ENTERPRISE API AND TOOLING GOVERNANCE · PLAYBOOK

One control plane governs every API, extension, and agent that reaches the workstation.

Enterprise development environments accumulate security, compliance, and productivity risk faster than any single team can govern it. This playbook builds one control plane across three hubs, Azure API Center for APIs, a VS Code Private Marketplace for extensions, and Copilot governance for MCP servers and agents, so nothing reaches a developer workstation unvetted. The artifacts change. The contract does not.

AUTHOR

Paula Silva

ROLE

AI-Native Software Engineer

CONTACT

in/paulanunes

READING TIME

~ 45 minutes, end to end

3	6	6
GOVERNANCE HUBS	CHAPTERS	PLATFORM PERSONAS

30-50%

SHADOW APIS FOUND

CHAPTERS

Chapter 00

Sprawl without a control plane

Why enterprise environments accumulate security, compliance, and productivity risk at once, how shadow APIs and unvetted extensions reach the workstation, the one-control-plane thesis, and the six platform personas this guide is for.

Concept →

Chapter 01

Three hubs, one workstation

The integrated control plane and the shared policy layer, the role of each hub, the concepts that separate API Center from API Management, MCP, and AllowedExtensions, and how the hubs converge on VS Code with GitHub Copilot.

Architecture →

Chapter 02

Azure API Center as the source of truth

The line between API Center and API Management, registration by portal, CLI, and CI/CD, required metadata for compliance, Spectral linting at the gate, shadow API discovery with Dev Proxy, and the VS Code workflow.

API Inventory →

Chapter 03

The Private Marketplace and extension policy

A stateless registry for internal, rehosted public, and air-gapped extensions, AllowedExtensions enforced through Group Policy and Intune, the bootstrap install, and the footprint drop from roughly 150 to about 12 vetted.

Marketplace →

Chapter 04

Governing MCP, agents, and Copilot artifacts

Approval and distribution of MCP servers, custom agents, and prompt files, model choice from the GitHub Copilot picker, agent mode confirmations, and the security architecture covering supply chain, prompt injection, and audit.

AI Tooling →

Chapter 05

From Day 0 to Day 2, and governance

The three deployment scenarios, the Day 0 to Day 2 sequence that stands the platform up and keeps it running, the ownership matrix with approval paths and review cadence, and the build-once inheritance principle.

Operations →

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 SPRAWL

Sprawl without a control plane: three risks, one thesis, six personas.

Enterprise development environments accumulate risk faster than any single team can govern it. APIs multiply without a central inventory. Extensions, MCP servers, and Copilot agents land on developer workstations without review. The three risks that follow, security, compliance, and productivity, are not independent, and most organizations have no single surface from which to address all three at once. This chapter names the sprawl, states the thesis that fixes it, and identifies the six platform personas this guide is written for.

Three converging risks that share no control surface

Enterprise environments accumulate risk in three compounding categories, and most organizations have no single surface from which to address all three at once. Security is the first pressure point. Extensions and MCP servers installed ad hoc, outside any approval process, can exfiltrate code or credentials. The IDE becomes the soft target. It sits inside the perimeter, has deep access to the filesystem and the network, and is rarely covered by the same threat model as production infrastructure. Compliance is the second. Sensitive data reaches AI model APIs without authorization, and audit teams cannot prove what left the environment, when it left, or to whom it was sent. As AI-assisted development becomes standard, and as those model calls are billed through GitHub AI Credits, the compliance gap widens faster than manual governance can close it.

Productivity is the third risk, and often the least intuitive. When developers cannot discover what APIs already exist, they rebuild. They reach for deprecated endpoints. They burn hours configuring tooling that lives somewhere else in the organization already. The cost of duplication compounds quietly. These three risks are not independent. Unvetted tooling drives shadow API usage. Shadow API usage

produces data-flow gaps. Data-flow gaps create compliance exposure. A control plane that addresses only one category leaves the other two open, which is why the fix has to be structural rather than a set of point solutions bolted onto the environment after the fact.

Shadow APIs, unvetted extensions, MCP servers, and agents

The sprawl has a specific shape. APIs proliferate without a central inventory, and shadow APIs get built because teams cannot see what already exists. Undocumented endpoints run in production, called by services that no register knows about. In practice, a first discovery pass against a live gateway commonly surfaces thirty to fifty percent of production APIs that appear in no register at all, and most teams underestimate that share by half until they run it. That gap is the sharpest single measure of how far inventory has drifted from reality, and it is the debt every later governance decision inherits.

The same pattern repeats above the API layer. VS Code extensions get installed without vetting, and a single developer base commonly spreads across about 150 distinct extensions before any policy is in place. AI agents and Copilot plugins pull data into external services without oversight. MCP servers are configured ad hoc, one developer at a time, each one a new integration point with its own trust assumptions. Custom agents defined in `.github/agents/` files and instructions in `.github/copilot-instructions.md` shape how GitHub Copilot behaves in agent mode, so an unreviewed instruction file is a governance gap, not just a convenience. Every one of these artifact types is a channel into the workstation, and every channel is ungoverned by default.

