SEMANTIC CONTEXT LAYER FOR AGENTIC DEVOPS · PLAYBOOK

Deterministic context decides whether agents deliver or hallucinate.

AI coding agents fail on context, not capability. SCL-AD treats semantic context as a versioned build artifact: derived from code with compiler accuracy, stored in a governed registry, and delivered to GitHub Copilot through copilot-instructions.md and MCP. The target is 40 to 70 percent less token spend per task, fewer hallucinations, and code that passes review on the first attempt.

AUTHOR

Paula Silva

ROLE

Software Global Black Belt

CONTACT

paulasilva@microsoft.com

READING TIME

~ 45 minutes, end to end

4

ARCHITECTURE LAYERS

6

CHAPTERS

8

CONTEXT RECIPES

40-70%

TOKEN REDUCTION TARGET

CHAPTERS

Chapter 00

The problem and the concept

Why agents fail on context, what that failure costs in tokens, hallucinations, and trust, why RAG and prompt engineering fall short, and what SCL-AD is in one sentence.

Concept →

Chapter 01

Architecture and code intelligence

The four-layer reference architecture, the capability pillars of each layer, and the compiler-accurate code model that grounds GitHub Copilot in facts.

Architecture →

Chapter 02

Context recipes

The eight canonical extractors, from Structural Inventory to Deployment Context: the question each answers, the format it emits, and when to run it.

Recipes →

Chapter 03

Registry, distribution, and integration

The Context Registry that stores and versions artifacts, the refresh and access policies, and the patterns that deliver context to GitHub Copilot.

Distribution →

Chapter 04

Governance, adoption, and metrics

Twelve controls for regulated environments, the five-phase roadmap from pilot to platform, and the metrics that prove Copilot got better, not just cheaper.

Governance →

Chapter 05

Build vs buy, and reference

The three implementation paths on the GitHub and Microsoft stack, the six-question decision framework, the essential glossary, and the consolidated references.

Decision →

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

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 PART I, THE PROBLEM AND THE CONCEPT

The problem and the concept: why agents fail, and what SCL-AD is.

AI coding agents do not fail because the underlying model is weak. They fail because they operate on incomplete, reconstructed, or fabricated context. In a small repository that failure is invisible; across a multi-thousand-repository enterprise estate it becomes the dominant cost driver. This chapter names the failure pattern, measures its cost, and defines the fix in one sentence.

The failure pattern: agents fail on context, not on capability

Enterprise deployments of AI coding agents, with GitHub Copilot in agent mode leading the field, share a common failure pattern. The agent reconstructs context on every interaction: it re-reads files, re-parses imports, and re-infers architectural boundaries on each task. It then generates code that violates conventions the codebase already encodes. The root cause is not the agent, and it is not the model. It is the absence of an authoritative, machine-readable representation of how the codebase actually works.

Three failure patterns recur across deployments. The token reconstruction tax is paid every time an agent opens a session: without persistent semantic context, everything the previous session learned is discarded and recomputed, wasted compute billed per token. The inference gap appears when the agent guesses at facts the codebase already encodes: symbol resolution, transitive dependency versions, and service boundaries are not opinions, they are deterministic properties of the code, and when an agent infers them from raw text it introduces hallucinations into otherwise sound reasoning. Convention drift shows up in code quality: each repository has implicit rules for exception handling, logging, and test organization, and an agent that cannot see those rules produces code that compiles but does not belong.

The cost: tokens, hallucinations, and eroded trust

These patterns compound, and the bill arrives in three currencies. The first is tokens: as adoption scales from pilot teams to platform-wide deployment, token costs grow superlinearly with repository count, because every session on every repository pays the reconstruction tax again. The second is hallucinations: inferred facts that were never facts, type errors, deprecated API usage, and convention violations slipping into otherwise plausible code. The third is trust: every hallucinated change a reviewer catches makes the next agent suggestion harder to accept, and convention drift creates a review and rework burden that offsets the productivity gains the agent was supposed to deliver.

External evidence puts a number on the problem. A 2025 study by METR found that experienced developers took 19% longer to complete tasks when using AI tools in large, mature repositories, with fewer than half of AI outputs accepted as-is. The cause was not the model's reasoning capability; it was the absence of structured context about how the codebase actually works. That result is the sharpest available statement of the thesis of this playbook: in enterprise codebases, context, not model quality, is the binding constraint on agentic coding.

Why existing approaches fall short

Four common approaches address parts of the problem, and none solve it structurally. Retrieval-augmented generation (RAG) retrieves snippets of source code by similarity to the prompt; the result is partial and token-heavy, the agent still infers the relationships between snippets, and the retrieval itself is probabilistic. RAG is well suited to documentation and Q&A, not to grounding code modifications. Embeddings-based search maps code into vector space and retrieves by proximity, which produces approximate matches without semantic guarantees: two functions with similar embeddings may have entirely different runtime behavior, and embeddings cannot distinguish a deprecated method from its replacement.

