CONFIGURATION CONTEXT ARCHITECTURE · PLAYBOOK

One file covers 10%. Ten layers across three tiers make Copilot auditable.

Most teams turn on GitHub Copilot in agent mode, write a single .github/copilot-instructions.md, and stop. That one file covers roughly 10% of the configuration surface VS Code exposes for Copilot. This playbook maps the other 90%: ten configuration layers grouped into three memory tiers, hot, warm, and cold, so every rule loads when it is relevant, nothing gets truncated, and Copilot behavior becomes auditable, versioned, and owned instead of tribal.

AUTHOR

Paula Silva

ROLE

AI-Native Software Engineer

CONTACT

in/paulanunes

READING TIME

~ 45 minutes, end to end

10	3	6
CONFIGURATION LAYERS	MEMORY TIERS	CHAPTERS

10%

SURFACE ONE FILE COVERS

CHAPTERS

Chapter 00

Ten layers, not one file

The 10% trap, the 2,303-file empirical study and the 14.5% security blind spot, and why single-file manifests collapse past 3,000 tokens in a 108,000-line, 283-session codebase.

Problem →

Chapter 01

The map: three tiers across ten surfaces

The ten configuration layers in the order VS Code loads them, the hot, warm, and cold memory model from Vasilopoulos 2026, and the one rule for deciding where each rule of your own belongs.

Map →

Chapter 02

The workspace layer: what is always true

copilot-instructions.md kept under 1,000 tokens, the Security and Performance sections most teams skip, and the applyTo glob that keeps the hot file short across four scoping patterns.

Workspace →

Chapter 03

The authoring layer: what you invoke

Agents, chat modes, skills, and prompt files, chosen by how each one is triggered rather than what it contains, with least-privilege tool scoping and per-agent model selection.

Authoring →

Chapter 04

Runtime governance: what runs around the agent

Hooks as deterministic guardrails, MCP connections and plugins for fresh data and versioned bundles, settings precedence, and the nine-step assembly order that tells you why a rule was ignored.

Runtime →

Chapter 05

From zero to baseline in one week

One commit per day from copilot-instructions.md to a shared plugin, honest expectations for what the hierarchy will and will not do, and governance instead of prompt engineering.

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

Ten Layers, Not One File.

Most teams turn on GitHub Copilot in agent mode, write a single .github/copilot-instructions.md, and consider the setup finished. That one file covers roughly 10% of the configuration surface that VS Code exposes for Copilot. The other 90% stays empty. In a small repository the gap is invisible. Across a multi-thousand-repository enterprise estate it becomes the reason Copilot behavior stays tribal instead of auditable. This chapter names the failure, grounds it in two empirical studies, and shows why one file cannot hold an enterprise codebase. The fix is not a bigger file. It is a layered surface.

The 10% trap

The pattern repeats across teams. Someone activates GitHub Copilot in agent mode, writes one .github/copilot-instructions.md, and stops. The file lists the stack, a handful of conventions, maybe the build command. It reads well in a demo. In practice it covers about 10% of the configuration surface that VS Code exposes for Copilot. The remaining 90% stays unconfigured, and unconfigured surface is where behavior drifts. A small repository hides the cost. A monorepo of dozens of packages, replicated across hundreds of repositories, does not.

The missing 90% is not one thing. It is nine more surfaces: scoped instructions with applyTo globs, agents, chat modes, skills, prompt files, hooks, MCP connections, plugins, and VS Code settings. Each answers a question the single file cannot. Which rules apply only to the payments folder. Which workflow the team runs on every release. Which command Copilot must never run against production. When those questions have no written answer, the model fills the gap with a guess, and the guess stays invisible until a reviewer catches it in a pull request.

The empirical gap: 2,303 context files and the 14.5% blind spot

The gap is measurable. A 2025 study analyzed 2,303 real agent context files across 1,925 repositories (Chatlatanagulchai et al., arXiv 2511.12884). The pattern is consistent. Teams write what makes the agent functional. Build and run commands appear in 62.3% of files. Implementation details appear in 69.9%. Architecture appears in 67.7%. These are the facts that get an agent producing code on the first attempt, so they get written first.

What teams skip is the set of rules that keep that code safe. Security guardrails appear in only 14.5% of files. Performance rules appear in another 14.5%. Error handling and degraded behavior are rare. The result is an agent that writes productive code but not predictable code. The problem is not the model. Copilot lets you pick Claude Fable 5, Opus 4.8, or GPT-5.x from the same model picker, and none of them can enforce a rule that was never written down. The 14.5% security blind spot is a configuration gap, not a model gap. Swapping models does not close it. Writing the missing rules does.