The thesis: nothing reaches a developer workstation unvetted

The fix is one rule, and it fits in a sentence: nothing reaches a developer workstation without passing through a governed catalog. Three hubs enforce it. Azure API Center is the authoritative inventory for every API in the organization, with Spectral linting at registration time and Dev Proxy discovery for the shadow APIs already in flight. The

VS Code Private Marketplace is the curated catalog for every extension, internal and rehosted public alike. Copilot governance covers every MCP server, custom agent, and prompt file. Three hubs, one control plane.

The three hubs converge on a single developer workstation, VS Code with GitHub Copilot, protected by a shared policy layer that enforces AllowedExtensions, RBAC, identity, and audit logging. GitHub Copilot in agent mode reads instructions from `.github/copilot-instructions.md` and calls tools through MCP, so the same policy layer that governs extensions and APIs also governs how the agent behaves. The catalog is not a gate that slows developers down. It is the fast path, because the vetted artifact is the one already installed and already trusted on day one.

One control plane, not three tools

The point is not three separate tools bolted together. Each artifact category, APIs, extensions, and AI tooling, gets its own governance contract, but every contract is enforced through the same control plane. Build it once and the next artifact category, whatever arrives after MCP servers and Copilot agents, inherits the same enforcement surface automatically. When the planes are wired together, every developer-facing surface produces a logged event, and that audit trail is the difference between a regulated-industry pilot that scales and one that gets revoked at the next external audit.

> DEFINITION
>
> Nothing reaches a developer workstation without passing through a governed catalog. Azure API Center governs every API, the VS Code Private Marketplace governs every extension, and Copilot governance covers every MCP server, agent, and prompt file. The artifacts change. The contract does not.

Who this guide is for: six platform personas

This guide is written for the platform teams that own the control plane, not for the developers who consume it. Six personas recur across enterprise deployments. The platform engineer builds and runs the three hubs. The IT administrator enforces extension policy through Group Policy on Windows and Intune on macOS and

managed devices. The security engineer owns the threat model, the supply-chain controls, and the audit posture. The developer enablement lead makes the governed path the easy path, so the curated catalog is also the fastest one. The enterprise architect keeps the three hubs coherent with the wider platform. The API program manager treats the inventory as a product, with owners, classifications, and a lifecycle.

Each persona reads the guide differently, and the chapters that follow are structured for that. Platform engineers usually start with the architecture and the operational sequence. Security engineers go straight to the threat model and the regulated-industry scenario. Enablement leads care most about the developer onboarding path. The guide builds the control plane one hub at a time, then wires the hubs together, so any persona can enter at the layer they own and still see how it connects to the rest of the platform.

References

Primary sources for the inventory, marketplace, and AI-tooling governance patterns in this chapter.

- Azure API Center, centralized inventory and governance, the authoritative register for every API, version, environment, and deployment.
- Analyze APIs in your API center, Spectral linting and analysis, how design-standard rules block non-compliant APIs at registration time.
- Find shadow APIs with Dev Proxy and API Center, surfacing undocumented APIs that developers already call before they become production debt.
- Link an Azure API Center to Azure API Management, how gateway truth and inventory truth converge into one record.
- Enterprise extension management in Visual Studio Code, AllowedExtensions policy through Group Policy and Intune.
- Use MCP servers in Visual Studio Code, the install and distribution model for governed MCP servers.

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 ARCHITECTURE

Three hubs, one workstation: the integrated control plane.

Three hubs converge on one developer workstation. Azure API Center governs the APIs, a private marketplace governs the VS Code extensions, and Copilot governance governs the AI artifacts. A shared policy layer sits across all three, and nothing reaches the workstation without passing through it. This chapter maps the architecture: the control plane, the role of each hub, the concepts that separate them, and how they meet inside VS Code with GitHub Copilot.

The integrated control plane and the shared policy layer

The architecture starts from one workstation: VS Code (Desktop) running GitHub Copilot. Everything else exists to govern what reaches that surface. Enterprise development accumulates risk in three places at once: unvetted extensions and MCP servers that can exfiltrate code, sensitive data flowing to model APIs without authorization, and duplicated work when a developer cannot find what already exists. A control plane that closes only one of those loops leaves the other two open. The integrated architecture closes all three by governing three artifact categories through one enforcement surface.

The shared policy layer is that surface. It enforces AllowedExtensions, RBAC, identity, and audit logging across every hub. AllowedExtensions is a VS Code enterprise policy that controls which extensions can install and run. RBAC scopes who can read and register artifacts. Identity propagation attaches a real user to every action, so audit logs stay attributable. The rule is simple and strict: nothing reaches the developer workstation unvetted. The policy does not change when the artifact type changes. APIs, extensions, and AI tools each get their own governance contract, and the same control plane enforces all of them.