Prompt engineering encodes context by hand in system prompts, repository instruction files, or chat templates. It works at small scale and breaks at enterprise scale: manual curation is brittle, inconsistent across teams, and blind to the structural facts that change as the code evolves. Custom integrations, calling language

servers, dependency graphs, or static analyzers from inside the agent loop, give the agent live access to facts, but each call is a round-trip in latency and tokens, and the agent still assembles the picture from many small queries. The shared limitation is that all four produce context on demand, per session, per task. They treat context as ephemeral. The structural fix is to treat it as a durable, versioned, deterministic artifact that exists independently of any agent session.

What SCL-AD is, in one sentence

The word context is overloaded, so SCL-AD uses a precise definition. Semantic context is a machine-readable representation of properties that can be derived deterministically from source code: resolved type information, symbol references with their definition and usage sites, direct and transitive dependency graphs, call graphs across functions and services, service interfaces and integration points, configuration that affects runtime behavior, test coverage mapped to production code, architectural boundaries, and the conventions the codebase encodes. It is distinct from syntactic context (the raw text of files), conversational context (the current chat), and probabilistic context (similarity-based retrieval). It is what a competent senior engineer would tell a new hire who asked how the codebase actually works, and its defining property is determinism: the same source at the same commit produces the same context every time it is computed.

On that definition, the Semantic Context Layer for Agentic DevOps is a platform capability that derives, stores, distributes, and refreshes deterministic semantic context for AI coding agents, treating context as a versioned build artifact rather than an ephemeral session input. The framework prescribes what to derive from code, how to derive it through compiler-accurate analysis rather than inference, where to store it, how to expose it through standard interfaces such as the Model Context Protocol, and how to govern it across thousands of repositories. It operationalizes that behavior along four capability pillars: Precompute, Target, Execute, and Coordinate. And it is deliberately not a product: it is a vendor-neutral methodology that an organization can implement with GitHub-native tooling around Copilot, with open-source analyzers, with commercial platforms, or any combination, without changing how agents behave.

> DEFINITION
>
> SCL-AD in one sentence: a platform capability that derives, stores, distributes, and refreshes deterministic semantic context for AI coding agents, treating context as a versioned build artifact rather than an ephemeral session input. Determinism over inference, reusability over recomputation, vendor neutrality over lock-in, auditability over magic, refresh over staleness.

References

Primary sources for the failure pattern, the empirical evidence, and the concepts SCL-AD builds on.

- METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity, the empirical evidence: experienced developers took 19% longer with AI tools in large, mature repositories.
- Moderne Prethink, Semantic Code Context for Coding Agents, reference articulation of the determinism-versus-inference distinction.
- Moderne, Lossless Semantic Tree, a concrete example of compiler-accurate code modeling.
- Anthropic, Effective context engineering for AI agents, foundational guidance on context as a first-class concern.
- GitHub Copilot Documentation, the reference platform for the agent integration patterns in this playbook.
- Model Context Protocol Specification, the open standard for agent context exchange.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 SCL-AD, REFERENCE ARCHITECTURE

Architecture and code intelligence: the substrate of SCL-AD.

SCL-AD organizes semantic context as four layers with explicit contracts. This chapter covers the reference architecture, the capability pillars of each layer, and the code intelligence layer that grounds GitHub Copilot in compiler-accurate facts: indices, symbols, and precompute.

Four layers, one direction of flow

SCL-AD is structured as four horizontal layers, each with a single responsibility, well-defined interfaces, and independent evolution and governance. Layer 1, Code Intelligence, produces a compiler-accurate, format-preserving model of each repository in scope, the substrate for everything else. Layer 2, Context Recipes, holds reusable, parameterized extractors that run against that model and produce semantic artifacts in Markdown, CSV, JSON, and CALM. Layer 3, the Context Registry, stores and distributes those artifacts. Layer 4, Agent Integration, connects the registry to GitHub Copilot, custom agents, MCP clients, and IDE extensions.

Information flows upward: the model feeds recipes, recipes produce artifacts, the registry stores and serves them, agents consume them at execution time. Operational flows run the other way: refresh triggers, governance policies, and access controls travel downward as cross-cutting concerns. In the reference topology, repositories on GitHub Enterprise or Azure DevOps feed a pipeline on GitHub Actions or Azure Pipelines, where Layers 1 and 2 execute; the pipeline publishes artifacts to the registry; and agent runtimes consume them through an MCP server, static files, or the API. Each box is a deployment boundary; each arrow, a stable contract.

> DEFINITION
>
> Identity, observability, security, and lifecycle cut across every layer. The anchor rule: an agent that cannot read repository X must not receive context derived from repository X. Identity propagation is mandatory.

The capability pillars, layer by layer

Layer 1 must satisfy four required properties: compiler accuracy, where the model agrees with what the compiler would produce and types are resolved, not inferred; format preservation, keeping whitespace, comments, and trivia so artifacts reference original source ranges precisely; cross-repository linkage, where a library used by ten services yields ten resolved references, not a name string; and deterministic regeneration per commit. Layer 2 recognizes seven recipe categories: Structural Inventory, Dependency Intelligence, Service Topology in CALM, Convention Extraction, Test Intent Mapping, Configuration Surface, and Security Posture Snapshot; same recipe version plus same commit yields byte-identical output.