Why single-file manifests collapse

The scaling argument has a case study behind it. One documented project built 108,000 lines of C# over 70 days and 283 Copilot sessions (Vasilopoulos, arXiv 2602.20478). It found that single-file manifests collapse early. A flat .github/copilot-instructions.md that grows past roughly 3,000 tokens gets truncated by the model. Everything after the cutoff is dropped without warning. The team keeps adding rules. The agent keeps reading fewer of them. The instructions look complete in the editor and are incomplete by the time the model sees them.

The fix in that project was a tiered structure: hot memory loaded on every request, warm memory loaded on trigger, cold memory queried through MCP. That structure held a knowledge-to-code ratio of 24.2% across a 14-package monorepo for all 283 sessions. Tiering also protects the bill. Copilot in agent mode meters usage in GitHub AI Credits, one credit equal to one US cent. A truncated instruction still consumes input tokens on the way to the model. Paying to send context that the model then discards is the worst of both outcomes: you are billed for the rule and the agent

never applies it. Tiering turns that waste into reuse: context computed once, loaded only where it earns its place.

> DEFINITION
>
> One file is a prototype tool. An enterprise codebase needs ten configuration layers across three tiers, hot, warm, and cold, so that every rule loads when it is relevant and no rule gets truncated for lack of room.

Three symptoms of an underused surface

You do not need a study to detect the problem. Three symptoms show up in daily work. Duplicate instructions: the same rule lives in three files, and Copilot follows whichever one the open file happens to match. Silent drift: the model generates useful code that ignores a team convention, because the convention exists only in someone's head and never in a file the model can read. Tribal knowledge: a strong prompt for building an API endpoint works, but only one person on the team knows it exists.

Each symptom points to a specific layer. Duplicate instructions mean the rule belongs in a scoped .github/instructions file with an applyTo glob, not copied into the root manifest. Scoped instructions merge with the workspace file, they do not replace it, so the root stays short while the rule loads only where it applies. Silent drift means a convention was never codified in the hot tier. Tribal knowledge means a workflow should be a skill or a prompt file, addressable by the whole team. The fix is structural, not conversational. Map each rule to the layer that owns it. The goal is not more configuration. The goal is configuration that is auditable, versioned, and unambiguous about who owns each layer.

References

Primary sources for the ten-layer surface, the empirical gap, and the case study behind the single-file failure.

- Agent READMEs: An Empirical Study of Context Files for Agentic Coding, the empirical gap: 2,303 context files, with security guardrails present in only 14.5% of them.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, the 108,000-line, 283-session case study for why single-file manifests collapse.
- Customize AI in Visual Studio Code, the reference for the ten configuration surfaces mapped in this chapter.
- GitHub Copilot Documentation, the platform behind agent mode, copilot-instructions.md, and the model picker.
- Model Context Protocol, the open standard that carries the cold tier of context to the agent.

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

Three tiers across ten surfaces: the map of every GitHub Copilot configuration layer.

Every GitHub Copilot response in VS Code is assembled from ten configuration surfaces, not from a single instruction file. Those ten surfaces collapse into three memory tiers, and the three tiers answer three questions about the codebase. This chapter is the map. It names the ten layers in the order Copilot loads them, groups them into the hot, warm, and cold tiers documented by Vasilopoulos 2026, and gives you the one rule that decides where each rule of your own belongs.

The ten configuration layers, in the order Copilot loads them

Every GitHub Copilot response is built from a layered context window, and there are ten surfaces that feed it. The workspace instruction file at .github/copilot-instructions.md loads always. Scoped instructions at .github/instructions/*.instructions.md load on a path match against an applyTo glob. Agents load when invoked by name, chat modes when selected, skills on a description match, and prompt files on a slash command. Hooks fire on lifecycle events. MCP connections attach per session. Plugins load when installed. VS Code settings apply always. Most teams write one of these, copilot-instructions.md, and stop, which covers roughly ten percent of the surface. The other ninety percent stays unconfigured, and that is why Copilot behavior stays tribal instead of auditable.

The order is not cosmetic, because Copilot assembles the context window deterministically. VS Code resolves settings first, default then user then workspace, with the most specific scope winning. It loads copilot-instructions.md unconditionally. It matches applyTo globs against the open files and merges every instruction set that hits. It attaches the active authoring primitive, whether that is an agent, a chat mode, a skill, or a prompt file. It runs the on-prompt-submit hooks,

which can deny, mutate, or annotate the request. It discovers the tools exposed by connected MCP servers. Then it dispatches the assembled system prompt, the tool catalog, and the user message to the model selected in the Copilot picker. Later layers refine earlier ones; they cannot contradict them. A skill cannot bypass an organization policy, and a prompt cannot override an enterprise admin setting.

Debug from the top down

This order is also a debugging tool. When Copilot ignores a rule, the fault is almost always in one of three places: the wrong settings scope, an applyTo glob that does not match the file you have open, or a hook that blocked or mutated the instruction before it reached the model. Walk the assembly order from the top before you conclude the model is at fault. The sequence is reproducible, so the bug is locatable.