DEFINITION

The anchor rule: nothing reaches the developer workstation unvetted. Each artifact category gets its own governance contract, but every contract is enforced through the same control plane. The artifacts change. The contract does not.

Hub roles: Azure API Center, Private Marketplace, Copilot governance

Each hub owns one artifact category. Hub 01 is Azure API Center, the authoritative inventory for every API in the organization. Every version, every definition, every environment and deployment is registered there. CLI and CI/CD pipelines feed registration automatically, so the catalog reflects what teams actually ship. Custom metadata fields capture ownership and classification, which makes the catalog both a discovery surface and a governance record. Spectral linting runs at registration time and blocks APIs that violate organizational design standards before they ship. Dev Proxy discovery surfaces undocumented APIs that developers are already calling, so shadow APIs enter the catalog before they harden into production debt.

Marketplace and Copilot governance

Hub 02 is the VS Code Private Marketplace, a curated extension catalog under enterprise control. The design is intentionally stateless: a container with object storage backing, deployable in any region, serving internal extensions and rehosted public ones. Air-gap deployments keep developers productive with no network egress to the public marketplace. Policy enforcement uses Group Policy on Windows and Intune on macOS and managed devices. A bootstrap install ensures every developer opens a curated VS Code on day one, not a raw, unmanaged installation.

Hub 03 is Copilot governance, which applies the same principle to AI artifacts. MCP servers are registered in `.vscode/mcp.json` and distributed through the marketplace, each with a documented trust model. Copilot extensions are reviewed for data flow and allowlisted at the GitHub Enterprise organization level. Custom agents live in `.github/agents/` files, version-controlled alongside the code they

govern and reviewed in the same pull request pipeline. Prompt files and skills are curated in a shared repository with an approval workflow before they reach the developer fleet.

Key concepts: API Center vs API Management, MCP, AllowedExtensions

Three concepts carry most of the weight, and the first is the difference between Azure API Center and Azure API Management. They are separate services with separate jobs. API Management is the runtime gateway: it fronts live traffic, applies policies, and routes requests. API Center is the inventory and governance record: it catalogs every API regardless of where it runs, including APIs that never pass through the gateway. The two connect, they do not overlap. API Management feeds API Center upstream automatically, so gateway truth and inventory truth converge into one record. A team can register an API in API Center before any gateway exists, then link it to API Management once it goes live.

MCP and AllowedExtensions

The second concept is the Model Context Protocol. MCP is an open standard that lets an agent call external tools and data sources through a uniform interface. In this architecture, MCP servers are the unit of AI capability: a server exposes a set of tools, GitHub Copilot agent mode queries it on demand, and the trust model for each server is documented before it ships. Registering servers in `.vscode/mcp.json` and distributing them through the marketplace turns an ad hoc integration into a governed artifact.

The third concept is AllowedExtensions, the VS Code enterprise policy that defines which extensions a managed installation may load. It is the mechanism behind the private marketplace: the policy points the editor at the internal catalog and blocks anything outside it. Enforced through Group Policy or Intune, AllowedExtensions is what makes a curated marketplace binding rather than advisory.

How the hubs converge on VS Code with GitHub Copilot

The three hubs resolve to the same place: the editor the developer works in. API Center is reachable from inside VS Code, so a developer discovers and registers APIs without leaving the IDE. The private marketplace is where extensions install from. Copilot governance decides which agents, MCP servers, and instructions are active in the session. The workstation is where governance becomes visible to the person doing the work, and where it either helps or gets in the way.

GitHub Copilot is the surface that ties them together. On session start it reads `.github/copilot-instructions.md`, so repository conventions travel with the code. In agent mode it queries the MCP servers that Copilot governance approved. The model picker lets a developer choose the model for the task, Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, or a GPT-5.x model, while the organization governs which models and agents are available. Usage bills against GitHub AI Credits, where one credit equals one cent, so agent activity carries a cost the platform can measure. The operational payoff shows up on Day 2: linting blocks non-compliant APIs at the gate, Dev Proxy surfaces shadow APIs before production, and audit logs give compliance teams continuous visibility. Build the enforcement surface once, and every new artifact category inherits it.

References

Primary sources for the three-hub architecture, the shared policy layer, and the key concepts.

- Azure API Center overview, the authoritative inventory hub for every API.
- Azure API Center key concepts, the distinction between API Center and API Management.
- Link an Azure API Center to Azure API Management, how the gateway feeds the inventory upstream.
- Enterprise extension management in Visual Studio Code, AllowedExtensions and managed installations.

- Use MCP servers in Visual Studio Code, registering MCP servers for GitHub Copilot agent mode.
- Model Context Protocol Specification, the open standard behind the AI capability hub.
- GitHub Copilot custom instructions, how .github/copilot-instructions.md reaches the session.
- Using GitHub Copilot with an AI model of your choice, the model picker on the workstation.

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 API INVENTORY

Azure API Center as the source of truth: one register for every API.