Layer 3 recognizes three storage patterns: co-located, artifacts in a .context/ directory versioned with the code, suitable for pilots; a centralized registry indexed for search, refresh, and access control at platform scale; and federated registries under sovereignty constraints. Against staleness, three refresh triggers are required: commit-driven, scheduled, and event-driven. Layer 4 supports four integration modes: static file injection into files such as copilot-instructions.md, which GitHub Copilot reads on session start; an MCP server that Copilot agent mode queries on demand; native integration; and hybrid, static bulk plus MCP detail.

Two normative contracts

Two interface contracts are normative. Every artifact carries a manifest: content-addressed id, recipe category, format, repository URL and commit SHA, recipe name and version, generation timestamp, classification as public, internal, or restricted, and signatures, the basis for traceability, access control, and integrity. The discovery API requires five operations: list repositories with context, list artifacts per repository,

filter by type, fetch content by hash, and search by attribute. Transport is implementation-defined; the semantics are normative.

Code intelligence: compiler accuracy or compounding error

Most code analysis tools are not compiler-accurate: they parse syntax and apply heuristics, fast, but they assume where the compiler resolves facts. A call service.process(item) in Java depends on the static type of service, the runtime type if the call is virtual, the generic parameter of item, the overloads of process, and the active dependency injection configuration. A heuristic parser sees a method call; a compiler-accurate model resolves a specific method on a specific type. When GitHub Copilot asks what calls this function, that gap separates a useful answer from a misleading one, and higher layers compound the inaccuracy.

Seven capabilities define a valid implementation, each testable with fixtures. Symbol resolution ties every reference to its definition, honoring scoping, visibility, imports, and generics. Type resolution assigns a static type to every expression and, under polymorphic dispatch, the set of possible runtime types. Inheritance and implementation graphs capture extends and implements chains. Call graphs include the possible targets of dynamic dispatch, not just the declared one. Dependency resolution pins versions, transitives computed. Configuration mapping links application.yaml, appsettings.json, and .env to consuming code. Source range preservation anchors every fact to file, lines, and columns.

Strategies and per-language tooling

Three strategies cover the space. Native compilers are the gold standard: OpenRewrite LST or Eclipse JDT for JVM languages, Roslyn Workspaces API for .NET, TypeScript Compiler API, Pyright plus LibCST for Python, resolving conservatively and marking the rest dynamic, go/packages and go/types for Go, rust-analyzer for Rust, each with its own toolchain. Unified platforms such as CodeQL, GitHub's semantic analysis engine, reduce operational complexity but add a vendor dependency. Tree-sitter plus type resolvers does not meet the bar alone: it is a fallback whose artifacts carry a lower fidelity label. For COBOL and PL/I, commercial modernization platforms are the practical path.

Indices across repositories, precompute at scale

Single-repository analysis is solved; cross-repository resolution is where most implementations break down. A function in repository A calls a class in repository B, which depends on an interface in repository C; resolving the chain requires all three with consistent versions. Three approaches: a monorepo-style workspace with Nx, Bazel, or Turborepo, the simplest and highest fidelity; an internal package registry such as Artifactory, Azure Artifacts, or GitHub Packages, with accuracy bounded by publish cadence; and coordinated multi-repository analysis, with an inventory, dependency-ordered scheduling, and a final linkage step, the pattern where commercial platforms provide the most value.

At enterprise scale, thousands of repositories and millions of lines, three patterns keep the precompute affordable. Incremental analysis re-analyzes only what changed. Caching by content hash exploits determinism: the same commit produces the same model, reusable across recipe executions. Parallel execution distributes analysis across worker pools. Compiler-accurate analysis costs more than syntactic analysis, so scope it to repositories in active development, in production, and assisted by AI agents. On build versus buy: language-native tooling when one or two languages dominate, a commercial platform for polyglot breadth, or a hybrid; Layer 2 and above contract only with the model interface, so the choice stays reversible.

References

Primary sources for the SCL-AD architecture and the code intelligence layer.

- Moderne Technology, Lossless Semantic Tree, reference example of a Code Intelligence layer.
- Model Context Protocol Specification, the standard for the agent integration layer.
- FINOS Common Architecture Language Model (CALM), standard format for Service Topology artifacts.
- CodeQL Documentation, the GitHub semantic analysis platform.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

02 SCL-AD, CONTEXT RECIPES

Context recipes: eight extractors that turn a codebase into facts GitHub Copilot can trust.

A recipe is a deterministic, scoped, declarative extractor over the code intelligence model. This chapter walks through the eight canonical categories: the question each answers, the format it emits, and when to run it.

What a recipe is

A recipe is the unit of context engineering in SCL-AD: a deterministic function from the code intelligence model to one or more semantic artifacts, parameterized, versioned, tested, and reusable across repositories. Three properties define a valid recipe. It is deterministic: given the same model version and the same parameters, it produces byte-identical output, which is what makes the artifact registry trustworthy. It is scoped: it extracts one specific class of semantic fact, not a broad sweep, and narrow scope is what makes recipes composable. And it is declarative about its outputs: it declares the artifact types it produces, the format of each, and the schema, so consumers, from the registry to GitHub Copilot agent mode, can validate output against the declaration.