Hot, warm, cold: the three-tier memory model from Vasilopoulos 2026

The ten surfaces group into three memory tiers, a taxonomy documented by Vasilopoulos (2026, arXiv 2602.20478). That case study built and instrumented a 108,000-line C# system over 70 days and 283 sessions across a 14-package monorepo, and held a knowledge-to-code ratio of 24.2 percent. The finding that matters here is that a single-file manifest collapses well before that scale. A flat copilot-instructions.md that grows past roughly 3,000 tokens gets truncated by the model, and everything after the cutoff drifts silently. The fix is not a longer file. The fix is tiers.

Hot memory is layers 01 and 02 plus the Copilot-related VS Code settings: always loaded, on every request. Warm memory is layers 03 through 07, the scoped instructions, agents, chat modes, skills, prompt files, and hooks, loaded on demand when a trigger fires by name, selection, glob, description match, slash command, or lifecycle event. Cold memory is layers 08 and 09, the MCP connections and plugins, queried explicitly when the agent reaches for data that lives outside the repository. Hot memory rides on every turn, so every token in it is paid on every request. Since June 1 2026, GitHub Copilot bills premium usage in GitHub AI Credits, one credit equal to one US cent, which turns an overweight hot tier into a recurring line item

rather than a one-time cost. Warm and cold memory are cheap because they load only when the work calls for them.

> DEFINITION
>
> Hot memory is expensive because it loads on every request and its tokens are billed every turn. Warm and cold memory are cheap because they load only when triggered. The mistake most teams make is putting everything in hot.

Workspace, authoring, runtime: the three responsibilities each tier serves

The three tiers also map onto three responsibilities, and naming them makes ownership obvious. The workspace layer answers the question "what is always true about this codebase." It is copilot-instructions.md plus the scoped .github/instructions/*.instructions.md files driven by applyTo globs. Together they are the team contract, and they load on every request. A path-scoped subset refines the global rules without overriding them; when several files match the same path, they merge rather than replace. That is how a React rule scoped to src/components and the global workspace rules load side by side.

The authoring layer answers the question "what I invoke." Agents, chat modes, skills, and prompt files are reusable knowledge packaged as artifacts and bound to a task, not to a file. The selection rule is simple: pick the primitive by how it is triggered, not by what it contains. An agent is invoked by name for a whole class of tasks such as review or test. A chat mode switches the persona for the length of a conversation. A skill auto-loads when the user request matches its description. A prompt file runs behind a slash command the user types on purpose.

The runtime layer answers the question "what runs around the agent." Hooks intercept every tool call, before and after, to block, log, or rewrite it. MCP connections extend the agent reach to external systems, databases, issue trackers, observability, internal APIs, through one protocol discovered at session start. Plugins bundle third-party capabilities behind a single manifest. VS Code settings dial the deterministic behavior of the model. All four surfaces are owned by the platform

team, set once and changed rarely. Editing runtime configuration per feature or per sprint is the anti-pattern that this layer exists to prevent.

The rule: promote to hot only what is true on every line of the workspace

One rule decides where a given rule of your own belongs. Reserve the hot tier for what is true on every line of code in the workspace: naming conventions, security policy, performance baselines. If a rule applies to only one task, one file type, or one directory, it does not belong in hot; it belongs in the warm tier as a scoped instruction, an agent, a skill, or a prompt file. If it configures the environment around the agent rather than the prompt content, it belongs in the cold tier. The corollary is operational: enforce policy at the outer scope and do not police it per user. Lock the contract at the organization or workspace level, and let inner scopes add ergonomic preferences that cannot undermine it.

The empirical gap is where this rule earns its keep. A study of 2,303 real context files across 1,925 repositories (Chatlatanagulchai et al., 2025, arXiv 2511.12884) found that teams write what makes the agent functional and skip what makes it safe. Build and run commands appear in 62.3 percent of files, implementation details in 69.9 percent, and architecture in 67.7 percent, but security guardrails appear in only 14.5 percent and performance rules in another 14.5 percent. The result is an agent that produces productive code that is not necessarily safe or predictable. Security and performance are exactly the rules that hold on every line of the workspace, so they belong in the hot tier by the rule above, and their absence there is the most common and most expensive gap a Copilot rollout leaves open.

> DEFINITION
>
> Promote to hot only what is true on every line of the workspace: conventions, security policy, performance baselines. Everything task-specific stays warm. Everything that configures the runtime stays cold. Enforce at the outer scope; do not police per user.

References

Primary sources for the three-tier memory model, the empirical gap in real context files, and the configuration surfaces this chapter maps.