A governance platform is only as honest as its inventory. If a team cannot say which APIs exist, which version is canonical, and which ones touch sensitive data, every control above that gap is guesswork. This chapter puts Azure API Center at the base of the stack: how it differs from API Management, how APIs get registered and tagged, how linting and shadow API discovery keep the register honest, and how developers reach all of it from inside VS Code.

Drawing the line between API Center and API Management

Azure API Center and Azure API Management solve different problems, and the platform breaks when a team treats them as one product. Azure API Center is the single register for every API in the organization. It records what APIs exist, which version is canonical, whether a definition conforms to the style guide, and who owns each entry. It holds metadata, versions, OpenAPI definitions, and deployment environments across the full lifecycle, from early design to deprecation. Azure API Management is the runtime layer. It is the gateway that publishes, secures, rate-limits, and monitors live traffic for the subset of APIs it manages.

The relationship runs one way and then back. API Center knows about all APIs. API Management governs traffic for the ones it fronts. The two connect through synchronization: link an Azure API Management instance and its full catalog flows into API Center automatically, so gateway truth and inventory truth converge into one record without manual re-entry. A practical way to explain it to stakeholders is that API Management is the traffic officer at the intersection, and API Center is the planning office that knows every road exists. The line matters for cost too. When you link an API Management instance at the Standard, Standard v2, Premium, or Premium

v2 tier, the Standard API Center plan comes at no additional cost for as long as one qualifying instance stays linked.

Registration by portal, CLI, and CI/CD, plus metadata for compliance

A register only earns trust when every API reaches it the same way every time. API Center supports three registration paths that map to how teams actually work. The Azure portal handles manual registration: add the API, set its type (REST, GraphQL, gRPC, SOAP, Webhook), pick a lifecycle stage, and upload the OpenAPI definition. The Azure CLI scripts the same steps through az apic api create, az apic api version create, and az apic api definition import-specification, which makes registration repeatable and reviewable. CI/CD closes the gap. A GitHub Actions workflow that triggers on changes to openapi files registers a new version on every merge to main, so the catalog never drifts behind the code.

Metadata turns a list into a governed inventory

Registration records that an API exists. Custom metadata records what compliance and security need to know about it. Define the properties once with az apic metadata create and require them on every entry: a team owner, a boolean for whether the API processes PII, a business-domain enum. Mark the properties required and the catalog stops accepting orphan APIs with no owner and no data classification. That is what separates a discovery surface from a governance record. When an auditor asks which production APIs touch PII, the answer is a metadata filter, not a survey.

Spectral linting at the gate, shadow API discovery with Dev Proxy

Two controls keep the register honest: one blocks bad specs on the way in, the other finds the APIs that never asked permission. API Center runs built-in linting powered by the open-source Spectral engine. Upload your organization's style guide as a Spectral ruleset, and every OpenAPI definition is validated against it at registration time. Definitions that break naming, versioning, or security rules are flagged before

they ship, not caught in review three weeks later. The rules are the same everywhere, so API design stops being a matter of who reviewed the pull request.

Shadow APIs are the calls an application already makes that appear in no catalog. They are technical debt and compliance exposure at the same time. Microsoft Dev Proxy intercepts an application's HTTP traffic, records every call, and compares the observed endpoints against the API Center inventory. The report lists every API that was called but never registered, with suggested registration steps. In practice, teams that run discovery for the first time find that 30 to 50 percent of the APIs in production were missing from any register. Bringing those into API Center before they accumulate is the difference between an inventory that reflects reality and one that documents intentions.

The VS Code workflow, from browse to Copilot grounding

The register is only useful if developers reach it without leaving the editor. The Azure API Center extension for VS Code brings the full inventory into the Activity Bar. Developers browse instances by subscription, open any OpenAPI definition with syntax highlighting, and filter by metadata with a query such as domain:payments pii:true lifecycle:production. They register new APIs by right-clicking a spec, which keeps registration a design-time act rather than a post-deployment cleanup. They generate a typed client in TypeScript, Python, C#, Java, or Go from any registered definition, so consumer code matches the current version instead of a hand-written guess. The same Spectral ruleset runs inline as they type, surfacing violations in the Problems panel before the spec is ever uploaded.

This is where GitHub Copilot earns its place. With the @azure Copilot extension enabled, agent mode grounds its answers in the live API Center inventory rather than guessing. Ask which APIs handle customer authentication, or which ones process PII, and Copilot returns contextual answers from the register, not from training data. Point it at a controller and it drafts an OpenAPI spec that conforms to the registered style, ready to register through the extension. Expose the inventory through the Model Context Protocol and any agent in the picker, whether Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, or GPT-5.x, works from the same authoritative context. Record the grounding rules in .github/copilot-instructions.md

so every developer's agent starts from the same source of truth. GitHub Copilot billing runs on GitHub AI Credits, where one credit equals one US cent, so grounded queries that avoid re-deriving the catalog on every session are cheaper as well as more accurate.