Every recipe shares the same anatomy, regardless of language. A YAML or JSON manifest declares identity and contract: name, version, category, inputs such as the required model version and include and exclude patterns, and each output artifact with its format and schema. The implementation traverses the model and emits artifacts; common choices are a Java or Kotlin Recipe class for OpenRewrite pipelines, a C# analyzer for Roslyn pipelines, and Python or TypeScript scripts that call the query API of the model. A fixture-based test suite pairs small input repositories with expected outputs and covers the happy path, empty inputs, edge cases such as circular dependencies and generated code, and error cases. Documentation closes the contract; without it, recipes are not safe to depend on.

DEFINITION

The anchor of this chapter: SCL-AD recognizes eight canonical recipe categories, each answering one distinct question about the codebase. The artifacts they emit are what GitHub Copilot actually reads: the auto-wiring pattern keeps pointers to them in .github/copilot-instructions.md, so agent mode, custom agents, and MCP-connected tools ground every task in extracted facts instead of expensive raw file exploration.

Recipes 1 to 4: what the code is and how it is written

The first four categories describe the static shape of the codebase: contents, dependencies, service boundaries, and style. Implement all eight for repositories where comprehensive grounding is needed, and a subset for narrower use cases.

Structural Inventory

Answers: what is in this repository? Emits Markdown plus CSV listing modules, packages, and top-level types with short descriptions, entry points, and public surface counts. Run it first on every repository; the framework ranks it as the highest leverage per token of effort, because it replaces the exploration phase in which Copilot agent mode opens file after file just to learn the layout.

Dependency Intelligence

Answers: what does it depend on, and what depends on it? Produces CSV plus an SBOM, with SPDX as the reference format, covering direct and transitive dependencies with version, license, and advisory metadata. Use it when Copilot handles upgrades, vulnerability fixes, or migrations; the source ranks it as the most actionable recipe for migration and security agents.

Service Topology

Answers: what services exist and how do they relate? The output is CALM-formatted JSON, the FINOS architecture schema, describing services, their interfaces, and

their relationships. Use it in multi-service estates, so a Copilot custom agent proposing a cross-service change sees the real blast radius instead of inferring it from folder names.

Convention Extraction

Answers: how is code written in this codebase? Emits Markdown describing observed conventions: logging patterns, exception handling, naming, and how consistently each is applied. The framework ranks it as the highest impact on agent output quality; use it when the goal is Copilot generating code that looks like it belongs and passes review on the first attempt.

Recipes 5 to 8: what the code does at runtime

The next four categories capture behavior and operations: tests, configuration, security posture, and deployment. They matter most when agents move from reading code to changing it.

Test Intent Mapping

Answers: what is tested, how, and what is the test telling us? A CSV links each production symbol to its test symbols with intent labels such as happy path or validation. Use it whenever Copilot modifies behavior; it shows the agent which tests protect the code being touched and where the intent of the coverage has gaps.

Configuration Surface

Answers: what configuration affects runtime behavior? Markdown plus JSON map each configuration key to its type, default, environment override, and the exact code sites that consume it. Use it for tasks that touch settings or feature flags; it stops an agent from changing a value without seeing every code path that reads it.

Security Posture Snapshot

Answers: what security-relevant facts are encoded in the code? Markdown covers authentication mechanisms, cryptographic primitives in use, secret references, and

exposed surfaces. Use it before letting agents work near auth or crypto code, and as compact grounding when Copilot assists a security review.

Deployment Context

Answers: what runs where and under which constraints? Added in v1.1.0 as the eighth canonical category, it emits Markdown plus JSON connecting Docker, Kubernetes, and infrastructure-as-code artifacts back to the application code they affect: images, resource limits, replicas, health endpoints, ConfigMaps, and Secrets. It is critical for agents proposing changes with operational impact: without it, an agent can propose a configuration change that compiles cleanly but fails at runtime because of a missing ConfigMap entry.

Authoring, lifecycle, and where to start

Eight guidelines govern recipe authoring. Prefer narrow scope over breadth. Output for both humans and machines: at least one Markdown artifact for review and, where applicable, CSV or JSON for programmatic consumption. Respect determinism: sort outputs in a stable order and keep timestamps out of artifact bodies. Embed traceability, so every fact references its source range in the original code. Fail loudly on ambiguity, emitting explicit unresolved entries instead of guesses. Version the recipe, the model, and the artifact schema separately. Test with real, sanitized fixtures rather than synthetic ones. And keep artifacts compact: a 100,000-line repository should not produce a 100,000-line inventory, because every artifact competes for space in the context window of a Copilot agent session. Recipes then follow a software lifecycle: authored in a recipe repository, tested before release, published to a registry under semantic versioning, deployed through configuration, and deprecated like a library.