- Vasilopoulos, Codified Context: Infrastructure for AI Agents in a Complex Codebase, the source for the hot, warm, cold memory model: a 108,000-line system over 283 sessions at a 24.2 percent knowledge-to-code ratio.
- Chatlatanagulchai et al., Agent READMEs: An Empirical Study of Context Files for Agentic Coding, the empirical gap across 2,303 context files: 62.3 percent cover build and run, only 14.5 percent cover security.
- Customize AI in Visual Studio Code, the reference for how the configuration surfaces load and how settings scopes resolve.
- GitHub Copilot Documentation, the reference platform for the agent mode, instruction file, and MCP behavior in this chapter.
- Model Context Protocol Specification, the open standard behind the cold-tier MCP connections.

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 WORKSPACE LAYER

What is always true: the two files that load on every request.

The workspace layer answers one question: what is always true about this codebase. It is the foundation every GitHub Copilot project needs, and it is two files with one job each. The first is copilot-instructions.md, loaded on every request. The second is a set of scoped instruction files that load only where they apply. Get this layer right and Copilot onboards the next contributor for you. Get it wrong and every review catches the same avoidable mistakes.

copilot-instructions.md: the file the model reads every time

copilot-instructions.md lives at .github/copilot-instructions.md. It loads on every GitHub Copilot request: chat, edits, and agent mode. It is plain Markdown with no frontmatter, and there is exactly one file per workspace. No cascading, no second copy that overrides it. This is the highest-impact file you can write for Copilot, because it is the only content that reaches the model on every single line of code.

The constraint is size. Keep it under 1,000 tokens. Past that budget, Copilot summarizes the file and you lose determinism: the model reads a compressed version, and you can no longer predict which rules survive. A flat file that grows past 3,000 tokens gets truncated outright, and everything after the cutoff drifts silently. Reserve this file for what is true everywhere: the stack, the conventions, the security policy, the performance baseline. If a rule applies to only one task or one folder, it does not belong here.

Why one file stops scaling

One documented C# project of 108,000 lines, built over 70 days across 283 sessions in a 14-package monorepo, found that single-file manifests collapse early (Vasilopoulos, 2026). The project held a knowledge-to-code ratio of 24.2% only

after it moved to a tiered memory model. The lesson is not that copilot-instructions.md is weak. The lesson is that it has a job and a budget, and everything beyond that budget belongs in the layers below it.

A working example, with the Security and Performance sections most teams skip

Here is what the file looks like when it is short and specific. It names the stack in one line: Next.js 14 App Router, TypeScript in strict mode, Tailwind, Drizzle ORM over PostgreSQL, Vitest. It states conventions as rules the model can follow: files in kebab-case, React components in PascalCase, tests co-located next to the code they cover, no direct imports from node_modules. It lists the key commands for dev, build, lint, and test. None of that is unusual.

What is unusual is the two sections most teams leave out. A Security section: never log request bodies or auth headers, validate every external input before using it, and check the session before any database write. A Performance section: prefer server components, keep payloads small, and paginate long lists. These are the rules that make Copilot output safe and predictable, not just productive. They cost four lines each and they close the gap that produces most rejected suggestions.

A study of 2,303 real context files across 1,925 repositories measured exactly this gap (Chatlatanagulchai et al., 2025). Build and run commands appeared in 62.3% of files, implementation details in 69.9%, architecture in 67.7%. Security rules appeared in only 14.5%. Performance rules in another 14.5%. Error handling was rarer still. The default failure mode of a Copilot rollout is not a missing context file. It is a context file that makes the agent productive and leaves it unsafe.

> DEFINITION
>
> The two sections that close the gap are the two most teams never write. Only 14.5% of real context files include security rules, and only 14.5% include performance rules. Adding a Security block and a Performance block to copilot-instructions.md is the cheapest correction available.

Scoped instructions: the applyTo glob that keeps the hot file short

The workspace file stays under budget by moving specialized guidance out of it. Scoped instructions live in .github/instructions/*.instructions.md. Each file carries an applyTo glob in its frontmatter, and Copilot loads the file only when an open file matches that glob. A file scoped to applyTo: "src/components/**/*.tsx" reaches the model when someone edits a component and stays silent otherwise. The detailed rules exist, but they only cost tokens when they are relevant.

The rule that makes this safe is that instructions merge, they do not override. Multiple instruction files can match the same path, and all of them load together with the workspace file. Nothing replaces the global contract; scoped files refine it. This is how you keep copilot-instructions.md small without losing the detail a specific stack needs. The React rules live in one file, the SQL rules in another, and the hot file only carries what is true everywhere.

When a scoped rule does not load

When Copilot ignores a scoped rule, the cause is almost always the glob. Check that the applyTo pattern actually matches the file you have open before concluding the model is at fault. A pattern that targets src/components will not fire on a file under app/components, and a rule written for the wrong extension never reaches the model at all. Walk the glob before you argue with the agent.