> DEFINITION
>
> The rule is simple: an API is real when it is in API Center, not when it is deployed. Registration is a design-time act, metadata is required and not optional, linting runs at the gate, and Dev Proxy finds what slipped past. Everything the developer and the agent do flows through one register.

References

Primary Microsoft Learn sources for API Center as the inventory hub, its registration and governance controls, and the VS Code workflow.

- Microsoft Learn, What is Azure API Center, the centralized inventory and governance hub for every API in the organization.
- Microsoft Learn, Synchronize APIs from Azure API Management, how gateway truth flows into the inventory automatically.
- Microsoft Learn, Register APIs with GitHub Actions, CI/CD registration that keeps the catalog current on every merge.
- Microsoft Learn, Enable API analysis and linting, Spectral rules validating every definition at registration time.
- Microsoft Learn, Discover shadow APIs with Dev Proxy, finding the APIs that run in production but appear in no register.
- Microsoft Learn, Build and register APIs with the VS Code extension, the browse, register, and generate-client workflow inside the editor.

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 PART II, CURATED TOOLING

The Private Marketplace and extension policy: one channel to the editor.

A curated marketplace is the second control plane. The first governs APIs. This one governs what runs inside the editor. Every extension, every MCP server, and every GitHub Copilot artifact reaches the developer through one channel the platform team owns. This chapter describes that channel: a stateless registry, the AllowedExtensions policy that enforces it, the bootstrap install that seeds it, and the footprint reduction it produces.

A stateless registry: internal, rehosted public, air-gapped

The Private Marketplace is a curated extension catalog that the enterprise controls. Its architecture is stateless by design. The registry runs as a container backed by object storage, and it can be deployed in any region the organization needs. It serves two kinds of extension. The first is internal: extensions the organization builds for its own workflows, published straight to the catalog. The second is rehosted public: extensions from the public VS Code marketplace, downloaded, vetted, and republished on internal storage so the editor never has to reach the public gallery.

The stateless design is what makes the air-gapped scenario work. In a restricted network with no public egress, the marketplace holds every extension a developer needs on internal storage, and the editor points at the internal endpoint instead of the public one. The same registry serves the cloud-connected enterprise and the regulated bank. Only the network boundary changes. This is Hub 02 in the three-hub architecture, and it sits under the same policy layer as Azure API Center and Copilot governance. Nothing installs in the editor without passing through it.

AllowedExtensions enforced through Group Policy and Intune

A catalog is a suggestion until a policy makes it the only option. AllowedExtensions is that policy. It is a VS Code enterprise setting that names the extensions a managed editor is permitted to install, and it blocks everything else. On Windows the platform team pushes it through Group Policy. On macOS and other managed devices it goes through Intune. The mechanism differs by operating system. The result is identical: the developer editor can install what the catalog allows and nothing more.

The policy is not a lock that traps people. An extension outside the allowlist is still installable on request, and the request leaves a traceable approval. That distinction matters for the compliance posture. In a regulated deployment, every extension on every workstation maps back to an approval record, and an auditor can read the list without asking the platform team to reconstruct it. AllowedExtensions also governs the AI surface. A GitHub Copilot extension, or any extension that carries an MCP server, is subject to the same gate as a syntax highlighter. The curated catalog is where the approved AI tooling lives, not a separate channel.

Bootstrap install and the recommended enterprise catalog

The policy decides what is allowed. The bootstrap install decides what a developer starts with. A bootstrap install is a scripted first-run setup that hands every developer a curated VS Code on day one, not a raw editor they configure by hand. It points the editor at the Private Marketplace, applies the AllowedExtensions policy, and installs the recommended catalog. A new hire opens the editor and finds the linters, the language support, and the internal extensions the team already agreed on.

The recommended catalog is where the GitHub Copilot lens sharpens. GitHub Copilot arrives in that install already governed. Its model picker is pre-populated with the models the organization has approved, drawn from the same set every Copilot user sees: Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, and the GPT-5.x family. Its behavior is shaped by a checked-in .github/copilot-instructions.md, and the MCP servers it can reach are the ones registered in .vscode/mcp.json and distributed through the same marketplace. Agent-mode work

bills against GitHub AI Credits, where one credit is one cent under the model in effect since June 2026, and a governed install is what makes that spend attributable to a team and a repository rather than a mystery line on an invoice.

Reference footprint: 150 extensions down to roughly 12 vetted

The footprint number is the one to defend in a review. A typical developer base accumulates around 150 distinct extensions across all workstations, installed ad hoc over time with no vetting. When AllowedExtensions enforces the curated catalog, that number drops to roughly 12 vetted extensions plus the internal ones the organization publishes. This is a reference reduction observed in practice, not a target set by policy. The vetting is what shrinks the surface, and the policy is what holds it there.

A smaller footprint is a smaller attack surface. Every extension installed in the editor runs with the developer filesystem and network access, which is why an unvetted extension is a plausible path for code or credential exfiltration. Cutting 150 down to 12 removes most of that exposure without removing capability, because the 12 are the ones developers actually needed. The blocked extensions remain available on request, so a developer with a real need files a request and gets a traceable approval. The default is curated. The exception is documented. That is the whole design principle of the marketplace, and it is the same contract Azure API Center applies to APIs and Copilot governance applies to agents.

> DEFINITION
>
> The extension footprint in one line: from roughly 150 ad hoc extensions across a developer base to about 12 vetted ones plus internal extensions, enforced by AllowedExtensions and reversible by a traceable approval. Curated by default, documented by exception, attributable end to end.

References

Primary sources for the Private Marketplace architecture, the extension policy mechanics, and the AI tooling distribution described in this chapter.

- Visual Studio Code, Private Marketplace announcement, the reference for the stateless registry that serves internal and rehosted public extensions.
- VS Code Private Marketplace, setup and rehosting readme, deployment and public-extension rehosting steps for air-gapped and connected networks.
- Visual Studio Code, Enterprise extension management, the AllowedExtensions setting that turns the curated catalog into the only installable set.
- Visual Studio Code, Enterprise policies, how the policy is pushed through Group Policy on Windows and Intune on managed devices.
- Use MCP servers in Visual Studio Code, how MCP servers ride the same distribution channel as the curated extensions.

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 AI TOOLING

Governing MCP, agents, and Copilot artifacts: the AI tooling contract.

MCP servers, custom agents, prompt files, and Copilot instructions are code that runs inside the developer workstation. They read the filesystem, call internal APIs, and act on behalf of the developer. Treat them as tooling artifacts with the same approval, distribution, and audit contract as any extension or any API. This chapter defines that contract for the AI tooling layer, then closes with the security architecture that holds it together.

MCP servers: approval, distribution, and credentials

The Model Context Protocol connects a GitHub Copilot agent to external tools and data during a conversation. Without it, Copilot reads only the code in the editor. With it, an agent in agent mode can query Jira, read a database, call an internal API, or search Confluence. MCP uses a client-server model. VS Code is the client. Each system exposes an MCP server that declares its capabilities as tools, resources, and prompts. The agent decides which tools to invoke from the request it is given. Servers run locally over stdio or remotely over HTTP. For organizational tools, shared remote servers are the preferred model: one configuration, one place to manage credentials.

An MCP server is a developer tooling artifact. Govern it with the same rigor as an extension or an API. A server that connects to an internal database is as sensitive as the database credentials it holds. Every server passes a security gate before it reaches the approved catalog: source code or vendor report reviewed, permissions scoped to the minimum, read-only where possible, remote servers on HTTPS with authentication, and no transmission of request content to unauthorized destinations. Only then is the server added to the organization catalog and cleared for distribution.

Distribution and credential handling

Distribute approved servers where developers already work. A `.vscode/mcp.json` file committed to the repository gives every clone the same tools with no manual setup. A base configuration pushed through Intune or JAMF pre-loads organization-approved servers on every managed workstation, and developers add repository-specific servers on top. Credentials never live in `settings.json` or `mcp.json` as literal strings. Reference them as environment variables and source them from a secrets manager such as Azure Key Vault at session start. The config declares which secret it needs; the secret itself stays in the vault. Credential exposure and prompt injection are the two named risks of the MCP trust model, and both are addressed here.

Custom agents, prompt files, and instructions

GitHub Copilot has several artifact types, and each extends it differently. A custom agent is a Markdown file in `.github/agents/`, selected from the agent picker in Copilot Chat. It encodes team domain knowledge, conventions, and scoped tool access directly in the repository. A prompt file is a reusable `.prompt.md` template, invoked with a slash command for recurring complex tasks. Copilot instructions live in `.github/copilot-instructions.md` and apply automatically to every Chat session in the repository. Because all three are files, they are version-controlled next to the code they govern and reviewed in the same pull request pipeline.

Place an agent in the `.github/agents/` directory of the organization `.github` repository and it becomes available across every repository with no per-repository setup. The same holds for org-wide prompt files and instructions. This reach is exactly why the artifacts need a review contract. Custom agents are reviewed by developer enablement before merge. Prompt files that call terminal commands require a security review. A central inventory of agents, prompts, and instructions prevents uncontrolled proliferation, which is the AI tooling version of extension sprawl.

Instructions as the grounding surface

The instructions file is where the repository tells the agent how the code actually works: the API standards, the technology stack, the naming and error conventions. It

is also where safety guardrails live. Instruct the agent to treat text returned from external tools as untrusted data, to show the full command before it modifies infrastructure, and to never push to a remote without explicit confirmation. Instructions are grounding, not decoration. An agent that reads them produces code that belongs in the repository, instead of code that merely compiles.

Model choice and agent mode confirmations

GitHub Copilot exposes a model picker. The developer chooses the model per task from the available set: Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, and the GPT-5.x family. A custom agent can pin a default model in its definition, and the developer can override it at the picker. Match the model to the work. A large refactor across many files rewards a stronger model; a short, well-scoped edit runs fine on a faster one. Billing is metered in GitHub AI Credits, where one credit equals one cent, in effect since June 1, 2026. Model choice is therefore a cost decision as much as a quality one, and it belongs in the same governance conversation as everything else on this layer.

Agent mode is where the model stops suggesting and starts acting: it edits files, runs terminal commands, and invokes MCP tools. That autonomy needs a brake. Configure VS Code to require developer confirmation before sensitive actions: terminal commands, file writes, and MCP invocations. Set these confirmations to always for write operations. The confirmation prompt is the point where a human reads the plan and approves or rejects each step. It is the difference between an agent that assists and an agent that surprises.

> DEFINITION
>
> The rule for agent mode: confirmation is mandatory for every write. Terminal commands, file writes, and MCP invocations each require an explicit human approval. The developer sees exactly what the agent will do before it does it.

Security architecture: threat model, supply chain, audit

The AI tooling layer adds attack surfaces, and each one needs a named mitigation. A malicious extension can exfiltrate code: the AllowedExtensions policy and a private marketplace with security review block it. An MCP tool response can carry a prompt injection: confirmation for write actions and agent guardrails contain it. A third-party Copilot extension can receive code context: an organization-level allow list and a data-flow review govern it. A custom agent can take a destructive action: confirmation policies and guardrail instructions constrain it. Supply chain security is defense in depth. The policy blocks unapproved extensions, the private marketplace rehosts only reviewed packages, and a regular audit compares what is installed against the approved list.

Prompt injection is the signature risk of the agentic layer. It happens when data returned by a tool, a Jira ticket body or a Confluence page, contains instructions that manipulate the agent. The mitigations are layered: require confirmation before every agent action, train developers to read the full plan before approving, and add guardrail instructions that mark all external tool output as untrusted. Above the controls sits the audit trail. When the planes are wired together, every API call, every MCP invocation, and every agent action produces a logged event into Microsoft Purview or the SIEM of choice. The audit trail is the difference between a regulated pilot that scales and one that is revoked at the next external audit.

> DEFINITION
>
> Treat text returned from external tools as untrusted data. A Jira ticket, a Confluence page, or a database record can carry instructions aimed at the agent, and the agent must not follow them. Confirmation gates plus guardrail instructions in copilot-instructions.md are the defense.

References

Primary sources for MCP governance, custom agents, model choice, and the Copilot security posture.

- Model Context Protocol Specification, the standard that connects Copilot agents to external tools and data.
- Use MCP servers in Visual Studio Code, configuration and distribution through mcp.json.
- GitHub Copilot Custom Agents, the Markdown agent format in .github/agents/.
- GitHub Copilot Custom Instructions, the copilot-instructions.md grounding and guardrail surface.
- Using GitHub Copilot with an AI model of your choice, model choice from the Copilot picker.
- GitHub Copilot Trust Center, data privacy and security posture for enterprise Copilot.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 RUN AND GOVERN

From Day 0 to Day 2, and the governance framework.

An architecture is only as good as its rollout and its steady state. This chapter covers the three deployment scenarios the control plane supports, the Day 0 to Day 2 sequence that stands the platform up and keeps it running, and the governance framework that decides who owns what and how each artifact reaches a developer. The design goal is a platform that a new artifact category can join without a new governance project.

Three deployment scenarios: full integrated, air-gapped, regulated

The same architecture supports three deployment shapes, and the choice depends on network posture and regulatory load, not on a different design. In the full integrated scenario, the enterprise is cloud connected. API design happens in VS Code, registers to Azure API Center, syncs to Azure API Management, and is consumed back in the IDE. The VS Code Private Marketplace and GitHub Copilot both run online. Agent mode reads .github/copilot-instructions.md and calls MCP servers registered in .vscode/mcp.json. This is the baseline, and it exercises every hub end to end.

The air-gapped scenario removes public network egress. The marketplace rehosts public extensions on internal object storage, MCP servers run on internal endpoints, and GitHub Copilot Enterprise reaches the organization over a private link. Developers stay productive with no path to the public marketplace. The regulated scenario, for finance and public sector, adds depth to the compliance posture rather than changing the topology: audit log retention, classification labels on artifacts, signed binaries, and per-request identity propagation. The auditor is independent from the platform team. Every action is attributable and every artifact is traceable.

> DEFINITION
>
> One architecture, three postures. Full integrated exercises every hub online. Air-gapped removes egress and rehosts internally. Regulated adds retention, classification, signed artifacts, and per-request identity. The topology does not change; the controls tighten.

From Day 0 to Day 2: plan, deploy, operate

Day 0 is planning and platform standup. Azure API Center is provisioned as the authoritative inventory. The VS Code Private Marketplace is deployed as a stateless container with object storage behind it. GitHub Copilot governance policies are configured. Identity and RBAC are wired across all three hubs before a single developer is onboarded. Day 0 also decides the AI Credits budget for Copilot usage: since June 1, 2026, GitHub Copilot premium requests are billed in AI Credits, where one credit is one US cent, so the model picker choice between Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, and the GPT-5.x family is a cost decision the platform team plans up front.

Day 1 is the deployment sequence and fleet onboarding. Bootstrap installs distribute curated VS Code instances through Group Policy on Windows and Intune on managed macOS devices, so every developer opens a governed IDE on the first day rather than a raw installation. API registrations begin flowing through CI/CD into API Center. Azure API Management is linked so gateway truth and inventory truth converge into one record. MCP server trust models are documented, one per server, before the servers reach the fleet. Custom agents in .github/agents/ and instructions in .github/copilot-instructions.md ship version controlled alongside the code they govern.

Day 2: steady state and continuous onboarding

Day 2 is steady state operations, and it is where the platform earns its keep. Spectral linting runs at registration time and blocks non-compliant APIs at the gate. Dev Proxy surfaces shadow APIs that developers are already calling and brings them into the catalog before they become production debt. Audit logs give compliance teams

continuous visibility. Onboarding does not stop at Day 1: every new API, extension, MCP server, and Copilot agent enters through the same gates, so the fleet grows without the governance weakening.

> DEFINITION
>
> Day 0 stands up API Center, the Private Marketplace, Copilot policies, identity, and RBAC, and sets the AI Credits budget. Day 1 pushes curated VS Code through Group Policy and Intune, flows API registrations through CI/CD, and documents MCP trust models. Day 2 runs the gates: linting blocks non-compliant APIs, Dev Proxy catches shadow APIs, audit logs stay continuous.

The governance framework: ownership, approval paths, review cadence

Governance starts with an ownership matrix. Every artifact category has a named owner and a defined approver. APIs are owned by the producing team and approved against organizational design standards enforced by Spectral. Extensions are owned by the platform team and allowlisted through the marketplace. MCP servers carry a documented trust model and an owner accountable for the data they can reach. Copilot extensions are reviewed for data flow and allowlisted at the GitHub Enterprise organization level. Custom agents are owned by the team whose code they touch. RBAC in API Center and organization level controls in GitHub make the matrix enforceable rather than aspirational.

Each artifact category has its own approval path, and the path matches the artifact's risk. An API is approved when it passes linting at registration and carries the required ownership and classification metadata. An extension is approved when it is reviewed and published to the private marketplace. An MCP server is approved when its trust model is documented and its endpoint is vetted. A Copilot agent or an instruction file in .github/copilot-instructions.md is approved in the same pull request pipeline as the code it governs, reviewed by the same people, under the same branch protections. The approval path is code review, not a separate ticketing queue.

Review cadence

Review cadence keeps the controls from decaying. Linting rules and classification schemas are reviewed on a fixed cycle as standards evolve. MCP server trust models are re-reviewed when a server's scope changes. The allowlist is audited so that nothing lingers that no longer has an owner. The model picker policy is revisited as new models enter the picker and as AI Credits consumption data comes in, because the cost of premium requests is a governance signal, not just a finance line. Cadence is the difference between governance that holds and governance that erodes quietly.

> DEFINITION
>
> Ownership names who is accountable. Approval paths route each artifact through the control that fits its risk. Review cadence stops the controls from decaying. Together they make the ownership matrix enforceable, not aspirational.

The design principle: build once, inherit enforcement

The three hubs share one design principle. Each artifact category, whether an API, an extension, an MCP server, or a Copilot agent, gets its own governance contract, but every contract is enforced through the same control plane. The artifacts change. The contract does not. This is why the platform is not a compliance layer bolted onto an existing development environment. It is the platform itself.

The payoff is inheritance. Build the control plane once, and every new artifact category that appears after MCP servers and Copilot agents inherits the same enforcement surface automatically. When the next category arrives, the work is to write its governance contract and route it through the existing gates, not to stand up a new governance project. The IDE stays inside the perimeter, but nothing reaches it unvetted. That is the whole point: governance that scales by inheritance rather than by repetition.

DEFINITION

Build once, inherit enforcement. Each artifact category gets its own governance contract; all contracts run through one control plane. The artifacts change, the contract does not, and every new category inherits the same enforcement surface without a new governance project.

References

Primary sources for the deployment scenarios, the Day 0 to Day 2 sequence, and the governance controls this chapter relies on.

- What is Azure API Center: centralized inventory and governance, the authoritative inventory provisioned at Day 0.
- Link an Azure API Center to Azure API Management, the Day 1 step that converges gateway truth and inventory truth.
- Enterprise extension management in Visual Studio Code, the basis for distributing curated VS Code through Group Policy and Intune.
- Use MCP servers in Visual Studio Code, the trust model documented per server before the fleet consumes it.
- Analyze APIs in your API center: Spectral linting and analysis, the Day 2 gate that blocks non-compliant APIs at registration.
- How to find shadow APIs with Dev Proxy and API Center, continuous shadow API detection in steady state.

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