Where to start: organizations implementing SCL-AD without a commercial platform build a starter set of three recipes first, Structural Inventory, Convention Extraction, and Dependency Intelligence, and expand to all eight over Phases 2 and 3 of the adoption roadmap. For orientation, the Moderne Prethink documentation publishes two starter recipes: UpdatePrethinkContextNoAiStarter, pure deterministic analysis producing Dependency Intelligence, Test Intent Mapping, Service Topology, and Structural Inventory artifacts, and UpdatePrethinkContextStarter, which adds AI-

generated knowledge graph and test summaries through Bring Your Own Model. Both automatically update agent configuration files, including .github/copilot-instructions.md, wiring the artifacts straight into GitHub Copilot. Custom recipes, from regulatory mappings for PCI-DSS, HIPAA, LGPD, and SOX to internal API extraction and multi-tenancy checks, follow the same anatomy and lifecycle. The test is simple: if the question a recipe answers is deterministic and reusable, it is a recipe; if the answer depends on prompt phrasing or session state, it is a prompt template and belongs in the agent integration layer.

References

Sources for the eight recipe categories, the recipe anatomy, and the reference implementations.

- OpenRewrite Recipe Documentation, the reference model for recipe design.
- Moderne Prethink Documentation, concrete examples of context recipe outputs and the starter recipe definitions.
- Moderne Blog: Changing the AI context, reference for the semantic categories that informed this framework.
- FINOS CALM Specification, the schema for Service Topology artifacts.
- SPDX Specification, the reference SBOM format for Dependency Intelligence outputs.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

03 CHAPTER 03

Registry, distribution, and agent integration: from artifact to agent.

Semantic artifacts only create value when an agent reads them at the right moment. This chapter covers the Context Registry that stores and versions them, the refresh and access policies that keep them trustworthy, and the integration patterns that deliver them to GitHub Copilot and the wider agent ecosystem.

The Context Registry: where semantic artifacts live

The Context Registry sits between two clocks: recipes execute on a build cadence (a commit, a schedule, an event), while agents query for context on a session cadence, at every developer interaction. The registry serves both through five responsibilities. It stores artifacts durably: anything published stays retrievable until explicitly archived. It discovers: agents find artifacts by repository, type, version, or content attribute without knowing the storage layout. It versions: every artifact gets a stable, content-addressed identifier. It enforces access: every read is checked against the same policies that govern the source code. And it observes: every read and write is logged in enough detail to reconstruct what context any agent saw at any moment.

Discovery answers three questions. By repository: given a repo, return all its artifacts, optionally filtered by type or commit. By capability: given an artifact type such as service topology, return the repositories that have it, optionally with a minimum freshness. By content: cross-cutting queries such as which repositories use a given library below a given version. Every artifact also carries a stable URI encoding registry, repository, commit, artifact type, and recipe version: the canonical reference for citations, unchanged for the artifact's lifetime.

Three storage patterns, one maturity curve

Co-located storage keeps artifacts inside the repository they describe, typically under .context/ at the root: the simplest setup, access control identical to the code, versioning by Git, no extra agent configuration, but no cross-repository queries or centralized governance. A centralized registry is a dedicated service with write APIs for recipes and read APIs for agents: native cross-repo queries and central lifecycle management, at the price of a new service to operate. Federated storage links multiple registries through a common discovery layer, scoped by business unit, region, or compliance domain: the answer for data residency and regulated industries. Start co-located in the pilot, centralize at platform scale, federate only when sovereignty demands it.

Versioning, refresh, and access: keeping context trustworthy

Every artifact body is hashed with SHA-256, and the hash is the immutable identifier: recipe runs that produce identical bytes share one hash, which deduplicates storage. The manifest carries four explicit versions, recipe, schema, model, and registry, so a schema change never invalidates older artifacts; it only changes which schema agents validate against. On top of hashes, aliases make the registry usable: latest resolves to the most recent artifact for a repository and type, stable to the most recent one that passed validation, and commit plus SHA to a specific build. Aliases matter because agents ask for the latest service topology, not for a content hash.

Stale context is the failure mode the registry must prevent, so refresh is multi-modal. Commit-driven: re-execute only the recipes affected by the changed files, typically within minutes. Scheduled: daily for active repositories, weekly for stable ones, catching drift from transitive dependency updates. Event-driven: a security advisory regenerates Dependency Intelligence, a major framework upgrade regenerates Convention Extraction. Each artifact type has a freshness window: 24 hours for Dependency Intelligence, Configuration Surface, and Security Posture Snapshot; 7 days for Structural Inventory, Service Topology, and Test Intent Mapping; 30 days for Convention Extraction. Stale artifacts are not removed automatically; their staleness is surfaced so consumers can decide.

DEFINITION

Access control follows two principles without exception. Identity propagation: if a requester cannot read repository X in source control, they cannot read artifacts derived from it in the registry. Least privilege through classification labels that can be overridden upward, never downward. Every read produces an audit entry with requester identity, artifact URI, content hash, access result, and calling context, retained typically 12 to 36 months in regulated environments.

Integration patterns: how context reaches GitHub Copilot

Agent platforms differ in how they consume context, and the integration layer must be lossless, bounded, and controllable: no truncation or paraphrasing at the boundary, static injection within the context window, dynamic queries within the tool call budget, and platform teams able to tune what each agent sees without touching recipes. The first pattern, static file injection, writes selected artifacts to known paths: a .context/ directory with condensed inventory, conventions, topology, and a manifest of URIs to the full artifacts. Its primary target is GitHub Copilot's repository custom instructions in .github/copilot-instructions.md, read on session start across Copilot surfaces (IDE, GitHub.com, Copilot Chat). The pipeline auto-wires a managed block, delimited by clear markers, with pointers to .context/, preserving everything outside the block; without auto-wiring, the integration drifts within days of the first manual edit. Never inject the full Dependency Intelligence or Configuration Surface: they are large and rarely needed in their entirety.

MCP for deep queries, hybrid for production

The second pattern puts a Model Context Protocol server in front of the registry, exposing tools such as list_repository_artifacts, get_artifact, search_dependencies, find_callers, describe_service, and find_convention. It fits when the platform supports MCP natively and token economy demands that only relevant slices reach the model. Two topologies exist: a per-developer server running locally under the developer's identity, or a shared server with identity propagated through OAuth. Either way, queries must be answered in single-digit seconds at worst, with

responses cached by URI and invalidated by registry refresh events; slow retrieval is worse than none, because agents time out and fall back to inference. Production combines the patterns as hybrid distribution: static injection of conventions and a summarized inventory in every repository, managed blocks, an MCP server for impact analysis and dependency lookups, and direct API access for batch tooling.

Per-agent mapping, GitHub Copilot first

GitHub Copilot offers the most complete surface for SCL-AD. Repository custom instructions in copilot-instructions.md carry the static layer. Copilot custom agents take curated, role-specific subsets: a migration agent receives Structural Inventory, Dependency Intelligence, and Convention Extraction; a review agent receives conventions, Test Intent Mapping, and Security Posture Snapshot. Copilot Agent Mode and Coding Agent, which execute multi-step tasks, benefit most from MCP-served dynamic context, with the server registered in the user's or the organization's configuration. Around Copilot, the vendor-neutral AGENTS.md convention is the recommended primary static target, with platform-specific files generated from the same source: ecosystem tools such as Claude Code with CLAUDE.md and Cursor with .cursorrules receive the same managed blocks without a second pipeline. Custom agentic frameworks call the registry API or the MCP server directly, the highest-fidelity path.

> DEFINITION
>
> Six anti-patterns are explicitly discouraged: dumping the full registry into instruction files, live parsing inside the agent loop, snapshots with no refresh pipeline, MCP endpoints that blur the developer's identity, magic prompts that compensate for missing context, and duplicate context stores inside the agent platform. Bypassing the registry, even temporarily, creates artifacts it cannot govern, refresh, or audit.

References

Sources for the registry contract, the distribution modes, and the integration patterns.

- GitHub Copilot Documentation: repository custom instructions, GitHub, the copilot-instructions.md static injection target.
- Model Context Protocol Specification, the standard for dynamic, query-driven context distribution.
- AGENTS.md Convention, the emerging vendor-neutral convention for agent instruction files.
- OCI Distribution Specification, foundation for OCI-based registry implementations.
- SLSA Provenance Specification, reference for artifact provenance and signing.
- Moderne Blog: Changing the AI context, reference for the auto-wiring pattern of agent configuration files.
- Anthropic Engineering: Effective context engineering for AI agents, industry guidance on context strategy.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

04 PART IV, GOVERNANCE AND ADOPTION

Governance, adoption, and metrics.

Controls that make a context registry safe in regulated environments, the roadmap from pilot to platform, and the metrics that separate a cheaper GitHub Copilot from a better one.

Context is not neutral data

Semantic context derived from source code encodes architectural decisions, security postures, dependency footprints, and proprietary patterns. The registry that stores it describes how the organization's software actually works, and every GitHub Copilot session that consumes it inherits that knowledge. Poorly governed, it creates three risks. Information leak: context from a sensitive repository reaches unauthorized agents, and an agent that read the payment service's context has, in practice, read its architecture. Integrity: tampered artifacts make Copilot reason from corrupted ground truth, and trust in the layer is irrecoverable once broken. Compliance: regulated data stored or transmitted against applicable rules. The framework treats governance as a peer to architecture, not a layer bolted on top.

Five principles, five roles

Five non-negotiable principles govern operations. Identity-first: every read, write, and refresh is tied to an authenticated identity, and service accounts are identities while 'the system' is not. Trace-by-default: every artifact is traceable to its recipe, source code, and operator. Least exposure: agents and humans see the minimum context their task requires, default deny. Verifiable integrity: every artifact carries a signature or content hash. Documented purpose: every recipe and integration declares in writing why it exists. Five roles operate the model: platform owner, recipe stewards, repository owners, who decide what is exposed and are not bypassable, security and compliance reviewers, and auditors, who read logs but never artifact bodies.

Six policy domains

Six domains require documented decisions: scope, which repositories are in, with all production-impacting code as the default; classification, labels such as public, internal, confidential, and restricted; access, with identity propagation as the baseline; refresh, staleness windows and their triggers; retention, aligned with source code and regulation; and integration, which agents may consume registry context, reviewed before they call it.

Twelve controls and the audit trail

Twelve controls implement the policies in four families. Identity and access: authenticated identity on every read, an authorization mirror of source code permissions, and just-in-time privileged access. Integrity: artifact signing, content addressing with automatic tamper detection, and recipe attestation linking each artifact to the recipe version that produced it. Lifecycle: stale critical artifacts block agent consumption until refreshed, artifacts of archived repositories are archived within 7 days, and deletion is logged, authorized, and notified. Observability: read and write logging plus anomaly detection over access patterns. The audit trail is append-only, retained 12 months by default and 36 to 84 months in regulated environments. Incident response defines three templates, integrity, disclosure, and availability, all closing with a postmortem within 30 days.

> DEFINITION
>
> The anchor rule is the authorization mirror: reading an artifact requires read access to its source repository. GitHub Copilot sees exactly what its user could already see, never more.

Five phases, from baseline to platform

Phase 0, discovery and baseline, takes 2 to 4 weeks: inventory the agent landscape, which tools, GitHub Copilot at which license tier and adoption per team; capture the baseline metrics; select 1 to 3 pilot repositories; and decide build, buy, or hybrid for the code intelligence layer. Phase 1, the pilot, takes 4 to 8 weeks: implement two or three of the seven canonical recipes, starting with structural inventory, convention

extraction, and dependency intelligence, store artifacts in the repository's .context/ directory, and inject them statically into copilot-instructions.md so Copilot grounds every request in them. Exit bar: token spend per task drops 30% or more, quality metrics improve, and teams report better output.

Phase 2, expansion, takes 3 to 6 months: 10 to 50 repositories chosen for diversity, a centralized registry, the remaining canonical recipes, MCP integration so Copilot's agent mode queries the registry on demand, and the first five governance controls live in production. Phase 3, platform, takes 6 to 12 months: every in-scope repository onboarded, all twelve controls operational, the layer present in the IDE, code review, and internal portals, and ROI demonstrated against the Phase 0 baseline. Phase 4, optimization, is ongoing: unused recipes are deprecated, integrations evolve with the agent platforms, and costs are tuned continuously.

Five recurring adoption risks

Pilot success that does not scale, mitigated with realistic diversity. Governance retrofit, the costliest mistake, mitigated by enabling identity propagation, audit logging, and classification from Phase 1, even at low fidelity. Recipe sprawl, mitigated by naming stewards early and deprecating aggressively. Vendor lock-in, mitigated by enforcing the framework's interface contracts even with a single vendor. And token savings without quality improvement, a cheaper Copilot that is not a better one, mitigated by treating quality as the primary criterion.

Six metrics that prove value

Token spend per task measures what an agent consumes to complete a defined task category, targeting a 40 to 70% reduction from baseline by the end of Phase 2. Hallucination rate measures the fraction of outputs with at least one factual error from incorrect context, such as Copilot citing a nonexistent function or a deprecated API, with measurable reduction by the end of Phase 1. Convention adherence measures how much agent-generated code complies with documented conventions on first generation, targeting at least 30 percentage points of improvement by the end of Phase 2.

Time to first useful suggestion measures the wall-clock time from prompt to the first Copilot output the developer accepts. Developer satisfaction is a periodic NPS-style

survey of trust and helpfulness, where willingness to use is often already high. Pull request throughput is a deliberately tertiary metric, attributed cautiously. Two operational metrics complete the set: registry freshness at 95% or higher, and audit coverage at 100%.

The ROI model

The model has three components. Direct savings: 500 developers, 100 agent-assisted tasks per developer per month, a 50,000 token baseline per task, a 50% reduction, and a blended $0.005 per 1,000 tokens yield roughly $62,500 per month, and that is the floor. Quality: if hallucination rate drops 50% and an average production-impacting agent error costs 8 engineer-hours to remediate, rework savings dominate token savings by an order of magnitude. Platform leverage is the upside after Phase 3, not the justification for Phases 0 through 2. Costs: 2 to 4 platform engineers through Phase 2, tooling, infrastructure, and governance overhead. Positive ROI typically lands between Phase 2 exit and the first quarter of Phase 3.

References

Sources for the governance model, the adoption roadmap, and the metrics.

- NIST Cybersecurity Framework, NIST, the reference behind the control catalog organization.
- SLSA Specification, the provenance and attestation model behind artifact signing.
- ISO/IEC 42001, the AI management systems standard for agentic deployments.
- DORA State of DevOps Reports, the reference for measuring developer platform impact.
- GitHub Octoverse 2024, GitHub, reference data on enterprise AI tooling adoption.
- Effective context engineering for AI agents, Anthropic, industry context on structured context.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 BUILD VS BUY AND REFERENCE

Build vs buy, and the reference material.

SCL-AD is vendor-neutral by design: the framework defines what semantic context is and how it must behave, never what you must buy. This chapter compares the three implementation paths through a GitHub Copilot lens, condenses the essential glossary, and consolidates the references.

The three paths to production

Every organization adopting the framework eventually faces the same decision: build the four layers in-house, buy a commercial platform that already implements them, or combine the two. Path A, build, implements everything on existing platform investments, typically GitHub Enterprise plus Microsoft Azure. Roslyn covers .NET, OpenRewrite covers the JVM, the TypeScript Compiler API and Pyright cover web and Python, and CodeQL adds cross-language call graphs. Recipes run as reusable GitHub Actions workflows, the Context Registry lives on Azure Blob Storage plus Cosmos DB behind Azure API Management, and a GitHub Actions job auto-wires copilot-instructions.md and AGENTS.md on every commit. The reward is sovereignty, native GitHub Copilot integration and customization without limit; the price is a dedicated team of three to five platform engineers for nine to fifteen months, with every recipe authored and maintained in-house.