Four scoping patterns, including the risk-surface pattern most teams miss

Four patterns cover most repositories. The first scopes by framework layer: one file for src/components, one for src/server, one for src/db, each carrying the rules that belong to that layer. The second scopes by file type: testing conventions on **/*.test.ts, documentation style on **/*.md, query review rules on **/*.sql. The third scopes by domain or bounded context: one file per package in a monorepo, so packages/billing and packages/auth each get their own contract.

The fourth pattern is the one most teams never write, and it is the highest value. Scope by risk surface. Put PCI rules on **/payments/**. Put privileged-endpoint rules on **/api/admin/**. Put destructive-change rules on migrations/**. These are the paths where a wrong suggestion costs the most, and they map exactly onto the 14.5% security gap the empirical study measured. The workspace file cannot carry this level of detail without blowing its budget. A risk-surface instruction file can, and it only loads when someone touches the code that warrants it.

> DEFINITION
>
> The risk-surface pattern is the single scoping decision most teams miss. Scope the detailed guardrails to the paths where a wrong suggestion is expensive: payments, admin endpoints, migrations. It is the same security gap the research measured, solved at the file level instead of the prompt level.

References

Primary sources for the workspace layer, the empirical evidence, and the configuration surfaces this chapter builds on.

- Customize AI in Visual Studio Code, the reference documentation for copilot-instructions.md, scoped instruction files, and the applyTo glob.
- An Empirical Study of Context Files for Agentic Coding, the source for the 14.5% security and performance gap across 2,303 real context files.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, the 108,000-line case study on why single-file manifests collapse and tiered memory is required.
- GitHub Copilot Documentation, the reference platform for the agent mode and configuration patterns 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

03 AUTHORING LAYER

What you invoke: agents, chat modes, skills, and prompt files, and how to pick between them.

The authoring layer is where you package reusable behavior as artifacts. It holds four primitives: agents, chat modes, skills, and prompt files. All four sit in the warm tier, which means GitHub Copilot loads them only when something triggers them, never on every request. The mistake teams make is picking the wrong primitive because they compare what each one contains. The right question is how each one is triggered. This chapter walks the decision, then each primitive in turn.

The decision tree: pick by how it is triggered, not by what it holds

The four authoring primitives look similar on the page. Each is a Markdown file with frontmatter and instructions, each shapes how Copilot responds, and each lives under .github/. The temptation is to choose between them by reading their contents, but the contents overlap so much that the comparison stalls. The reliable rule is different: pick by the invocation mechanism. An agent is invoked by name. A chat mode is selected in the picker. A skill is matched automatically against your request. A prompt file runs on a slash command. Trigger, not content, is the axis that separates them cleanly.

The mechanism maps to intent. If you want a named specialist for a whole class of tasks, that is an agent. If you want to change the conversation style or the depth of explanation, that is a chat mode. If you want Copilot to reach for packaged domain knowledge on its own, that is a skill. If you want a one-shot recipe you invoke deliberately, that is a prompt file. In the assembly order, only one authoring primitive attaches per turn, at step six, after the hot-tier instructions and any scoped instructions have already loaded. So the primitive you invoke refines the workspace contract. It does not replace it.

DEFINITION

The selection rule in one line: choose the authoring primitive by how it is triggered, not by what it contains. Invoked by name is an agent. Selected in the picker is a chat mode. Matched by description is a skill. Run on a slash command is a prompt file.

Agents: named personas with their own tools and least privilege

Agent files live in .github/agents/ with the .agent.md extension. Each defines a specialized persona with its own instructions, its own behavioral scope, and, most important, its own tool allowlist. One agent is active at a time. When you invoke it by name, it replaces the default chat persona for that class of work. The tools field is what sets agents apart from every other primitive. You list exactly the tools the agent may call, and nothing else is available to it. A reviewer agent that only reads and comments does not get shell or write access. That is least privilege expressed as configuration.

The frontmatter also carries a model field, which routes the agent to a specific model in the Copilot picker. A cheap classification agent can run on Haiku 4.5. A hard refactor agent can run on Opus 4.8 or Claude Fable 5. Because billing runs on GitHub AI Credits, where one credit is one cent, routing routine agents to a smaller model is a direct cost lever, not a preference. The empirical case study on codified context found nineteen specialist agents averaging 711 lines for higher-capability agents and 327 for standard ones, with over half of each spec being project-domain knowledge rather than behavioral instruction. An agent is mostly what it knows, not how you tell it to act.

Chat modes and skills: conversation style versus auto-loaded knowledge

Chat modes live in .github/chatmodes/ and differ from agents in three ways. They carry no separate tool allowlist. They are selected rather than invoked. And they stay

active across turns until you switch out of them. A chat mode changes the style and the authority of explanation, not the capability. Teach mode annotates every suggestion line by line. Summarize mode compresses. Rubber-duck mode asks questions back. When the behavior you want is about tone, depth, or persona for a whole conversation, and not about which tools are allowed, a chat mode is the right primitive and an agent is overkill.

When to reach for a skill instead

Skills live in .github/skills/{name}/SKILL.md, each in its own folder so it can carry reference files, scripts, and templates alongside the instructions. The distinction that matters is the trigger. Copilot loads a skill automatically when your request matches the skill description in frontmatter. You do not name it and you do not select it. That makes skills the answer when the same multi-step workflow keeps getting re-explained. Write it once, describe it well, and Copilot finds it on its own. The community registry at Awesome GitHub Copilot distributes skills and prompts this way, so a strong description is what makes a skill discoverable and reusable, on your team and beyond it.

Prompt files: one-shot slash commands for known recipes

Prompt files live in .github/prompts/ with the .prompt.md extension. Unlike skills, they are never auto-triggered. The user calls them explicitly with a slash command, so /release-notes or /pr-summary runs a specific, predictable recipe on demand. Use a prompt file when the workflow is discrete, the steps are known, and the person running it already knows they want it. A prompt file can declare its own tools and its own model in frontmatter, the same as an agent, but it is a single invocation rather than a persistent persona. It does one job and returns.

The line between a skill and a prompt file is the trigger, again. A skill is for knowledge Copilot should reach for on its own when the moment fits. A prompt file is for a recipe a human deliberately runs. If you catch yourself wishing Copilot had known to do something, that is a skill. If you keep typing the same long instruction to make it do a known task, that is a prompt file. Both keep the hot tier lean by living in the

warm tier and loading only when triggered, which keeps the always-loaded context small and the AI Credits spent per turn predictable.

References

Primary sources for the authoring primitives, their invocation mechanisms, and the empirical evidence behind agent and skill design.

- Customize AI in Visual Studio Code, the reference documentation for agents, chat modes, skills, and prompt files in VS Code.
- Agent READMEs: An Empirical Study of Context Files for Agentic Coding, the source for the 14.5% of context files that carry explicit security guidance.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, the source for the 711 and 327 lines-per-agent benchmark across nineteen specialist agents.
- GitHub Copilot documentation, the reference platform for agent mode and the authoring primitives in this chapter.
- Awesome GitHub Copilot, the community registry that distributes skills, prompts, and agents as shareable artifacts.

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 RUNTIME GOVERNANCE

What runs around the agent: the runtime layer that governs every Copilot response.

The first parts of this playbook describe what GitHub Copilot knows: the workspace instructions it always reads, and the authoring artifacts it loads on demand. This chapter describes what runs around the agent while it works. Four runtime surfaces (hooks, MCP connections, plugins, and VS Code settings) decide what the agent is allowed to do, where it gets fresh data, and how its behavior is locked across an organization. None of them carry prompt content. They govern the loop. Owned by the platform team, set once and changed rarely, they are the difference between agent mode that is productive and agent mode that is safe.

Hooks: deterministic guardrails on every tool call

Hooks live in .github/hooks/*.json. They fire on lifecycle events (preToolUse, postToolUse, onSessionStart, onPromptSubmit, onFileChange) and can allow, deny, mutate, or log an event before or after Copilot acts. A hook is code that runs around an agent action, not inside it. It does not ask the model to behave. It enforces the rule deterministically, the way CI enforces a build. A preToolUse hook that matches a destructive shell command and returns deny stops "clean up the cluster" from becoming kubectl delete namespace prod. The model never gets the chance to be clever.

Hooks matter most because of a measured gap. Across the 2,303 real agent context files studied by Chatlatanagulchai and colleagues in 2025, explicit security guidance appears in only 14.5% of repositories. That leaves 85.5% with no written security rule the model could follow. Hooks are the deterministic backstop for that 85.5%. Where an instruction file says "please do not," a hook simply refuses. Five lines of JSON on a preToolUse event prevent the incident a team has not had yet, and unlike

a prompt, the guarantee does not depend on which model sits in the Copilot picker or how the request was phrased.

MCP connections and plugins: fresh data and versioned bundles

MCP, the Model Context Protocol, is the open standard Copilot uses to reach external systems at runtime: databases, issue trackers, observability platforms, internal APIs. Servers are declared in .mcp.json for the workspace or in settings.json for the user, discovered once per session, and each exposes tools, resources, and prompts the agent can call. The rule for when to add one is simple. If the data lives outside the repository and changes between sessions (open tickets, log lines, rows in a table), it belongs behind an MCP server, not pasted into chat. If the action a server exposes is destructive, wrap it with a preToolUse hook. Every MCP call spends latency and GitHub AI Credits, so least-privilege credentials and a narrow tool surface are part of the configuration, not an afterthought.

Plugins solve distribution. A plugin bundles agents, chat modes, skills, prompt files, hooks, and MCP declarations behind a single plugin.json manifest, pinned to a version. It is how one standard travels across many repositories without copy-paste. The moment a team finds itself copying .github/ contents from one repository into another, the drift has already started: forty repositories with forty hand-edited instruction files diverge within a quarter. One plugin, pinned per repository, turns that into a single pull request against the plugin. The community registry, Awesome GitHub Copilot, distributes shared work the same way. Plugins are cold configuration: installed once, versioned, and changed through review, never edited per feature.

Settings precedence: the team contract flows down, personal taste flows up

Every Copilot behavior that is not a file is a setting, and settings resolve in a strict order, from least specific to most specific: default (built into the product), user (~/.../User/settings.json, never committed), workspace (.vscode/settings.json, committed to the repository), and workspace folder (for one package in a multi-root

setup). The most specific scope wins, with one exception that matters. At the top sits GitHub Enterprise policy, which sets floors: content exclusion, telemetry, and a model lock that no workspace or user setting can override. An organization that locks the model choice in the Copilot picker, for example between Claude Fable 5, Opus 4.8, Sonnet 5, Gemini 3.1 Pro, or a GPT-5.x model, does it here, and the lock holds all the way down.

The inheritance is one-directional by design. The organization sets floors, the workspace sets the team contract, the user personalizes, and the session (pinned files, open tabs, the current selection) is the innermost and most ephemeral layer. The outer scope cannot be overridden from inside, and inner settings never leak outward, so a personal preference is never committed to the repository by accident. The operational principle follows from the shape: enforce at the outer scope, do not police per user. Lock policy at the organization or workspace level, put useInstructionFiles and instructionsFilesLocations in .vscode/settings.json so the team contract is real rather than aspirational, and let each developer add ergonomic preferences that cannot undermine the contract.

Assembly order: the nine steps that tell you why a rule was ignored

When a developer submits a prompt, Copilot builds the effective context in a deterministic nine-step sequence. Steps one to three resolve settings in precedence order, default then user then workspace. Step four loads hot memory, .github/copilot-instructions.md, unconditionally. Step five matches scoped instructions, evaluating every applyTo glob against the open files and merging the sets that match. Step six attaches the active authoring primitive, whichever agent, chat mode, skill, or prompt file is in play. Step seven runs the onPromptSubmit hooks, which may deny, mutate, or annotate the request. Step eight discovers MCP tools, listing the connected servers and exposing their catalog. Step nine sends the assembled system prompt, tool catalog, and user message to the model.

The value of knowing the order is diagnostic. When Copilot appears to ignore a rule, the model is almost never the cause. The bug is in step three (the setting resolved at the wrong scope), step five (the applyTo glob does not match the file that is actually open), or step seven (a hook silently mutated or blocked the instruction). Walk the

nine steps from the top before arguing with the model. The same determinism that makes the sequence debuggable also makes it auditable: the same repository at the same commit assembles the same context every time, so a rule that worked yesterday and not today points to a change in one of nine named places, not to a mood.

> DEFINITION
>
> Runtime governance in one line: hooks decide what the agent may do, MCP decides where it gets fresh data, plugins decide how the standard travels, and settings decide who may change any of it. When a rule is ignored, walk the nine assembly steps top down. The answer is almost always step three, five, or seven, not the model.

References

Primary sources for the runtime surfaces: the security gap that justifies hooks, the MCP standard and its threat model, the plugin registry, settings precedence, and the inheritance evidence.

- Chatlatanagulchai et al., An Empirical Study of Agent Context Files, the measured gap: explicit security guidance appears in only 14.5% of 2,303 real context files, the 85.5% that hooks back up.
- Model Context Protocol Specification, the open standard Copilot uses to reach external systems at runtime.
- MCP Security, Threat Model and Mitigations, the basis for wrapping any destructive MCP action in a preToolUse hook.
- Awesome GitHub Copilot, the community registry that distributes plugins, agents, chat modes, and prompts.
- VS Code, Settings Precedence, the reference for how default, user, workspace, and folder scopes resolve.
- Vasilopoulos, Codified Context Infrastructure, the 108,000-line case study that held a single workspace constitution across 283 sessions at a 24.2% knowledge-to-code ratio.

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 MONDAY MORNING

From zero to baseline in one week.

The ten-layer surface is worthless if nobody touches it. This chapter is the smallest sequence of commits that moves a team from zero to a real configuration baseline in one week. One commit per day, each one closing a specific gap the research documents. By Friday the workspace, authoring, and runtime layers all have a first tenant, and the behavior of GitHub Copilot in the repository is auditable instead of tribal.

