierra con la primitiva correcta, no con un modelo más grande.

Referencias

Fuentes primarias para la brecha de capacidad, la decisión de enrutamiento y las seis primitivas que este capítulo nombra.

- FeatureBench, Benchmarking Agentic Coding for Complex Feature Development, la brecha empírica: un modelo de frontera con 74,4% en SWE-bench resuelve el 11% de las tareas de feature real sin contexto de dominio.
- Agent Skills in the Wild, An Empirical Study of Security Vulnerabilities at Scale, contexto de por qué importan los límites y la gobernanza de skills a medida que las skills proliferan en un workspace.
- Customize AI in Visual Studio Code, la referencia para skills, custom agents, instructions y prompt files en el workspace.
- GitHub Copilot Documentation, la plataforma de referencia para agent mode, MCP y la jerarquía de configuración de este playbook.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 EL MAPA

Cuatro puntos de control, un flujo de tarea: cómo la pila del workspace ensambla una respuesta.

Cuando envías una tarea a GitHub Copilot en agent mode, la respuesta no la produce el modelo solo. La produce una pila de archivos que se ensamblan alrededor de tu pedido. Lee la pila de abajo hacia arriba y cuenta una sola historia: la skill carga el conocimiento de dominio, las herramientas se conectan a las capacidades externas, los hooks controlan lo que puede ejecutarse, y el agente responde con todo eso en contexto. Este capítulo mapea los cuatro puntos de control, las seis capas de prompt que alimentan y dónde vive cada primitivo en el repositorio.

La pila del workspace, leída de abajo hacia arriba

Leo un workspace de Copilot como leo una línea de fábrica. Empiezo por abajo y sigo el material hacia arriba. La skill carga primero: un SKILL.md trae el conocimiento de dominio que la tarea necesita, las convenciones y el workflow. Las herramientas se conectan después: servidores MCP, CLIs y APIs le dan al agente capacidades que no tiene por defecto. Los hooks controlan la ejecución: un hook preToolUse puede negar un comando peligroso antes de que se ejecute. El agente responde al final, con el conocimiento de dominio, las herramientas y los guardrails ya en su lugar. Cuatro capas, cuatro funciones, una respuesta.

El valor de cada capa se vuelve visible cuando se la omite. Omite la skill y el agente adivina las convenciones que la base de código ya codifica; la telemetría de producción de GitHub Copilot Enterprise en el primer trimestre de 2026 mostró que un agente generalista manejando una tarea que una skill especializada podría responder directamente consumió 3 veces más tokens de contexto para el mismo trabajo. Ese overhead no es abstracto. Llega a la cuenta como GitHub AI Credits, donde un crédito es un centavo de dólar. Omite las herramientas y el agente no

alcanza la base de datos o la API que necesita, así que fabrica una respuesta plausible en su lugar. Omite los hooks y nada queda entre un comando generado y tu repositorio. Omite la identidad del agente y un solo generalista carga cada tarea, sin alcance delimitado y sin herramientas exclusivas.

La capa de skill es donde se cierra la mayor brecha

La medición más clara de la capa de skill viene de dos benchmarks. En SWE-bench, que puntúa correcciones de bug precisas y aisladas, un modelo de frontera resuelve cerca del 74% de las tareas. En FeatureBench, que representa trabajo de feature real, el mismo modelo sin contexto de dominio cae al 11%. Con contexto de skill, los patrones de arquitectura, las convenciones de código, el vocabulario de dominio, la cifra sube a aproximadamente 38%. La capa de skill cierra cerca de 27 puntos porcentuales de esa brecha. La distancia restante es para lo que sirven los custom agents con herramientas dedicadas. Por eso vale leer la pila como pila: cada capa recupera una capacidad que la capa de abajo no alcanza.

Los cuatro tipos de archivo centrales

Cuatro tipos de archivo sostienen todo el ecosistema, y cada uno responde a una pregunta distinta. El primero son las instrucciones, y vienen en dos alcances. El archivo de workspace, .github/copilot-instructions.md, guarda las reglas que valen para todo el repositorio: el stack, las convenciones de nombres, los comandos. Las instrucciones con alcance viven en .github/instructions/ como archivos .instructions.md, cada uno con un glob applyTo que lo limita a una ruta como src/components/**/*.tsx. Las instrucciones responden a la pregunta de qué debe hacer el agente, y se aplican automáticamente, sin que el usuario lo pida. El agente no puede ignorarlas; funcionan como guardrails.

Los otros tres tipos de archivo responden a las preguntas restantes. Las herramientas, declaradas en .mcp.json, responden qué puede hacer el agente: un servidor MCP lo conecta a GitHub, a una base de datos o a un servicio interno; una CLI le da git o az; una API le da una llamada HTTP. El archivo .agent.md en .github/agents/ responde quién es el agente: su nombre y rol, las herramientas que puede usar, las tareas que acepta y rechaza, y las reglas de handoff para cuando delega. El archivo SKILL.md en .github/skills/ responde qué sabe el agente: los

templates, el workflow paso a paso y el checklist de validación para un tipo de salida. Las reglas restringen el comportamiento, las herramientas habilitan capacidad, la identidad delimita el alcance, el conocimiento eleva la calidad. Los cuatro son independientes y combinables, y ese es el punto.

Las seis capas de prompt que ensamblan el contexto final

Esos archivos no llegan al modelo como archivos. Se fusionan en un único contexto, ensamblado en seis capas. La base del modelo viene primero: el system prompt definido por GitHub y por el modelo que elegiste en el selector de Copilot, sea Claude Opus 4.8, Sonnet 5, Haiku 4.5, Claude Fable 5, Gemini 3.1 Pro o un modelo GPT-5.x. Las instrucciones de copilot-instructions.md se inyectan después, luego las instrucciones con alcance que coinciden con el archivo actual. La identidad del agente se carga del .agent.md activo. El conocimiento de la skill se agrega del SKILL.md activado. Tu prompt se coloca al final, como la entrada de mayor prioridad. Las capas posteriores refinan las anteriores, pero no las contradicen: una skill no anula una regla de seguridad, y un prompt no evade un comando negado.

La razón de seis capas separadas, alimentadas por archivos separados, es la evolución. Cada tipo de archivo apunta a una capa para poder cambiar por su cuenta. Puedes versionar un SKILL.md y promoverlo por un modelo de madurez sin tocar el agente que lo carga. Puedes ajustar un glob applyTo en una instrucción con alcance sin editar el archivo de workspace que lee cada desarrollador. Puedes cambiar el modelo en el selector sin reescribir una sola instrucción. La separación de responsabilidades a nivel de archivo se vuelve evolución independiente en tiempo de ejecución. Esa es la diferencia entre un workspace que puedes mantener y un megaprompt único que nadie se atreve a editar.

> DEFINICIÓN
>
> Las seis capas, en orden de ensamblaje: base del modelo, luego instrucciones de workspace, luego instrucciones con alcance, luego identidad del agente, luego conocimiento de la skill, luego tu prompt al final como la entrada de mayor prioridad. Cada capa refina la anterior, y ninguna puede contradecirla.

La jerarquía de configuración como Piedra de Rosetta

En GitHub Copilot, cada primitivo tiene un lugar fijo dentro de .github/, y la ubicación es el contrato. Las instrucciones de workspace viven en .github/copilot-instructions.md. Las instrucciones con alcance viven en .github/instructions/ como archivos .instructions.md con globs applyTo. Los agentes viven en .github/agents/ como archivos .agent.md. Las skills viven en .github/skills/ como archivos SKILL.md dentro de una carpeta con nombre. Los hooks viven en .github/hooks/ como archivos JSON que declaran handlers preToolUse, postToolUse, sessionStart y sessionEnd. Las plantillas de prompt reutilizables viven en .github/prompts/ como archivos .prompt.md. Las conexiones MCP viven en .mcp.json en la raíz. Los plugins empaquetan varios de estos juntos mediante un manifiesto plugin.json, distribuidos por el marketplace Awesome GitHub Copilot.

Llamo a esta tabla Piedra de Rosetta porque el mismo concepto existe en Claude Code y en GitHub Copilot con nombres y rutas distintas, y cuando puedes leer una columna, puedes leer la otra. El copilot-instructions.md es el equivalente de CLAUDE.md en Copilot. El .github/instructions/ es el equivalente de .claude/rules/. El .github/agents/ corresponde a .claude/agents/, y el .github/skills/ corresponde a .claude/skills/ con un formato SKILL.md idéntico. La única capa que no escribes es la base del modelo; la eliges en el selector y la pagas en AI Credits. Todo lo demás es un archivo que escribes, colocas en un lugar predecible y dejas que el orquestador descubra. Esa previsibilidad es lo que permite que un workspace escale más allá de una única ventana de chat.

Referencias

Fuentes primarias para los cuatro tipos de archivo, las capas de prompt y la jerarquía de configuración de GitHub Copilot.

- GitHub Copilot, Adding repository custom instructions, la referencia para copilot-instructions.md y archivos .instructions.md con alcance y globs applyTo.
- Customize AI in Visual Studio Code, la visión general de agents, prompt files y skills como primitivos de personalización separados.

- Prompt Files in VS Code, el formato para plantillas .prompt.md reutilizables con frontmatter de agent mode, modelo y herramientas.
- GitHub Copilot Hooks Configuration Reference, el esquema JSON y el modelo de eventos para hooks preToolUse, postToolUse, sessionStart y sessionEnd.
- Model Context Protocol Specification, el estándar abierto detrás de las conexiones de herramientas en .mcp.json.
- GitHub Copilot: Instructions vs Prompts vs Custom Agents vs Skills, una comparación lado a lado de cada primitivo de configuración de 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 EMPAQUETANDO CONOCIMIENTO

SKILL.md: anatomía, activación y madurez.

Un SKILL.md no es un prompt, no es un archivo de configuración y no es un script. Es un artefacto de conocimiento de dominio con una estructura fija, tres modos de activación y un camino de madurez que va de un borrador de cuarenta líneas a una entrada de registro con versión fija. Este capítulo desarma el artefacto. Cubre las cinco secciones que toda skill incluye, la prueba SFA que controla el uso en producción, cómo cada modo de activación decide el costo en tokens, el ciclo de vida de la idea a la distribución, y el modelo de cinco niveles que permite a un equipo tratar la skill como código.

Cinco secciones, y la prueba SFA que controla la producción

Todo SKILL.md incluye cinco secciones, y saltarse una es donde la mayoría de las skills falla en la revisión. El archivo abre con frontmatter YAML que declara name, description, globs y alwaysApply. La sección uno enuncia la descripción y el propósito. La sección dos lista las reglas de dominio: las restricciones no negociables, las convenciones de nomenclatura y los anti-patrones que guardan el conocimiento duro del dominio. La sección tres es un flujo de trabajo numerado, paso a paso, lo bastante explícito para que otro ingeniero o un agente pueda reproducirlo, con referencias a las herramientas y los scripts que invoca. La sección cuatro trae las plantillas de salida, ejemplos concretos de formato, estructura y nomenclatura de archivos, para que el agente siga una plantilla en lugar de inventar una. La sección cinco es una lista de verificación con criterios de aprobación o rechazo aplicados antes de la entrega. Esa última sección es un quality gate integrado en la propia skill: si algún elemento falla, el agente revisa antes de responder.

Antes de llegar a producción, una skill tiene que pasar la prueba SFA. SFA significa Scope, Format y Audience, y los tres deben estar explícitos en el archivo. Scope dice qué cubre la skill y, igual de importante, qué rechaza. Format fija la forma de la salida. Audience nombra para quién existe la salida. Trato la prueba SFA como la puerta de entrada porque una skill que no responde esas tres preguntas es una skill que va a derivar. El retorno es medible. En FeatureBench, que representa trabajo real de features en lugar de correcciones aisladas de bugs, un modelo de frontera en el picker de Copilot como Claude Opus 4.8 resuelve cerca del 11 por ciento de las tareas sin contexto de dominio. Con contexto de skill, es decir, patrones de arquitectura, convenciones de código y vocabulario de dominio, esa cifra sube a cerca del 38 por ciento. Las cinco secciones son cómo se codifica ese contexto.

Tres modos de activación, y el costo en tokens de cada uno

Una skill tiene tres formas de entrar al contexto, y cada una intercambia conveniencia por costo. El glob match es automático: un archivo entra al conjunto de trabajo, su ruta coincide con el patrón glob que la skill declara y la skill se carga sin acción humana. Es el modo por defecto eficiente para la especialización por tipo de archivo. El alwaysApply en true carga la skill en cada interacción, sin importar el contexto. Sirve para preocupaciones transversales como reglas de marca, estándares de seguridad y convenciones de código compartidas, las cosas que deberían aplicarse en todas partes. La invocación explícita es el tercer modo: el orquestrador o el desarrollador referencia la skill por su nombre, bajo demanda. Es precisa y sirve para tareas compuestas que reúnen varias skills a la vez, pero depende de que quien la invoca sepa que la skill existe.

La elección importa porque la activación es donde se decide el costo en tokens. La telemetría de producción de GitHub Copilot Enterprise en el primer trimestre de 2026 muestra que un generalista manejando una tarea que una skill especializada podría responder directamente consume cerca de tres veces más tokens de contexto por tarea. En un solo desarrollador, eso es ruido. En un equipo, es una línea real de costo. El alwaysApply es el modo a vigilar: se aplica de forma amplia y factura de forma amplia, así que lo reservo para las pocas preocupaciones que de verdad pertenecen a cada llamada. Para el resto, applyTo y los patrones glob delimitan la skill a los archivos y carpetas donde paga sus tokens, y a ningún otro. Una hora

invertida en diseñar esa delimitación, en lugar de cargar todo en todas partes, puede reducir el gasto promedio de tokens en cerca del 30 por ciento.

El ciclo de vida: fases esencial, opcional y de calidad

Construir una skill pasa por tres fases, y solo la primera es obligatoria. La fase esencial tiene tres pasos. Paso uno, identificar y organizar: definir el propósito, decidir si la skill es reutilizable entre proyectos o específica de uno, establecer una convención de nomenclatura y confirmar que pasa la prueba SFA. Paso dos, crear la estructura: escribir el SKILL.md con su frontmatter y sus cinco secciones, y agregar ejemplos para mayor claridad. Paso tres, estimar la distribución: decidir la visibilidad, en todo el workspace, con alcance de carpeta o específica del proyecto, y configurar applyTo para que los archivos correctos activen la skill. Cualquier skill que completa esos tres pasos es utilizable.

La fase opcional agrega poder. Puedes anexar scripts de automatización en Python, Bash o Node, con validación de argumentos y manejo de errores adecuados, y puedes enriquecer la skill con documentos de referencia, ejemplos reales y enlaces a skills relacionadas. La fase de calidad es donde la skill se gana la confianza de producción. Validas y pruebas de extremo a extremo con prompts reales, confirmas que la salida coincide con las plantillas y ejecutas cada elemento de la lista de verificación. Luego la distribuyes a su alcance y confirmas que otros agentes pueden descubrir e invocar la skill, y que la activación por glob y applyTo se dispara como se espera. De la idea a la distribución no es una línea recta. Es lo esencial primero, después el poder, después la prueba.

El modelo de madurez de cinco niveles: una skill es código

Las skills maduran en cinco niveles, y saber dónde está una skill dice qué le falta todavía. El nivel uno es un SKILL.md de cerca de cuarenta líneas que cubre descripción y flujo de trabajo. Funciona, y es frágil. El nivel dos agrega referencias y completa las plantillas de salida. El nivel tres es el objetivo para la mayoría de las skills enterprise: las cinco secciones, scripts de automatización, referencias a

portales, contenido enriquecido y patrones glob ajustados. El nivel cuatro agrega pruebas automatizadas y un gate de CI, para que un cambio que rompe la skill falle en el build. El nivel cinco es una entrada con versión fija en un registro cross-workspace con promoción automatizada vía CI. En ese punto la skill es un activo de software de primera clase, no una anotación en un editor.

La mayoría de los equipos entrega skills de nivel uno que funcionan, y el costo aparece después. Las skills de nivel uno se proliferan sin promoción, y terminas con salidas inconsistentes, conocimiento de dominio duplicado y ningún quality gate en ninguna parte. El modo de falla tiene nombre: skills editadas in place, nunca versionadas, donde la calidad de la salida cambia en silencio, sin rollback y sin changelog. La corrección es tratar todo SKILL.md como código. Versiónalo, revísalo en un pull request, pruébalo y promuévelo por los niveles. Una skill no es documentación que por casualidad vive en un repositorio. Es un activo que entrega comportamiento, y el comportamiento que se entrega sin control de versiones es comportamiento en el que no puedes confiar.

> DEFINICIÓN
>
> Trata todo SKILL.md como código, no como una anotación. Cinco secciones que pasan la prueba SFA, un modo de activación elegido por su costo en tokens y un nivel de madurez que puedes nombrar. Versiónalo, revísalo, pruébalo, promuévelo. Una skill que entrega comportamiento sin control de versiones es comportamiento en el que no puedes confiar.

Referencias

Fuentes primarias para la anatomía de la skill, la activación, el ciclo de vida y el modelo de madurez.

- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, el benchmark detrás de la brecha de contexto de skill: cerca del 11 por ciento sin contexto de dominio, cerca del 38 por ciento con él.
- Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, el caso empírico para validar y versionar skills antes de la distribución.

- Customize AI in Visual Studio Code, referencia para la activación por glob, applyTo y archivos de instrucciones en GitHub Copilot.
- GitHub Copilot Documentation, la referencia de plataforma para agent mode, skills y 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 ENRUTANDO EL TRABAJO

Dos tipos de agente y el árbol de decisión: envía cada tarea al generalista o a un especialista.

GitHub Copilot te da dos tipos de agente: un orquestador generalista, que resuelve la mayor parte del trabajo, y custom agents, cada uno dueño de un dominio bien delimitado. Elegir entre ellos en cada tarea es una decisión de enrutamiento, y enrutar por instinto es donde el costo y la calidad se desvían sin que nadie lo note. Este capítulo define los dos tipos, el contrato de seis secciones que hace seguro a un custom agent y el árbol de cuatro preguntas que resuelve cada decisión de enrutamiento en menos de un minuto.

Dos tipos de agente, y la diferencia que decide el enrutamiento

El runtime de GitHub Copilot opera con dos tipos de agente, y toda decisión de enrutamiento empieza por distinguir uno del otro. Un agente generalista es el orquestador por defecto. Resuelve del 70% al 80% del trabajo en un workspace maduro, y se vuelve más preciso cuando cargas un SKILL.md que aporta el conocimiento de dominio que le falta. Usa herramientas compartidas a nivel de workspace y no lleva un archivo de identidad propio. Es la elección correcta para la mayoría de las tareas de desarrollo: cualquier cosa que se beneficie de conocimiento de dominio inyectado sin necesitar herramientas exclusivas ni una identidad operativa distinta.

Un custom agent es un componente de software, no un prompt. Se define por un archivo .agent.md que fija su identidad, sus conexiones exclusivas de herramientas, su alcance delimitado, sus contratos de entrada y salida y sus reglas de escalamiento. Los custom agents se conectan a MCP servers, CLIs y APIs que no se comparten con otros agentes. Atienden el 20% al 30% de las tareas que necesitan acceso exclusivo a herramientas, un tono distinto (una revisión de seguridad

rigurosa se lee distinto de una documentación amigable) o un lugar en un pipeline de varias etapas con handoffs tipados.

La brecha de capacidad es real, y por eso importa la división. En SWE-bench, un modelo de frontera resuelve cerca del 74% de las correcciones de bug con alcance bien acotado. En FeatureBench, que mide trabajo real de feature, el mismo modelo sin contexto de dominio cae a cerca del 11%, y el contexto de skill lo eleva de vuelta hacia el 38% (FeatureBench, arXiv 2026). Las skills cierran la mayor parte de esa brecha. Los custom agents con herramientas dedicadas cierran el resto. La proporción que señala un workspace saludable es de cerca de un custom agent por cada ocho a doce skills.

> DEFINICIÓN
>
> La proporción saludable es de aproximadamente un custom agent por cada ocho a doce skills. Por encima de eso, estás sobreingenierizando especialistas donde bastaba una skill. Por debajo, estás subescribiendo skills y el generalista termina adivinando conocimiento que nadie escribió. Generalista para el 70% al 80% del trabajo, apoyado por skills; custom agents para el 20% al 30% que exige herramientas exclusivas, una identidad distinta o handoffs de pipeline.

El contrato de seis secciones del .agent.md

Un custom agent se define enteramente por su archivo .agent.md, y ese archivo es un contrato de seis secciones con el orquestador. La sección uno es identidad y persona: nombre, rol, tono, dominio. La sección dos es el manifiesto de herramientas: los MCP servers, CLIs y APIs exclusivos que este agente puede llamar, y nada fuera de eso se ejecuta, aunque el cuerpo lo mencione. La sección tres es alcance y fronteras: qué atiende el agente y qué rechaza de plano. La sección cuatro es el contrato de entrada y salida: el payload de handoff que acepta y el artefacto tipado que devuelve. La sección cinco lista las skills que carga para conocimiento de dominio. La sección seis son las reglas de escalamiento: las condiciones que devuelven la tarea al orquestador sin completarla.

Cuatro agentes de producción

Cuatro agentes de workspaces reales muestran la forma. Un security agent tiene acceso exclusivo a un Vault MCP server, al Semgrep CLI y a la API de Snyk; devuelve un reporte de seguridad con puntajes CVSS; escala ante un CVE crítico o una exposición de credenciales. Un test agent tiene un GitHub Actions MCP server, Jest y una herramienta de cobertura; devuelve una suite de pruebas más los resultados de ejecución; escala ante pruebas inestables. Un modernization agent tiene un Terraform MCP server y el Azure SDK; devuelve un plan de migración y un diff; escala ante cualquier riesgo de pérdida de datos. Un documentation agent tiene un Confluence MCP server y la guía de estilo de la marca; devuelve documentación estructurada y publicada; escala en el instante en que una tarea se convierte en revisión de código.

Dos propiedades hacen seguros a estos agentes. Cada uno es dueño de un único dominio delimitado, así que el primer incidente queda contenido a ese dominio y no al workspace entero. Y cada uno nombra su disparador de escalamiento, así que devuelve el control en lugar de improvisar más allá de su competencia. El manifiesto de herramientas es donde vive el menor privilegio. La regla de escalamiento es donde vive la honestidad. En un estudio de caso instrumentado de un sistema C# de 108.000 líneas construido a lo largo de 283 sesiones, los agentes especialistas promediaban cerca de 700 líneas en los roles de mayor capacidad y cerca de 330 en los estándar, y más de la mitad de cada spec era conocimiento de dominio del proyecto, no instrucción de comportamiento (Vasilopoulos, arXiv 2026).

> DEFINICIÓN
>
> Seis secciones, en orden: identidad y persona, manifiesto de herramientas, alcance y fronteras, contrato de entrada y salida, skills cargadas, reglas de escalamiento. El manifiesto de herramientas impone el menor privilegio. La regla de escalamiento evita que un agente improvise más allá de su alcance. Un agente sin lista explícita de herramientas y sin disparador explícito de escalamiento no es un contrato, es un pasivo.

El árbol de decisión de cuatro preguntas, resuelto en menos de un minuto

El enrutamiento no necesita intuición. Necesita cuatro preguntas, hechas en orden, una vez por tarea. Primera: la tarea exige acceso exclusivo a herramientas, un MCP server, una API o un CLI dedicado? Si es sí, enruta a un custom agent más una skill. Segunda: la tarea exige una identidad o un tono distintos, como un revisor de seguridad riguroso difiere de un redactor de documentación amigable? Si es sí, custom agent más skill. Tercera: es un trabajo de varias etapas con handoffs entre agentes? Si es sí, un pipeline de custom agents. Cuarta, solo si las tres anteriores fueron no: ya existe un SKILL.md que cubre este dominio? Si es sí, generalista más skill. Si no, generalista solo, o escribe la skill primero.

El árbol llega a tres resultados. Custom agent más skill, para el trabajo que necesita su propio .agent.md con identidad única, herramientas exclusivas y un alcance definido: un security agent con un Vault MCP server, un test agent con herramientas de CI, un modernization agent con Terraform. Generalista más skill, cuando una skill ya carga el conocimiento y no se requiere una identidad especializada: escribir un DOCX, generar un diagrama, analizar un CSV. Generalista solo, para el trabajo que no exige ningún conocimiento especializado: una pregunta factual, un cálculo rápido, una lluvia de ideas. La evaluación toma bastante menos de un minuto por tarea. El punto es la consistencia: el mismo tipo de tarea se enruta igual cada vez, para cada persona del equipo.

> DEFINICIÓN
>
> Cuatro preguntas, tres resultados, menos de un minuto por tarea. Herramientas exclusivas, una identidad distinta o handoffs de pipeline envían la tarea a un custom agent más una skill. Un dominio cubierto la envía al generalista más una skill. Nada especial la envía al generalista solo. Enrutar por un árbol, y no por instinto, es lo que hace repetibles el costo y la calidad.

Custom agents en GitHub Copilot: .github/agents, agent mode y el picker

En GitHub Copilot el contrato vive en el repositorio. Los custom agents son archivos Markdown con frontmatter YAML en .github/agents/, cada uno nombrado con la extensión .agent.md. El frontmatter declara el nombre del agente, su descripción, su array de herramientas permitidas y el modelo en el que corre. El cuerpo es el system prompt: identidad, la lista ordenada de lo que el agente busca y el formato de salida. Un agente está activo a la vez, y cuando lo invocas en agent mode reemplaza la persona de chat por defecto y trabaja solo con las herramientas que le permitas. Las reglas de todo el repositorio en .github/copilot-instructions.md siguen aplicando por debajo, así que un agente hereda la base del workspace y le suma su especialización encima.

Elegir el modelo, y pagarlo

El array de herramientas es donde se impone el menor privilegio, y el campo de modelo es donde eliges el motor. El picker de Copilot expone varios modelos, y la mentalidad de enrutamiento continúa dentro del agente. Ve por Claude Opus 4.8 o Claude Fable 5 en el razonamiento más difícil: el refactor de varios archivos, la revisión de seguridad que tiene que estar bien. Sonnet 5 es el default equilibrado para la mayor parte del trabajo diario de agente. Haiku 4.5 es la opción rápida y de bajo costo para tareas de alto volumen y alcance acotado, donde la latencia importa más que la profundidad. Gemini 3.1 Pro y los modelos GPT-5.x están ahí cuando la tarea encaja con sus fortalezas. Combina el modelo con la tarea igual que combinaste el agente con la tarea: capacidad donde compensa, velocidad y costo donde no.

La facturación sigue esa elección. GitHub Copilot mide el trabajo de agente en GitHub AI Credits, donde un crédito equivale a un centavo de dólar, en vigor desde el 1 de junio de 2026. Un modelo más pesado en una tarea más pesada gasta más créditos, así que el árbol de enrutamiento y el picker de modelo son la misma palanca de control de costo vista desde dos ángulos. Una hora dedicada a armar el árbol de cuatro preguntas puede recortar el gasto promedio de tokens en cerca del 30% y estabilizar la calidad de la salida en todo el equipo, que es el trabajo de infraestructura más barato que la mayoría de los equipos tiene a la mano.

DEFINICIÓN

Los custom agents viven en .github/agents/*.agent.md: frontmatter para nombre, herramientas y modelo; cuerpo para el system prompt. El agent mode cambia a esa persona mientras .github/copilot-instructions.md permanece por debajo. Elige el modelo por tarea en el picker de Copilot: Opus 4.8 o Fable 5 para el razonamiento más difícil, Sonnet 5 como default equilibrado, Haiku 4.5 para trabajo rápido de alto volumen, Gemini 3.1 Pro y GPT-5.x donde encajen. Copilot lo mide todo en GitHub AI Credits, a un crédito por centavo de dólar.

Referencias

Fuentes primarias para los dos tipos de agente, el árbol de enrutamiento, el contrato .agent.md y la selección de modelo en GitHub Copilot.

- GitHub Copilot documentation, la plataforma de referencia para agent mode, custom agents y la superficie de configuración que este capítulo enruta.
- GitHub Docs, Choosing the right AI model for your task, la guía oficial detrás de emparejar un modelo del picker de Copilot con el trabajo que tienes enfrente.
- Customize AI in Visual Studio Code, cómo se crean e invocan agentes, skills y prompt files en el editor.
- FeatureBench, Benchmarking Agentic Coding for Complex Feature Development, la brecha de capacidad entre correcciones de bug bien acotadas y trabajo real de feature que motiva enrutar hacia especialistas.
- Vasilopoulos, Codified Context Infrastructure, el estudio de caso instrumentado detrás del conteo de líneas de los agentes especialistas y la porción de conocimiento de dominio 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 EJECUCIÓN Y GOBERNANZA

Orquestración, handoffs y rieles.

Un workspace de GitHub Copilot a escala no es un agente respondiendo una pregunta. Es un orquestador que recibe cada solicitud, la enruta y coordina una cadena de especialistas detrás de una única conversación. Este capítulo describe el runtime que lo hace posible: cuatro fases de la entrada a la entrega, handoffs que acumulan contexto en lugar de perderlo, hooks que aplican política de forma determinística, y una jerarquía de configuración que decide quién puede definir qué regla. Los agentes hacen el trabajo. Los rieles deciden si ese trabajo es seguro para poner en producción.

El runtime de cuatro fases, y el único agente con quien habla el usuario

Cada solicitud en un workspace de Copilot pasa por cuatro fases. Entrada: el orquestador recibe el prompt, carga su propia identidad en .agent.md y clasifica el tipo de tarea. Análisis y enrutamiento: busca un SKILL.md correspondiente y evalúa el árbol de decisión, eligiendo entre un agente generalista con una skill y un custom agent con herramientas dedicadas. Ejecución: las skills cargan, el .github/copilot-instructions.md y cualquier .instructions.md con alcance se aplican automáticamente, las herramientas MCP conectan, y la tarea corre con contexto completo. Validación y entrega: la salida se verifica contra el checklist de la propia skill, un quality gate revisa formato, contenido y convenciones, y solo entonces la respuesta llega al usuario.

La decisión de enrutamiento ocurre una vez, en la segunda fase, no en cada turno. Lo que importa más que las fases es la forma que imponen: el orquestador es el único agente con quien habla el usuario. Todo sub-agente corre detrás de él. Todo handoff regresa al orquestador, que decide el próximo paso. Los agentes nunca hablan directamente entre sí. Esa única regla es lo que mantiene depurable a un sistema multi-agente. Cuando un resultado sale mal, hay un único lugar donde mirar, un único contexto que inspeccionar y un único responsable de armar la respuesta

final. En el model picker de Copilot, el orquestador también puede enrutar por costo, enviando escaneos baratos al Haiku 4.5, generación al Sonnet 5 y análisis difícil al Opus 4.8.

Handoffs que acumulan contexto, sin perderlo

El patrón multi-agente más claro es un pipeline de cuatro etapas: Plan, Code, Test, Security. El agente de planificación recibe la solicitud del usuario y devuelve un plan estructurado con criterios de aceptación. El agente de código recibe la solicitud más el plan y devuelve la implementación con la lista de archivos que tocó. El agente de pruebas recibe la solicitud, el plan y el código, y devuelve pruebas con sus resultados de ejecución. El agente de seguridad recibe el cuadro completo, solicitud, plan, código y resultados de pruebas, y devuelve un informe de hallazgos. Cada etapa ve todo lo que produjeron las etapas anteriores.

Esa es la propiedad que hace funcionar al pipeline. El contexto se acumula en cada handoff. Un handoff pasa no solo la nueva entrada, sino todos los artefactos previos, de modo que ninguna etapa parte de cero y ninguna decisión anterior se descarta en silencio. El agente de seguridad solo puede hacer su trabajo si ve el plan que el código debía implementar y las pruebas que debían cubrirlo. Quita ese historial y la revisión de seguridad se vuelve adivinanza.

Artefactos tipados, orden estricto

Dos restricciones mantienen honesta la cadena. Cada agente emite un artefacto tipado, un plan, un diff, una suite de pruebas, un informe, que se vuelve la entrada de la próxima etapa, de modo que un handoff es un contrato, no una conversación. Y la secuencia es estrictamente ordenada: cada agente termina antes de que empiece el siguiente. El orquestador guarda el contexto acumulado y entrega la porción correcta al especialista correcto. Los hooks envuelven cada handoff y registran lo que cruzó la frontera, de modo que toda la cadena se puede auditar después.

Hooks como gates deterministicos

Skills y agents moldean lo que el modelo tiende a hacer. Los hooks deciden lo que de hecho ocurre, sin importar cómo se comporte el modelo. Viven en .github/hooks/*.json y disparan en seis eventos de ciclo de vida de GitHub Copilot: sessionStart cuando una sesión comienza o se reanuda, userPromptSubmitted cuando el usuario envía un prompt, preToolUse antes de que el agente use cualquier herramienta, postToolUse después de que una herramienta termina, sessionEnd cuando la sesión cierra, y errorOccurred cuando algo falla. Solo el preToolUse puede bloquear: puede denegar una llamada de herramienta antes de que corra. Los otros cinco observan, registran y reaccionan. Los hooks de varios archivos y plugins se combinan en lugar de sobrescribir, y para el coding agent en la nube los archivos de hook tienen que estar en el branch predeterminado del repositorio para cargar.

Cuatro hooks pertenecen a todo rollout el primer día, porque el modelo no los va a agregar por ti. Un estudio empírico de 2.303 archivos de contexto de agente en 1.925 repositorios encontró orientación de seguridad explícita en apenas 14,5% de ellos (Chatlatanagulchai et al. 2025). Los hooks son el respaldo determinístico para el 85,5% que lo deja fuera. Un secrets block en preToolUse deniega cualquier escritura o commit que cargue una credencial. Un formatter en postToolUse corre el linter en cada archivo que el agente edita, de modo que el estilo nunca se vuelva una discusión de review. Un audit log, disparado por sessionStart y postToolUse, registra cada acción, versión de modelo y justificación, de modo que un cambio pueda explicarse y revertirse. Un policy gate en preToolUse deniega comandos destructivos, el rm, el DROP TABLE, el kubectl delete contra producción, y llama a un humano en su lugar.

> DEFINICIÓN
>
> Los cuatro hooks del primer día: un secrets block y un policy gate en preToolUse que deniegan antes del daño, un formatter en postToolUse que cierra los debates de estilo, y un audit log en sessionStart y postToolUse que vuelve cada acción de agente explicable y reversible. Rieles determinísticos, no buena voluntad del modelo.

Gobernanza de configuración: pisos, contratos y un deny absoluto

El último riel es quién puede definir qué regla, y el modelo es unidireccional por diseño. La organización define los pisos mediante las políticas de GitHub Enterprise Copilot: exclusión de contenido, telemetría, acceso a modelos y aplicación de hooks que un único repositorio no puede aflojar. El workspace define el contrato del equipo en .github: copilot-instructions.md, .instructions.md con alcance, hooks, y el .vscode/settings.json que todo clon hereda. El usuario personaliza encima, preferencias de editor y configuraciones locales que nunca se filtran a un commit. Cada capa refina la de arriba. Ninguna puede contradecirla.

Una regla está por encima de todas las demás: el deny es absoluto. Una capa más baja puede agregar permiso donde se le dio espacio, pero nunca puede sobrescribir un deny definido más arriba. Una skill no puede saltarse una política. Un prompt no puede reactivar un comando que la organización bloqueó. Una configuración de usuario no puede debilitar un guardrail de workspace que el equipo comiteó. Eso es lo que vuelve a la jerarquía un modelo de gobernanza, y no una pila de archivos de config. También mantiene a los agentes rindiendo cuentas cuando actúan sobre dinero: las ejecuciones de agente se miden en GitHub AI Credits, donde un crédito equivale a un centavo de dólar desde junio de 2026, así que un techo por módulo detiene un loop descontrolado antes de que lo haga la factura, y el MCP expone los sistemas corporativos por una sola interfaz auditada, en lugar de credenciales dispersas.

Referencias

Fuentes primarias para el runtime de orquestación, el ciclo de vida de hooks de Copilot, la evidencia empírica de la brecha de seguridad y el modelo de gobernanza de configuración de este capítulo.

- GitHub Copilot, Hooks configuration reference, los seis eventos de ciclo de vida y el schema JSON que sigue todo archivo de hook.
- GitHub Copilot, Using hooks with the coding agent, por qué los archivos de hook tienen que estar en el branch predeterminado para cargar en el agente de la nube.

- GitHub Copilot in VS Code, Agent hooks, el modelo local de hooks para gates de preToolUse y postToolUse.
- GitHub Copilot, Repository custom instructions, el copilot-instructions.md y los archivos de instrucción con alcance que forman el contrato del equipo.
- Chatlatanagulchai et al., An Empirical Study of Agent Context Files, el hallazgo de que solo 14,5% de 2.303 archivos en 1.925 repositorios llevan orientación de seguridad explícita.
- GitHub Copilot documentation, la plataforma de referencia para el agent mode, el model picker y el MCP.
- Model Context Protocol Specification, el estándar abierto para exponer sistemas corporativos a los agentes por una sola interfaz 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 ADOPCIÓN Y CIERRE

Anti-patrones, diagnósticos y 90 días: de la teoría de enrutamiento a una práctica repetible.

Estar de acuerdo con la teoría de enrutamiento es fácil. Sostenerla es difícil. Este capítulo es la capa de adopción: los cuatro anti-patrones que aparecen antes de que se consolide cualquier buena práctica, seis preguntas diagnósticas que revelan brechas en cualquier workspace, y un plan de 90 días que convierte el árbol de decisión de cuatro preguntas en un hábito. Cierra con el único número que paga por todo esto.

Cuatro anti-patrones que aparecen antes de cualquier buena práctica

Cada workspace que vi escalar GitHub Copilot pasó primero por los mismos cuatro anti-patrones. El primero es la skill como megaprompt: un SKILL.md crece hasta dos mil líneas, cubre el dominio entero y se activa en cada glob match. El overhead de tokens por llamada pasa a superar el valor del conocimiento que carga, ahora una línea legible en GitHub AI Credits. La solución es dividirlo en tres skills enfocadas, cada una lo bastante estrecha para pasar la prueba SFA de Scope, Format y Audience. El segundo es el agente para todo: un único custom agent que tiene todos los permisos y no rechaza ninguna tarea. Funciona hasta el primer incidente, cuando el radio de impacto es el workspace entero. Un custom agent necesita un alcance delimitado en su .agent.md y un manifiesto de herramientas que no excede, porque en agent mode todo lo que ese agente alcanza vía MCP es lo que un atacante hereda si es comprometido.

El tercero son las skills nunca versionadas: archivos SKILL.md editados in place, calidad de las salidas deteriorándose, sin changelog y sin rollback cuando un cambio empeora los resultados. Trata cada SKILL.md como código: versiónalo, revísalo en un pull request, pruébalo y promuévelo. El cuarto es el enrutamiento por intuición,

donde cada desarrollador decide por llamada, por instinto, si una tarea necesita un custom agent, una skill o ninguno. El resultado es salida inconsistente, gasto de AI Credits inconsistente y ninguna línea base para medir. El árbol de decisión de cuatro preguntas elimina la adivinanza, y tarda alrededor de una hora en acordarse en equipo. Estas fallas son baratas de prevenir y caras de deshacer, y por eso pertenecen al inicio de cualquier plan de adopción.

> DEFINICIÓN
>
> Los cuatro anti-patrones, en el orden en que surgen: skill como megaprompt, agente para todo, skills nunca versionadas y enrutamiento por intuición. El primero infla los AI Credits, el segundo el radio de impacto, el tercero borra el rastro de auditoría, y el cuarto elimina la línea base.

Seis preguntas diagnósticas para cualquier workspace

Antes de planear cualquier cambio, mide dónde estás. Seis preguntas revelan las brechas en cualquier workspace de Copilot, y un no a cualquiera de ellas vale la pena abordarlo antes del próximo sprint. ¿Conoces todas las skills activas en tu workspace ahora mismo? ¿Cada custom agent tiene un .agent.md con lista explícita de herramientas? ¿Puedes rastrear qué skill se cargó para cualquier salida de tarea? Las tres primeras preguntas son sobre visibilidad. Si no puedes responderlas, estás operando sin instrumentación, y ninguna optimización sobrevive al contacto con un sistema que no ves.

Las tres últimas son sobre disciplina. ¿Existe un árbol de decisión de enrutamiento que todo el equipo sigue, o cada miembro decide de forma ad hoc? ¿Todas las skills están bajo control de versiones y se revisan antes de distribuirse? Y el único número que vale checar directamente: ¿la proporción de custom agents a skills está en 1:8 o mejor? Más custom agents que eso suele significar un workspace sobre-ingenierizado, pagando por identidades a medida donde una skill compartida en el agente generalista haría el mismo trabajo por una fracción de los AI Credits. Muchas menos skills que agentes significa subespecialización, forzando a los custom agents

a cargar conocimiento que pertenece a un SKILL.md reutilizable. La proporción de 1:8 es la lectura más rápida de cualquiera de los dos desvíos.

El playbook de 90 días: Fundación, Contrato, Medición

Los equipos que escalan el valor de Copilot siguen la misma progresión de tres fases, y cada fase termina con un resultado concreto. La Fundación va del día 1 al día 30: haz un inventario completo de skills y puntúa cada SKILL.md contra la prueba SFA, adopta el árbol de decisión de cuatro preguntas como equipo, revisa el .agent.md de cada custom agent y habilita el log de activación de skills. Consolida las convenciones del repositorio en .github/copilot-instructions.md, para que el agente generalista empiece cada tarea con las mismas reglas de base. El único resultado de la Fundación es una línea base de enrutamiento: un registro escrito de lo que existe y de cómo se asigna el trabajo.

El Contrato va del día 31 al día 60: promueve tus cinco principales skills al nivel de madurez 3, el nivel que carga las cinco secciones, scripts y patrones glob ajustados, luego pon el primer pipeline en marcha en agent mode, con plan, code y test haciendo handoff por contratos tipados, y empieza a medir los AI Credits por tarea. El único resultado del Contrato es un primer pipeline entregando artefactos tipados, en el que cada handoff acumula contexto en lugar de perderlo. La Medición va del día 61 al día 90: publica una biblioteca compartida de skills, aplica el árbol de decisión a al menos 80% de las tareas, da seguimiento a la eficiencia de tokens semanalmente en AI Credits y documenta un proceso replicable para agregar skills. El único resultado de la Medición es una biblioteca de skills operativa y calidad medible. El objetivo de los 90 días no es tener skills perfectas. Es tener un proceso repetible para hacerlas mejores.

Lo que devuelve una hora de diseño de enrutamiento

Este es el número que paga el ejercicio entero. Escribir el árbol de decisión de cuatro preguntas toma alrededor de una hora. En los workspaces que lo adoptan, el gasto promedio de tokens por tarea baja cerca de 30%, porque las tareas que una skill en

el agente generalista puede responder dejan de enrutarse a un custom agent que jala tres veces más contexto. En GitHub Copilot ese ahorro está denominado en GitHub AI Credits, donde un crédito equivale a un centavo de dólar desde el 1 de junio de 2026. Una reducción de 30% en el gasto de tokens es una reducción de 30% en la línea de AI Credits, legible en el panel de uso.

El segundo retorno es la consistencia, más difícil de reducir a un número y más fácil de sentir. Cuando el enrutamiento es deliberado, el mismo tipo de tarea sigue el mismo camino cada vez, carga la misma skill y pasa por la misma lista de verificación antes de la entrega. La salida deja de depender de qué desarrollador formuló el prompt, y la carga de revisión baja porque los revisores encuentran menos sorpresas. Una hora de diseño compra las dos cosas: un recorte medible en AI Credits y una salida más predecible en todo el equipo. Este es el cierre de este playbook. El enrutamiento es la primera decisión de ingeniería que tomas con GitHub Copilot, y la que sigue pagando.

> DEFINICIÓN
>
> Una hora de diseño de enrutamiento devuelve cerca de 30% menos gasto de tokens, medido en GitHub AI Credits, y una salida más consistente en todo el equipo. El enrutamiento es la primera decisión de ingeniería que tomas con GitHub Copilot, y la más barata de acertar.

Referencias

Fuentes primarias para el modelo de enrutamiento, la brecha de capacidad en trabajo de features real, la superficie de seguridad de las skills de agente y los primitivos de personalización de GitHub Copilot sobre los que se apoya este capítulo.

- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, la brecha de capacidad en trabajo de features real que las skills de dominio y los custom agents delimitados existen para cerrar.
- Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, la base empírica de por qué el anti-patrón del agente para todo es peligroso y de

por qué todo SKILL.md necesita control de versiones y revisión.

- Customize AI in Visual Studio Code, la referencia para los primitivos SKILL.md, .agent.md y .instructions.md sobre los que este playbook enruta.
- GitHub Copilot documentation, la plataforma de referencia para agent mode, conexiones MCP y el modelo de facturación en 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