Path B, buy, adopts a commercial platform that delivers the hardest layer instead of asking you to build it. Moderne is the most prominent example: the Lossless Semantic Tree covers JVM, JavaScript, TypeScript and COBOL, two starter Prethink recipes ship alongside more than 5,000 OpenRewrite recipes, and Moddy exposes it all as an MCP server that GitHub Copilot, Claude Code and Cursor consume directly. A pilot produces real artifacts in weeks, with a team of two to three engineers focused on integration. The costs: an enterprise subscription, vendor dependency for a peer-of-source-control system, and customization bounded by what the platform exposes. Path C, hybrid, is the most common production pattern: buy the

layers that scale across languages, the code intelligence and the recipe catalog, and build what encodes organizational uniqueness, the domain recipes, the registry, the MCP server and the governance on Entra ID and Sentinel.

How to decide, and what it costs

The decision framework asks six questions: how many languages dominate production, one or two tilts build, six or more tilts buy; how much platform engineering capacity is dedicated; how urgent time-to-value is; how strict sovereignty and customization requirements are; whether migration use cases are primary; and how comfortable the organization is with vendor dependency. Language count and engineering capacity dominate in practice. The cost picture: build runs USD 1.5M to 3M over nine to fifteen months, then 1.5 to 2 ongoing FTEs; buy runs USD 400K to 800K over three to six months plus the subscription, then 0.5 to 1 FTE; hybrid sits at USD 800K to 1.5M over six to nine months plus subscription, then 1 to 1.5 FTEs. Over a five-year horizon the paths often converge in total cost of ownership; what differs is the cash flow profile, capex-heavy for build, opex-steady for buy, and the capability each one leaves behind.

The recommended process is short and evidence-driven. Discovery, one to two weeks, documents the language portfolio, repository count, platform investments and sovereignty constraints. Evaluation, three to six weeks, runs a paid vendor pilot in parallel with a build feasibility spike, one or two engineers for two weeks producing a working slice of the Code Intelligence layer for the dominant language. The artifacts are concrete: a measured baseline of token consumption and hallucination rate, a working prototype of one capability pillar, typically Precompute, and a reasoned cost projection. The decision follows immediately, made with evidence rather than intuition, and the chosen path enters Phase 1 of the adoption roadmap.

> DEFINITION
>
> The decision is reversible. An organization that starts on the build path can pivot to buy if engineering capacity proves insufficient, and one that starts on buy can pivot to hybrid if customization requirements emerge that the platform cannot meet. The framework's interface contracts make these pivots manageable rather than catastrophic, which is exactly why SCL-AD stays vendor-neutral by design.

The essential glossary

A handful of terms carry the framework, and each means exactly one thing in every SCL-AD document. Semantic context is a machine-readable representation of properties derived deterministically from source code, distinct from conversational or probabilistic context. A code intelligence model is a compiler-accurate, queryable representation of a repository; the Lossless Semantic Tree, originated by Moderne, is one possible implementation, not a requirement. A recipe is a reusable, parameterized extractor that runs against that model and produces artifacts, the unit of context engineering; the framework recognizes eight canonical categories, from Structural Inventory to Deployment Context. The Context Registry stores versioned, content-addressed artifacts and exposes them through a discovery API, with explicit staleness windows and refresh triggered by commits, schedules or events.

On the agent side, four capability pillars name what the layer does: Precompute derives structured system knowledge ahead of time, Target finds the right code through pre-computed indexes, Execute applies changes deterministically across many repositories, and Coordinate shares change context across agents so duplication and wasted token spend drop. Auto-wiring keeps the agent configuration files, copilot-instructions.md, AGENTS.md and CLAUDE.md, pointed at current artifacts as part of the recipe pipeline, so GitHub Copilot never reads a stale pointer. Static file injection and the MCP server are the two main integration patterns. BYOM lets the pipeline call an approved model to enrich deterministic artifacts with summaries or knowledge graphs, optional and isolated from the deterministic core. And a hallucination is an output asserting a fact absent from the actual context;

determinism, same input, same output, is the property the whole framework deploys against it.

References

The foundational sources behind the competitive positioning, the glossary, and the framework itself.

- Moderne Prethink, Semantic Code Context for Coding Agents, the original articulation of deterministic semantic context.
- Moderne Agent Capabilities, the source of the four capability pillars.
- METR, Measuring AI Effects on Experienced Developer Productivity, empirical evidence that the context problem is measurable.
- Effective context engineering for AI agents, Anthropic, context as a first-class concern in agent design.
- The architect's guide to the AI-native developer platform, GitHub, the enterprise platform view behind the build path.
- Model Context Protocol Specification, the open standard behind the MCP integration pattern.
- OpenRewrite Documentation, the open-source LST and recipe engine available to all three paths.

Paula Silva

SOFTWARE GLOBAL BLACK BELT

paulasilva@microsoft.com

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps