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

The right primitive for the task decides whether a workspace scales or collapses.

Most teams adopt GitHub Copilot as a single chat box, and it breaks at the second use case. This playbook treats routing as the first engineering decision: six primitives, agents, skills, plugins, MCP servers, hooks, and the configuration hierarchy, each with a boundary that decides whether the workspace scales. The payoff is measurable, roughly 30 percent less token spend per task, now billed in GitHub AI Credits, and output that stays consistent across the team.

AUTHOR

Paula Silva

ROLE

AI-Native Software Engineer

CONTACT

in/paulanunes

READING TIME

~ 45 minutes, end to end

6
CORE PRIMITIVES

6
CHAPTERS

4
ROUTING QUESTIONS

~30%
TOKEN SPEND CUT

CHAPTERS

Chapter 00

Routing is the first engineering decision

Why the single chat box breaks at the second use case, the SWE-bench to FeatureBench capability gap in numbers, what a misrouted task costs in GitHub AI Credits, and the six primitives whose boundaries decide scale.

The Problem →

Chapter 01

Four control points, one task flow

How a Copilot reply is assembled from a stack of files: the workspace stack read bottom up, the four core file types, the six prompt layers, and the .github/ configuration hierarchy as a Rosetta Stone.

The Map →

Chapter 02

SKILL.md: anatomy, activation, and maturity

The five sections and the SFA test, the three activation modes and their token cost, the lifecycle from idea to distribution, and the five-level maturity model that treats a skill as code.

Skills →

Chapter 03

Two agent types and the decision tree

Generalist versus custom agents, the healthy one-agent-per-eight-to-twelve-skills ratio, the .agent.md six-section contract, the four-question routing tree, and choosing a model in the Copilot picker.

Agents →

Chapter 04

Orchestration, handoffs, and the rails

The four-phase runtime with one orchestrator, the Plan, Code, Test, Security handoff chain that accumulates context, the six hook lifecycle events, and the governance model where deny is absolute.

Execution →

Chapter 05

Anti-patterns, diagnostics, and 90 days

The four anti-patterns that appear first, six diagnostic questions, the 90-day adoption playbook, and what one hour of routing design returns in GitHub AI Credits.

Adoption →

READING TIPS

> **KEYBOARD SHORTCUTS**
>
> Press `/` to search. Use `j` and `k` to move between chapters. Press `t` to toggle theme. Press `g` to go to top.

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 THE PROBLEM

Routing is the first engineering decision.

Most teams adopt GitHub Copilot as a single chat box. Every task goes to the same generalist agent, every answer is rebuilt from the same context window, and the setup works until the second use case arrives. This chapter argues that routing, deciding which task goes to which primitive, is the first engineering decision, not a later optimization. It puts numbers on the capability gap, prices the cost of a misrouted task in GitHub AI Credits, and names the six primitives whose boundaries decide whether a workspace scales.

The single chat box trap: one agent, one context window, one bottleneck

The most common mistake I see teams make with GitHub Copilot is treating it as one generalist chat interface. Every task goes to the same agent in agent mode. Every response rebuilds context from the same window. In a mature workspace, a generalist orchestrator handles roughly 70% to 80% of tasks well, and that is fine. The trap is sending it the other 20% to 30% too, the work that needs a specific skill, an exclusive tool, or a distinct identity. One agent absorbs everything, and the context window becomes the bottleneck.

This bottleneck grows with the team, not with the individual. At one developer, a slightly bloated context window is noise. Across dozens of developers running the same generalist on every task, it becomes a measurable cost driver. The context window has a fixed budget. When a single agent carries brand rules, security standards, domain vocabulary, and the current task all at once, useful signal gets pushed out. The output drifts, reviewers catch more mistakes, and the productivity the tool promised stalls. The failure is not the model. It is the routing.

The capability gap in numbers: SWE-bench looks solved, real features do not

The distance between a benchmark and a real feature is where the case for routing gets concrete. On SWE-bench, a frontier model resolves 74.4% of well-scoped bug fixes, the tasks with tight, isolated requirements. That number reads like the problem is solved. On FeatureBench, which represents real-world feature work rather than localized patches, the same model without domain context drops to 11% (arXiv:2602.10975). The benchmark measured localized patches, not feature development, so the two numbers describe two different jobs.

The recoverable part of that gap is the interesting part. Add skill context, meaning architecture patterns, coding conventions, and domain vocabulary delivered through a SKILL.md, and the FeatureBench figure climbs to roughly 38%. Skills close about 27 percentage points of the distance. The rest is not closed by a bigger model. It is closed by custom agents with dedicated tool access, the primitives this playbook builds toward. The lesson is that context, delivered by the right primitive, moves the number more than raw model capability does.

What a misrouted task costs, now priced in GitHub AI Credits

A misrouted task is not just lower quality. It is more expensive, and now the cost is itemized. Production telemetry from GitHub Copilot Enterprise in Q1 2026 shows that a generalist agent handling work a specialized skill could answer directly consumes about three times more context tokens per task. Since June 1, 2026, that consumption is metered in GitHub AI Credits, where one credit equals one cent. The waste no longer hides inside a flat subscription. It appears on the bill.

Multiply three times the tokens by every developer who runs the generalist on skill-shaped work, and the number stops being noise. The same telemetry points the other way too: teams that put one hour into routing design, adopting a simple decision tree instead of choosing by feel, cut average token spend by around 30%. That is the core economic argument of this playbook. Routing is not a tax on speed. It is the difference between GitHub AI Credits spent reasoning and GitHub AI Credits spent rebuilding context the workspace already had.

The six primitives, and why their boundaries decide scale

The modern GitHub Copilot workspace is not one assistant. It is a composition of six primitives, each with its own file format, lifecycle, and contract with the others. Agents pick the path. Skills pack the domain knowledge in a versioned SKILL.md. Plugins reach out to the world. MCP servers expose typed tools through the Model Context Protocol. Hooks keep the deterministic rails, firing at fixed lifecycle events. And the configuration hierarchy, rooted in files like .github/copilot-instructions.md, decides who wins when two of them disagree.

The boundaries between these six are what matter. A skill is knowledge, not a tool. An agent is an identity and a scope, not a place to dump knowledge. A hook is a deterministic gate, not a prompt. When those boundaries are explicit, a workspace composes cleanly and scales from one developer to the whole organization. When they blur, everything collapses back into the single chat box, and the workspace breaks at the second use case. Getting the boundaries right is the whole job. The rest of this playbook works through them one primitive at a time.

> DEFINITION
>
> Routing in one sentence: decide which of the six primitives owns a task before you send it, because a generalist doing skill-shaped work costs about 3x the context tokens, now billed in GitHub AI Credits, and the capability gap between a solved benchmark and a real feature is closed by the right primitive, not by a bigger model.

References

Primary sources for the capability gap, the routing decision, and the six primitives this chapter names.

- FeatureBench, Benchmarking Agentic Coding for Complex Feature Development, the empirical gap: a frontier model at 74.4% on SWE-bench solves 11% of real feature tasks without domain context.

- Agent Skills in the Wild, An Empirical Study of Security Vulnerabilities at Scale, context for why skill boundaries and governance matter as skills proliferate across a workspace.
- Customize AI in Visual Studio Code, the reference for skills, custom agents, instructions, and prompt files in the workspace.
- GitHub Copilot Documentation, the reference platform for agent mode, MCP, and the configuration hierarchy in this 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 THE MAP

Four control points, one task flow: how the workspace stack assembles a reply.

When you send a task to GitHub Copilot in agent mode, the reply is not produced by the model alone. It is produced by a stack of files that assemble around your request. Read the stack bottom up and it tells one story: the skill loads the domain knowledge, the tools attach the external capabilities, the hooks gate what is allowed to run, and the agent answers with all of that in context. This chapter maps the four control points, the six prompt layers they feed, and where every primitive lives in the repository.

The workspace stack, read bottom up

I read a Copilot workspace the way I read a factory line. Start at the bottom and follow the material up. The skill loads first: a SKILL.md brings the domain knowledge the task needs, the conventions and the workflow. The tools attach next: MCP servers, CLIs, and APIs give the agent capabilities it does not have by default. The hooks gate the run: a preToolUse hook can deny a dangerous command before it executes. The agent answers last, with the domain knowledge, the tools, and the guardrails already in place. Four layers, four jobs, one reply.

The value of each layer becomes visible when it is skipped. Skip the skill and the agent guesses the conventions the codebase already encodes; production telemetry from GitHub Copilot Enterprise in the first quarter of 2026 showed that a generalist handling a task a specialized skill could answer directly consumed 3 times more context tokens for the same work. That overhead is not abstract. It arrives on the bill as GitHub AI Credits, where one credit is one cent. Skip the tools and the agent cannot reach the database or the API it needs, so it fabricates a plausible answer instead. Skip the hooks and nothing stands between a generated command and your

repository. Skip the agent identity and one generalist carries every task, with no bounded scope and no exclusive tools.

The skill layer is where the largest gap closes

The clearest measurement of the skill layer comes from two benchmarks. On SWE-bench, which scores tight, isolated bug fixes, a frontier model resolves about 74% of tasks. On FeatureBench, which represents real feature work, the same model without domain context drops to 11%. Add skill context, the architecture patterns, the coding conventions, the domain vocabulary, and the figure climbs to roughly 38%. The skill layer closes about 27 percentage points of that gap. The remaining distance is what custom agents with dedicated tools are for. This is why the stack is worth reading as a stack: each layer recovers capability the layer below cannot.

The four core file types

Four file types carry the whole ecosystem, and each answers a different question. The first is instructions, and it comes in two scopes. The workspace file, .github/copilot-instructions.md, holds the rules that apply to the entire repository: the stack, the naming conventions, the commands. Scoped instructions live in .github/instructions/ as .instructions.md files, each with an applyTo glob that limits it to a path such as src/components/**/*.tsx. Instructions answer the question of what the agent must do, and they apply automatically, without the user asking. The agent cannot ignore them; they function as guardrails.

The other three file types answer the remaining questions. Tools, declared in .mcp.json, answer what the agent can do: an MCP server connects it to GitHub, a database, or an internal service; a CLI gives it git or az; an API gives it an HTTP call. The .agent.md file in .github/agents/ answers who the agent is: its name and role, the tools it is allowed to use, the tasks it accepts and refuses, and the handoff rules for when it delegates. The SKILL.md file in .github/skills/ answers what the agent knows: the templates, the step-by-step workflow, and the validation checklist for one type of output. Rules constrain behavior, tools enable capability, identity bounds scope, knowledge raises quality. The four are independent and composable, which is the point.

The six prompt layers that assemble the final context

Those files do not reach the model as files. They are merged into a single context, assembled in six layers. The model base comes first: the system prompt set by GitHub and by whichever model you selected in the Copilot picker, whether that is Claude Opus 4.8, Sonnet 5, Haiku 4.5, Claude Fable 5, Gemini 3.1 Pro, or a GPT-5.x model. Custom instructions from copilot-instructions.md are injected next, then the scoped instructions that match the current file. Agent identity is loaded from the active .agent.md. Skill knowledge is added from the triggered SKILL.md. Your prompt is placed last, as the highest-priority input. Later layers refine earlier layers but do not contradict them: a skill cannot override a safety rule, and a prompt cannot bypass a denied command.

The reason for six separate layers, fed by separate files, is evolution. Each file type targets one layer so it can change on its own. You can version a SKILL.md and promote it through a maturity model without touching the agent that loads it. You can tighten an applyTo glob on one scoped instruction without editing the workspace file every developer reads. You can swap the model in the picker without rewriting a single instruction. Separation of concerns at the file level becomes independent evolution at runtime. That is the difference between a workspace you can maintain and a single megaprompt no one dares to edit.

> DEFINITION
>
> The six layers, in assembly order: model base, then custom instructions, then scoped instructions, then agent identity, then skill knowledge, then your prompt last as the highest-priority input. Each layer refines the one before it, and none can contradict it.

The configuration hierarchy as a Rosetta Stone

For GitHub Copilot, every primitive has a fixed home under .github/, and the location is the contract. Workspace instructions live at .github/copilot-instructions.md. Scoped instructions live in .github/instructions/ as .instructions.md files with applyTo

globs. Agents live in .github/agents/ as .agent.md files. Skills live in .github/skills/ as SKILL.md files inside a named folder. Hooks live in .github/hooks/ as JSON files that declare preToolUse, postToolUse, sessionStart, and sessionEnd handlers. Reusable prompt templates live in .github/prompts/ as .prompt.md files. MCP connections live in .mcp.json at the root. Plugins bundle several of these together through a plugin.json manifest, distributed through the Awesome GitHub Copilot marketplace.

I call this table a Rosetta Stone because the same concept exists in Claude Code and in GitHub Copilot under different names and paths, and once you can read one column you can read the other. copilot-instructions.md is the Copilot equivalent of CLAUDE.md. .github/instructions/ is the equivalent of .claude/rules/. .github/agents/ maps to .claude/agents/, and .github/skills/ maps to .claude/skills/ with an identical SKILL.md format. The one layer you do not author is the model base; you choose it in the picker and pay for it in AI Credits. Everything else is a file you write, put in a predictable place, and let the orchestrator discover. That predictability is what lets a workspace scale past a single chat box.

References

Primary sources for the four file types, the prompt layers, and the GitHub Copilot configuration hierarchy.

- GitHub Copilot, Adding repository custom instructions, the reference for copilot-instructions.md and scoped .instructions.md files with applyTo globs.
- Customize AI in Visual Studio Code, the overview of agents, prompt files, and skills as separate customization primitives.
- Prompt Files in VS Code, the format for reusable .prompt.md templates with agent mode, model, and tool frontmatter.
- GitHub Copilot Hooks Configuration Reference, the JSON schema and event model for preToolUse, postToolUse, sessionStart, and sessionEnd hooks.
- Model Context Protocol Specification, the open standard behind the .mcp.json tool connections.
- GitHub Copilot: Instructions vs Prompts vs Custom Agents vs Skills, a side-by-side comparison of every Copilot configuration primitive.

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 PACKING KNOWLEDGE

SKILL.md: anatomy, activation, and maturity.

A SKILL.md is not a prompt, not a config file, and not a script. It is a domain knowledge artifact with a fixed structure, three activation modes, and a maturity path that runs from a forty line draft to a version pinned registry entry. This chapter takes the artifact apart. It covers the five sections every skill carries, the SFA test that gates production use, how each activation mode decides token cost, the lifecycle from idea to distribution, and the five level model that lets a team treat a skill as code.

Five sections, and the SFA test that gates production

Every SKILL.md carries five sections, and skipping one is where most skills fail in review. The file opens with YAML frontmatter that declares name, description, globs, and alwaysApply. Section one states the description and purpose. Section two lists the domain rules: the non-negotiable constraints, naming conventions, and anti-patterns that hold the hard knowledge of the domain. Section three is a numbered step by step workflow, explicit enough that another engineer or an agent can reproduce it, with references to the tools and scripts it calls. Section four gives output templates, concrete examples of format, structure, and file naming, so the agent matches a template instead of inventing one. Section five is a validation checklist with pass or fail criteria applied before delivery. That last section is a quality gate built into the skill itself: if any item fails, the agent revises before it responds.

Before a skill reaches production it has to pass the SFA test. SFA stands for Scope, Format, and Audience, and all three must be explicit in the file. Scope says what the skill covers and, just as important, what it refuses. Format fixes the shape of the output. Audience names who the output is for. I treat the SFA test as the entry gate because a skill that cannot answer those three questions is a skill that will drift. The

payoff is measurable. On FeatureBench, which represents real feature work rather than isolated bug fixes, a frontier model in the Copilot picker such as Claude Opus 4.8 solves roughly 11 percent of tasks without domain context. Add skill context, meaning architecture patterns, coding conventions, and domain vocabulary, and that figure climbs to about 38 percent. The five sections are how that context gets encoded.

Three activation modes, and the token cost of each

A skill has three ways to enter context, and each trades convenience against cost. Glob match is automatic: a file enters the working set, its path matches the glob pattern the skill declares, and the skill loads with no human action. It is the efficient default for file type specialization. alwaysApply set to true loads the skill on every interaction regardless of context. It fits cross cutting concerns like brand rules, security standards, and shared coding conventions, the things that should apply everywhere. Explicit invoke is the third mode: the orchestrator or the developer references the skill by name, on demand. It is precise, and it suits composite tasks that pull in several skills at once, but it depends on the caller knowing the skill exists.

The choice matters because activation is where token cost is decided. Production telemetry from GitHub Copilot Enterprise in the first quarter of 2026 shows a generalist handling a task a specialized skill could answer directly consumes about three times the context tokens per task. At one developer that is noise. Across a team it is a real line item. alwaysApply is the mode to watch: it applies broadly and it bills broadly, so I reserve it for the few concerns that genuinely belong on every call. For everything else, applyTo and glob patterns scope a skill to the files and folders where it earns its tokens, and nowhere else. One hour spent designing that scoping, rather than loading everything everywhere, can cut average token spend by around 30 percent.

The lifecycle: essential, optional, and quality phases

Building a skill runs through three phases, and only the first is mandatory. The essential phase has three steps. Step one, identify and organize: define the purpose, decide whether the skill is reusable across projects or specific to one, set a naming convention, and confirm it passes the SFA test. Step two, create the structure: write the SKILL.md with its frontmatter and its five sections, and add examples for clarity. Step three, estimate distribution: decide the visibility, workspace wide, folder scoped, or project specific, and configure applyTo so the right files trigger the skill. Any skill that completes those three steps is usable.

The optional phase adds power. You can attach automation scripts in Python, Bash, or Node, with proper argument validation and error handling, and you can enrich the skill with reference documents, real examples, and links to related skills. The quality phase is where a skill earns production trust. You validate and test it end to end with real prompts, confirm the output matches the templates, and run every checklist item. Then you distribute it to its scope and confirm other agents can discover and invoke it, and that glob and applyTo activation fire as intended. Idea to distribution is not a straight line. It is the essential first, then power, then proof.

The five level maturity model: a skill is code

Skills mature along five levels, and knowing where a skill sits tells you what it still needs. Level one is a roughly forty line SKILL.md that covers description and workflow. It works, and it is fragile. Level two adds references and completes the output templates. Level three is the target for most enterprise skills: all five sections, automation scripts, portal references, enriched content, and tuned glob patterns. Level four adds automated tests and a CI gate, so a change that breaks the skill fails the build. Level five is a version pinned entry in a cross workspace registry with automated CI promotion. At that point the skill is a first class software asset, not a note in an editor.

Most teams ship level one skills that work, and the cost shows up later. Level one skills proliferate without promotion, and you get inconsistent outputs, duplicate domain knowledge, and no quality gate anywhere. The failure mode has a name:

skills edited in place, never versioned, where output quality shifts silently with no rollback and no changelog. The fix is to treat every SKILL.md as code. Version it, review it in a pull request, test it, and promote it through the levels. A skill is not documentation that happens to sit in a repository. It is an asset that ships behaviour, and behaviour that ships without version control is behaviour you cannot trust.

> DEFINITION
>
> Treat every SKILL.md as code, not as a note. Five sections that pass the SFA test, an activation mode chosen for its token cost, and a maturity level you can name. Version it, review it, test it, promote it. A skill that ships behaviour without version control is behaviour you cannot trust.

References

Primary sources for skill anatomy, activation, the lifecycle, and the maturity model.

- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, the benchmark behind the skill context gap: about 11 percent without domain context, about 38 percent with it.
- Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, the empirical case for validating and versioning skills before distribution.
- Customize AI in Visual Studio Code, reference for glob, applyTo, and instruction file activation in GitHub Copilot.
- GitHub Copilot Documentation, the platform reference for agent mode, skills, and 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 ROUTING WORK

Two agent types and the decision tree: send each task to the generalist or a specialist.

GitHub Copilot gives you two kinds of agent: a generalist orchestrator that handles most of the work, and custom agents that each own one bounded domain. Choosing between them per task is a routing decision, and routing by feel is where cost and quality quietly drift. This chapter defines the two types, the six-section contract that makes a custom agent safe, and the four-question tree that settles every routing call in under a minute.

Two agent types, and the difference that decides routing

The GitHub Copilot runtime works with two agent types, and every routing decision starts by telling them apart. A generalist agent is the default orchestrator. It handles 70% to 80% of the work in a mature workspace, and it gets sharper when you load a SKILL.md that carries the domain knowledge it lacks. It uses workspace-level shared tools and carries no dedicated identity file. It is the right choice for most developer tasks: anything that benefits from injected domain knowledge without needing exclusive tools or a distinct operating identity.

A custom agent is a software component, not a prompt. It is defined by a .agent.md file that pins its identity, its exclusive tool connections, its bounded scope, its input and output contracts, and its escalation rules. Custom agents connect to MCP servers, CLIs, and APIs that are not shared with other agents. They handle the 20% to 30% of tasks that need exclusive tool access, a distinct tone (rigorous security review reads differently from friendly documentation), or a place in a multi-stage pipeline with typed handoffs.

The capability gap is real, which is why the split matters. On SWE-bench a frontier model resolves roughly 74% of tightly scoped bug fixes. On FeatureBench, which measures real feature work, the same model without domain context drops to about 11%, and skill context lifts it back toward 38% (FeatureBench, arXiv 2026). Skills close most of that gap. Custom agents with dedicated tools close the rest. The ratio that signals a healthy workspace is about one custom agent for every eight to twelve skills.

> DEFINITION
>
> The healthy ratio is roughly one custom agent per eight to twelve skills. Above that, you are over-engineering specialists where a skill would have done. Below that, you are under-writing skills and the generalist is guessing at knowledge nobody wrote down. Generalist for 70% to 80% of the work augmented by skills, custom agents for the 20% to 30% that needs exclusive tools, a distinct identity, or pipeline handoffs.

The .agent.md six-section contract

A custom agent is defined entirely by its .agent.md file, and that file is a six-section contract with the orchestrator. Section one is identity and persona: name, role, tone, domain. Section two is the tool manifest: the exclusive MCP servers, CLIs, and APIs this agent may call, and nothing outside it runs even if the body mentions it. Section three is scope and boundaries: what the agent handles and what it refuses outright. Section four is the input and output contract: the handoff payload it accepts and the typed artifact it returns. Section five lists the skills it loads for domain knowledge. Section six is the escalation rules: the conditions that send the task back to the orchestrator without completion.

Four production agents

Four agents from real workspaces show the shape. A security agent holds exclusive access to a Vault MCP server, the Semgrep CLI, and the Snyk API; it returns a security report with CVSS scores; it escalates on a critical CVE or a credential exposure. A test agent holds a GitHub Actions MCP server, Jest, and a coverage

tool; it returns a test suite plus execution results; it escalates on flaky tests. A modernization agent holds a Terraform MCP server and the Azure SDK; it returns a migration plan and a diff; it escalates on any data-loss risk. A documentation agent holds a Confluence MCP server and the brand style guide; it returns published structured docs; it escalates the moment a task turns into code review.

Two properties make these agents safe. Each one owns a single bounded domain, so the first incident is contained to that domain rather than the whole workspace. And each one names its escalation trigger, so it hands control back instead of improvising past its competence. The tool manifest is where least privilege lives. The escalation rule is where honesty lives. In one instrumented case study of a 108,000-line C# system built across 283 sessions, the specialist agents averaged around 700 lines for the higher-capability roles and about 330 for the standard ones, and more than half of each spec was project-domain knowledge rather than behavioral instruction (Vasilopoulos, arXiv 2026).

> DEFINITION
>
> Six sections, in order: identity and persona, tool manifest, scope and boundaries, input and output contract, skills loaded, escalation rules. The tool manifest enforces least privilege. The escalation rule keeps an agent from improvising past its scope. An agent without an explicit tool list and an explicit escalation trigger is not a contract, it is a liability.

The four-question decision tree, resolved in under a minute

Routing does not need intuition. It needs four questions, asked in order, once per task. First: does the task require exclusive tool access, an MCP server, an API, or a dedicated CLI? If yes, route to a custom agent plus a skill. Second: does the task require a distinct identity or tone, the way a rigorous security reviewer differs from a friendly doc writer? If yes, custom agent plus skill. Third: is this a multi-stage job with handoffs between agents? If yes, a custom agent pipeline. Fourth, only if all three were no: does a SKILL.md already cover this domain? If yes, generalist plus skill. If no, generalist alone, or write the skill first.

The tree lands on three outcomes. Custom agent plus skill, for work that needs its own .agent.md with unique identity, exclusive tools, and a defined scope: a security agent with a Vault MCP server, a test agent with CI tools, a modernization agent with Terraform. Generalist plus skill, when a skill already carries the knowledge and no specialized identity is required: writing a DOCX, generating a diagram, analyzing a CSV. Generalist alone, for work that needs no specialized knowledge at all: a factual question, a quick calculation, a brainstorm. The evaluation takes well under a minute per task. The point is consistency: the same task type routes the same way every time, for every person on the team.

> DEFINITION
>
> Four questions, three outcomes, under a minute per task. Exclusive tools, a distinct identity, or pipeline handoffs each send the task to a custom agent plus a skill. A covered domain sends it to the generalist plus a skill. Nothing special sends it to the generalist alone. Routing by a tree, not by feel, is what makes cost and quality repeatable.

Custom agents in GitHub Copilot: .github/agents, agent mode, and the picker

In GitHub Copilot the contract lives in the repository. Custom agents are Markdown files with YAML frontmatter under .github/agents/, each named with the .agent.md extension. The frontmatter declares the agent name, its description, its allowed tools array, and the model it runs on. The body is the system prompt: identity, the ordered list of what the agent looks for, and the output format. One agent is active at a time, and when you invoke it in agent mode it replaces the default chat persona and works only against the tools you allow. The repo-wide rules in .github/copilot-instructions.md still apply underneath, so an agent inherits the workspace baseline and adds its specialization on top.

Choosing the model, and paying for it

The tools array is where least privilege is enforced, and the model field is where you choose the engine. The Copilot picker exposes several models, and the routing

mindset continues inside the agent. Reach for Claude Opus 4.8 or Claude Fable 5 on the hardest reasoning: the multi-file refactor, the security review that has to be right. Sonnet 5 is the balanced default for most day-to-day agent work. Haiku 4.5 is the fast, low-cost option for high-volume, well-scoped tasks where latency matters more than depth. Gemini 3.1 Pro and the GPT-5.x models are there when a task suits their strengths. Match the model to the task the same way you matched the agent to the task: capability where it pays, speed and cost where it does not.

Billing follows this choice. GitHub Copilot meters agent work in GitHub AI Credits, where one credit equals one US cent, in effect since June 1 2026. A heavier model on a heavier task spends more credits, so the routing tree and the model picker are the same cost-control lever seen from two angles. One hour spent building the four-question tree can cut average token spend by about 30% and steady output quality across the team, which is the cheapest infrastructure work most teams have on the table.

> DEFINITION
>
> Custom agents live at .github/agents/*.agent.md: frontmatter for name, tools, and model; body for the system prompt. Agent mode swaps in that persona while .github/copilot-instructions.md stays underneath. Pick the model per task from the Copilot picker: Opus 4.8 or Fable 5 for the hardest reasoning, Sonnet 5 for the balanced default, Haiku 4.5 for fast high-volume work, Gemini 3.1 Pro and GPT-5.x where they fit. Copilot meters it in GitHub AI Credits at one credit per US cent.

References

Primary sources for the two agent types, the routing tree, the .agent.md contract, and model selection in GitHub Copilot.

- GitHub Copilot documentation, the reference platform for agent mode, custom agents, and the configuration surface this chapter routes across.
- GitHub Docs, Choosing the right AI model for your task, the official guidance behind matching a Copilot picker model to the work in front of you.

- Customize AI in Visual Studio Code, how agents, skills, and prompt files are authored and invoked in the editor.
- FeatureBench, Benchmarking Agentic Coding for Complex Feature Development, the capability gap between well-scoped bug fixes and real feature work that motivates routing to specialists.
- Vasilopoulos, Codified Context Infrastructure, the instrumented case study behind the specialist agent line counts and the domain-knowledge share of each 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 EXECUTION AND GOVERNANCE

Orchestration, handoffs, and the rails.

A GitHub Copilot workspace at scale is not one agent answering one question. It is an orchestrator that receives every request, routes it, and coordinates a chain of specialists behind a single conversation. This chapter describes the runtime that makes that work: four phases from entry to delivery, handoffs that accumulate context instead of losing it, hooks that enforce policy deterministically, and a configuration hierarchy that decides who is allowed to set which rule. The agents do the work. The rails decide whether that work is safe to ship.

The four phase runtime, and the one agent the user talks to

Every request through a Copilot workspace passes four phases. Entry: the orchestrator receives the prompt, loads its own .agent.md identity, and classifies the task type. Analysis and routing: it searches for a matching SKILL.md and evaluates the decision tree, choosing between a generalist agent with a skill and a custom agent with dedicated tools. Execution: skills load, .github/copilot-instructions.md and any scoped .instructions.md apply automatically, MCP tools connect, and the task runs with full context. Validation and delivery: the output is checked against the skill's own checklist, a quality gate reviews format, content, and conventions, and only then does the response reach the user.

The routing decision happens once, in phase two, not on every turn. What matters more than the phases is the shape they enforce: the orchestrator is the only agent the user ever talks to. Every sub-agent runs behind it. Every handoff returns to the orchestrator, which decides the next step. Agents never talk to each other directly. That single rule is what keeps a multi-agent system debuggable. When a result is wrong, there is one place to look, one context to inspect, and one actor accountable for assembling the final answer. In the Copilot picker the orchestrator can also route by cost, sending cheap scans to Haiku 4.5, generation to Sonnet 5, and hard analysis to Opus 4.8.

Handoffs that accumulate context, not lose it

The clearest multi-agent pattern is a four stage pipeline: Plan, Code, Test, Security. The planning agent receives the user request and returns a structured plan with acceptance criteria. The coding agent receives the request plus the plan and returns the implementation with the list of files it touched. The testing agent receives the request, the plan, and the code, and returns tests with their execution results. The security agent receives the complete picture, request, plan, code, and test results, and returns a findings report. Each stage sees everything the stages upstream produced.

This is the property that makes the pipeline work. Context accumulates at every handoff. A handoff passes not only the new input but all prior artifacts, so no stage starts from a blank slate and no earlier decision is silently dropped. The security agent can only do its job if it sees the plan the code was meant to implement and the tests that were supposed to cover it. Strip that history and security review becomes guesswork.

Typed artifacts, strict order

Two constraints keep the chain honest. Each agent emits a typed artifact, a plan, a diff, a test suite, a report, that becomes the input of the next stage, so a handoff is a contract, not a conversation. And the sequence is strictly ordered: each agent finishes before the next begins. The orchestrator holds the accumulated context and hands the right slice to the right specialist. Hooks wrap each handoff and log what crossed the boundary, so the whole chain can be audited after the fact.

Hooks as deterministic gates

Skills and agents shape what the model tends to do. Hooks decide what actually happens, regardless of how the model behaves. They live in .github/hooks/*.json and fire on six GitHub Copilot lifecycle events: sessionStart when a session begins or resumes, userPromptSubmitted when the user submits a prompt, preToolUse before the agent runs any tool, postToolUse after a tool completes, sessionEnd when the session closes, and errorOccurred when something fails. Only preToolUse can block: it can deny a tool call before it runs. The other five observe, log, and react. Hooks

from multiple files and plugins merge rather than override, and for the cloud coding agent the hook files must live on the repository's default branch to load.

Four hooks belong in every rollout on day one, because the model will not add them for you. An empirical study of 2,303 agent context files across 1,925 repositories found explicit security guidance in only 14.5% of them (Chatlatanagulchai et al. 2025). Hooks are the deterministic backstop for the 85.5% that leave it out. A secrets block on preToolUse denies any write or commit that carries a credential. A formatter on postToolUse runs the linter on every file the agent edits, so style never becomes a review argument. An audit log, driven from sessionStart and postToolUse, records every action, model version, and rationale, so a change can be explained and reversed. A policy gate on preToolUse denies destructive commands, the rm, the DROP TABLE, the kubectl delete against production, and pages a human instead.

> DEFINITION
>
> The four day-one hooks: a secrets block and a policy gate on preToolUse that deny before damage, a formatter on postToolUse that ends style debates, and an audit log across sessionStart and postToolUse that makes every agent action explainable and reversible. Deterministic rails, not model goodwill.

Configuration governance: floors, contracts, and an absolute deny

The last rail is who is allowed to set which rule, and the model is one directional by design. The organization sets the floors through GitHub Enterprise Copilot policies: content exclusion, telemetry, model access, and hook enforcement that a single repository cannot loosen. The workspace sets the team contract in .github: copilot-instructions.md, scoped .instructions.md, hooks, and the .vscode/settings.json that every clone inherits. The user personalizes on top, editor preferences and local settings that never leak into a commit. Each layer refines the one above it. None can contradict it.

One rule sits above all the others: deny is absolute. A lower layer can add permission where it was given room, but it can never override a deny set higher up. A skill

cannot bypass a policy. A prompt cannot re-enable a command the org blocked. A user setting cannot weaken a workspace guardrail the team committed. This is what makes the hierarchy a governance model rather than a pile of config files. It also keeps agents accountable when they act on money: agent runs meter against GitHub AI Credits, where one credit is one US cent since June 2026, so a ceiling per module stops a runaway loop before the invoice does, and MCP exposes enterprise systems through one audited interface rather than scattered credentials.

References

Primary sources for the orchestration runtime, the Copilot hook lifecycle, the empirical security gap, and the configuration governance model in this chapter.

- GitHub Copilot, Hooks configuration reference, the six lifecycle events and the JSON schema every hook file follows.
- GitHub Copilot, Using hooks with the coding agent, why hook files must live on the default branch to load for the cloud agent.
- GitHub Copilot in VS Code, Agent hooks, the local hook model for preToolUse and postToolUse gates.
- GitHub Copilot, Repository custom instructions, the copilot-instructions.md and scoped instruction files that form the team contract.
- Chatlatanagulchai et al., An Empirical Study of Agent Context Files, the finding that only 14.5% of 2,303 files across 1,925 repositories carry explicit security guidance.
- GitHub Copilot documentation, the reference platform for agent mode, the model picker, and MCP.
- Model Context Protocol Specification, the open standard for exposing enterprise systems to agents through one audited interface.

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 ADOPTION AND CLOSE

Anti-patterns, diagnostics, and 90 days: from routing theory to a repeatable practice.

Agreeing with routing theory is easy. Keeping it is hard. This chapter is the adoption layer: four anti-patterns that appear before any good practice takes hold, six diagnostic questions that surface gaps in any workspace, and a 90-day plan that turns the four-question decision tree into a habit. It closes with the one number that pays for all of it.

Four anti-patterns that show up before any best practice

Every workspace I have watched scale GitHub Copilot passed through the same four anti-patterns first. The first is the skill as megaprompt: one SKILL.md grows to two thousand lines, covers the whole domain, and activates on every glob match. Token overhead per call then exceeds the value of the knowledge it carries, now a readable line in GitHub AI Credits. The fix is to split it into three targeted skills, each narrow enough to pass the SFA test for Scope, Format, and Audience. The second is the agent for everything: one custom agent that holds every permission and refuses no task. It works until the first incident, when the blast radius is the whole workspace. A custom agent needs a bounded scope in its .agent.md and a tool manifest it does not exceed, because in agent mode whatever that agent reaches through MCP is what an attacker inherits if it is compromised.

The third is skills that are never versioned: SKILL.md files edited in place, output quality drifting, no changelog and no rollback when a change makes results worse. Treat every SKILL.md like code: version it, review it in a pull request, test it, promote it. The fourth is routing by intuition, where each developer decides per call, by feel, whether a task needs a custom agent, a skill, or neither. The result is inconsistent output, inconsistent AI Credits spend, and no baseline to measure against. The four-question decision tree removes the guesswork, and it takes about one hour to agree

on as a team. These failures are cheap to prevent and expensive to unwind, which is why they belong at the front of any adoption plan.

> DEFINITION
>
> The four anti-patterns, in order of appearance: skill as megaprompt, agent for everything, skills never versioned, and routing by intuition. The first inflates AI Credits, the second the blast radius, the third erases the audit trail, and the fourth removes the baseline.

Six diagnostic questions for any workspace

Before you plan any change, measure where you stand. Six questions surface the gaps in any Copilot workspace, and a no to any one of them is worth addressing before the next sprint. Do you know every skill active in your workspace right now? Does every custom agent have an .agent.md with an explicit tool list? Can you trace which skill was loaded for any given task output? The first three questions are about visibility. If you cannot answer them, you are running without instrumentation, and no optimization survives contact with a system you cannot see.

The last three are about discipline. Is there a routing decision tree the whole team follows, or does each member decide ad hoc? Are all skills version-controlled and reviewed before they are distributed? And the one number worth checking directly: is the ratio of custom agents to skills at 1:8 or better? More custom agents than that usually means the workspace is over-engineered, paying for bespoke identities where a shared skill on the generalist agent would do the same work at a fraction of the AI Credits. Far fewer skills than agents means it is under-specialized, forcing custom agents to carry knowledge that belongs in a reusable SKILL.md. The 1:8 ratio is the fastest read on either kind of drift.

The 90-day playbook: Foundation, Contract, Measure

Teams that scale Copilot value follow the same three-phase progression, and each phase ends with one concrete outcome. Foundation runs from day 1 to day 30: run a full skill inventory and score every SKILL.md against the SFA test, adopt the four-question decision tree as a team, review the .agent.md of every custom agent, and enable skill activation logging. Fold repository-wide conventions into .github/copilot-instructions.md so the generalist agent starts every task from the same ground rules. The one outcome of Foundation is a routing baseline: a written record of what exists and how work gets assigned.

Contract runs from day 31 to day 60: promote your top five skills to maturity level 3, the level that carries all five sections, scripts, and tuned glob patterns, then stand up the first pipeline in agent mode, with plan, code, and test handing off through typed contracts, and start measuring AI Credits per task. The one outcome of Contract is a first pipeline delivering typed artifacts, where each handoff accumulates context instead of losing it. Measure runs from day 61 to day 90: publish a shared skill library, apply the decision tree to at least 80% of tasks, track token efficiency weekly in AI Credits, and document a replicable process for adding skills. The one outcome of Measure is a skill library that is operational and quality that is measurable. The point of 90 days is not perfect skills. It is a repeatable process for making them better.

What one hour of routing design returns

Here is the number that pays for the whole exercise. Writing the four-question decision tree takes about one hour. In workspaces that adopt it, average token spend per task drops by roughly 30%, because tasks a skill on the generalist agent can answer stop being routed to a custom agent that pulls three times the context. In GitHub Copilot that saving is denominated in GitHub AI Credits, where one credit has equaled one US cent since June 1, 2026. A 30% cut in token spend is a 30% cut in the AI Credits line, readable in the usage dashboard.

The second return is consistency, harder to reduce to a number and easier to feel. When routing is deliberate, the same kind of task takes the same path every time, loads the same skill, and passes the same validation checklist before delivery.

Output stops depending on which developer phrased the prompt, and review burden falls because reviewers meet fewer surprises. One hour of design buys both: a measurable cut in AI Credits and more predictable output across the team. That is the close of this playbook. Routing is the first engineering decision you make with GitHub Copilot, and the one that keeps paying.

> DEFINITION
>
> One hour of routing design returns roughly 30% less token spend, measured in GitHub AI Credits, and more consistent output across the team. Routing is the first engineering decision you make with GitHub Copilot, and the cheapest one to get right.

References

Primary sources for the routing model, the capability gap on real feature work, the security surface of agent skills, and the GitHub Copilot customization primitives this chapter builds on.

- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, the capability gap on real feature work that domain skills and bounded custom agents exist to close.
- Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale, empirical grounding for why the agent-for-everything anti-pattern is dangerous and why every SKILL.md needs version control and review.
- Customize AI in Visual Studio Code, the reference for the SKILL.md, .agent.md, and .instructions.md primitives this playbook routes across.
- GitHub Copilot documentation, the reference platform for agent mode, MCP connections, and the GitHub AI Credits billing model.

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