The day-by-day commit sequence

Monday: create the .github/copilot-instructions.md with four sections: Stack, Conventions, Security, and Performance. Keep the file under about 1,000 tokens. The Security and Performance sections matter most, because the study of 2,303 real context files found security guidance in only 14.5% of them and performance rules in another 14.5%. Writing those two sections on day one closes the gap that produces productive but unsafe code.

Tuesday: add the .vscode/settings.json with useInstructionFiles turned on and instructionsFilesLocations pointing at .github/instructions. This is the step that makes the team contract real instead of aspirational. Commit the workspace settings so every contributor loads the same behavior. Leave personal preferences in user settings, which never reach the repository.

Wednesday to Friday: scope, guardrails, reuse

Wednesday: write one scoped instruction file under .github/instructions/ with an applyTo glob aimed at the riskiest directory in the repository, migrations, payments, or an admin API surface. This is the move from a single monolithic file, which stops scaling past roughly 3,000 tokens, to the three-tier model. Thursday: write one hook under .github/hooks/ that denies destructive shell commands before they run. Five lines of JSON are the deterministic backstop for the code paths the model will not always protect on its own. Friday: extract one recurring multi-step workflow into a

skill under .github/skills/ with a strong description field, so agent mode finds it by description match instead of a teammate re-explaining it every sprint.

The following week, package the five artifacts into a plugin and install it across the top three repositories. A plugin bundles agents, chat modes, skills, prompt files, hooks, and MCP declarations behind one manifest, pinned to a version. Changes then flow through one pull request on the plugin instead of forty copy-and-paste edits across forty repositories.

> DEFINITION
>
> One commit per day: Monday copilot-instructions.md, Tuesday settings.json, Wednesday one scoped instruction, Thursday one hook, Friday one skill, next week one plugin. Each commit gives a layer its first tenant and closes a gap the research names.

Honest expectations: what the hierarchy will and will not do

The baseline will not turn a junior team into a senior team. GitHub Copilot amplifies what is written down. If a convention lives only in someone's head, the model cannot follow it, and the hierarchy changes nothing until the convention is codified. The layers are a place to put knowledge, not a substitute for having it.

What the baseline will do is make behavior reproducible. The Zoominfo evaluation of a rollout across more than 400 developers reported a sustained suggestion-acceptance rate near 30% once workspace-level settings were standardized, instead of results that varied from one machine to the next. A reproducible 30% baseline is a number a team can defend in a code review and improve on deliberately. The single-file approach cannot offer that, because it drifts silently the moment the file grows past the model's budget.

> DEFINITION
>
> What it will not do: make undocumented conventions appear. What it will do: give the team a reproducible acceptance baseline near 30% (Zoominfo 2025) instead of it-works-on-my-machine stories.

Governance, not prompt engineering: making each layer auditable and owned

The hierarchy is not a feature checklist. It is a governance model for AI in software engineering: who declares context, who authors instructions, and what runs at every prompt. Each of the ten layers has an owner. Workspace instructions and scoped instructions belong to the team as a contract, committed to the repository and reviewed like any other code. Agents, chat modes, skills, and prompt files belong to whoever owns the task they encode. Hooks, MCP connections, plugins, and settings belong to the platform team, because they configure the runtime around the agent rather than any single feature.

Auditability follows from that ownership. Every rule lives in a versioned file, reviewed in a pull request, attributed to a commit. When Copilot ignores a rule, the fix is to walk the assembly order top-down, checking settings scope, then the applyTo glob, then any hook that mutates the request, instead of arguing with the model. Enforce policy at the outer scope and do not police per user: lock the organization and workspace floors, and let inner scopes add ergonomic preferences without breaking the contract. Teams that treat the hierarchy as governance ship. Teams that treat it as prompt engineering plateau.

References and further reading

Primary sources for the day-by-day baseline, the empirical gaps it closes, and the acceptance numbers it targets.

- Agent READMEs: An Empirical Study of Context Files for Agentic Coding (Chatlatanagulchai et al., 2025), the source for the 14.5% security and

performance gap that Monday's commit closes, across 2,303 real context files.

- Codified Context: Infrastructure for AI Agents in a Complex Codebase (Vasilopoulos, 2026), the 108,000-line, 283-session case study behind the three-tier model and the single-file collapse past 3,000 tokens.
- A Four-Phase Evaluation of GitHub Copilot Across 400+ Developers (Zoominfo, 2025), the source for the reproducible acceptance rate near 30% once workspace settings are standardized.
- Customize AI in Visual Studio Code, the reference for instruction files, scoped instructions, and the settings that make the team contract real.
- GitHub Copilot Documentation, the platform reference for agent mode, copilot-instructions.md, and the runtime layers this chapter configures.
- Model Context Protocol Specification, the open standard for the MCP connections a plugin can bundle alongside hooks and skills.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps