CONTEXT ENGINEERING PLATFORM · PLAYBOOK

Four layers and a harness decide whether agents ship or join the cemetery.

Agents rarely fail in production because the model is weak. They fail because the engineering around the model was never built. The Context Platform Stack names the four layers, cloud and infrastructure, platform engineering, context engineering, and intent engineering, and the harness that runs them as one delivery loop, grounded in GitHub Copilot: agent mode, instruction files, MCP servers, the model picker, and budgets in GitHub AI Credits.

AUTHOR

Paula Silva

ROLE

Software Global Black Belt

CONTACT

paulasilva@microsoft.com

READING TIME

~ 45 minutes, end to end

4	6	62%
ENGINEERING LAYERS	CHAPTERS	DEPLOYMENT RATE WITH LAYER DISCIPLINE

95%

PILOTS WITH NO MEASURABLE VALUE

CHAPTERS

Chapter 00

The Context Platform Stack

The agent cemetery: 95% of pilots deliver no measurable value. The three root causes, the benchmark asymmetry, the four interdependent layers, and how the stack maps to GitHub Copilot.

Introduction →

Chapter 01

Cloud and infrastructure foundation

Where agents run and at what real cost: Kubernetes and Azure building blocks, the four-container pod, the 42/58 cost anatomy, SPIFFE identity, MCP mediation, and the path to autonomous operation.

Layer 01 →

Chapter 02

Platform engineering and governance

Golden paths, guardrails, safety nets, and review gates: how platform teams govern agent-driven delivery, per-agent RBAC, the MCP server registry, and the code review reality check.

Layer 02 →

Chapter 03

Context engineering

Instruction files as infrastructure, the three memory tiers, MCP retrieval discipline, and context budgets measured in GitHub AI Credits: curate the smallest set of high-signal tokens.

Layer 03 →

Chapter 04

Intent engineering

CONSTITUTION.md, EARS specifications, implementation plans, and spec-driven development with a human approval gate that cuts security defects by 73%.

Layer 04 →

Chapter 05

Integration and harness engineering

The four layers as one delivery loop, model routing by SDLC phase, the harness of guides, sensors, and evals, and the 90 day adoption sequence.

Harness →

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 INTRODUCTION

The Context Platform Stack: why agents die before production.

Agents rarely fail in production because the model is weak. They fail because the engineering around the model was never built. This chapter introduces the Context Platform Stack: four interdependent layers, infrastructure, platform, context, and intent, that separate the pilots that ship from the ones that join the agent cemetery.

The agent cemetery: near-universal adoption, single-digit value

MIT's State of AI in Business 2025 found that 95% of enterprise GenAI pilots deliver no measurable value, while 78% of organizations report using AI in at least one function according to McKinsey. Gartner forecasts that 40% of enterprise applications will feature task-specific AI agents by 2026, up from less than 5% in 2025. Near-universal adoption on one side, single-digit delivered value on the other. The gap between executive confidence and deployment reality is not a technology problem. It is an engineering discipline problem, and it is predictable.

The funnel behind the statistics is remarkably consistent. In a typical enterprise budget of 50 million dollars for AI initiatives, 42 proofs of concept get funded in a fiscal year. Eleven reach internal pilot. Three reach production serving real traffic. One generates measurable revenue or cost reduction: 2.4% of the investment. In the same engagements, organizations with a structured approach to the four layers reached a 62% deployment rate, an order of magnitude above the norm MIT documented. The difference is not the model or the tooling. It is layer discipline: each layer built deliberately, owned explicitly, and measured continuously.

Why context is the limiting factor

Three root causes explain 80% of the failures. Infrastructure treated as commodity: agentic workloads are stateful, noisy, and select tools at runtime, so generic Kubernetes works up to about three agents and breaks predictably from thirty onward. Context treated as a detail: the phrase that precedes failure is that the agent will learn on its own, when in reality context is an engineered artifact with memory tiers, a token budget, and weekly human curation. Intent treated as optional: no architecture survives a directive of doing your best, because agents need an explicit hierarchy of trade-offs or they learn an implicit one, usually the wrong one.

A fourth factor compounds the three: benchmark asymmetry. Agents demonstrated on bug-fix tasks are 7 times easier to show off than agents performing real feature work. The same frontier model that scores 74.4% on SWE-bench bug fixes reaches only 11.0% on FeatureBench feature development, so proofs of concept built around bug-fix demos systematically overestimate production capability. Context gives the stack its name because it is where teams underinvest the most: an agent without engineered context is a chatbot with delusions of capability, and everything the agent can know, when it can know it, and at what token cost is a design decision, not an emergent property.

AI-native development also adds two debts to the familiar technical kind: cognitive debt, when developers accept AI-generated code without understanding it, the state Storey (2026) calls cognitive surrender, and intent debt, when objectives, constraints, and rationale are never captured and the agent optimizes the wrong thing the moment conditions shift. Ignoring these debts does not defer the cost; it distributes it to whoever has to maintain, sponsor, or debug an agent that cannot explain itself.

The four layers, and how they relate

The Context Platform Stack defines four interdependent layers, each with a distinct owner, a distinct metric, and a distinct governance conversation. Layer 01, Cloud and Infrastructure, answers where agents run and at what real cost: compute, GPU, Kubernetes, CNCF patterns, observability. Layer 02, Platform Engineering, answers what agents can access and who governs it: the internal developer platform, golden

paths, guardrails, per-agent RBAC, quotas. Layer 03, Context Engineering, answers what agents can know, when, and at what token cost: the ACE pattern, skills, three memory tiers (hot, warm, cold), MCP servers. Layer 04, Intent Engineering, answers what agents must optimize and in what trade-off hierarchy: CONSTITUTION.md, specification-driven development with EARS syntax, intent debt.

The layers are co-dependencies, not sequential steps. Each layer serves the one above and constrains the one below. A robust platform without intent capture produces scaled chaos; intent without context engineering produces specifications nobody can build to; context without platform engineering produces islands of capability that cannot be operated. The four-layer cut is deliberate in both directions: merging context and intent generates drift, because any skill change then alters expected behavior and nobody knows whether it is a bug or a feature, while fragmenting infrastructure and platform into five or more layers dilutes ownership without gain. The stakes are concrete: a well-engineered agent stays within roughly 4,160 tokens per invocation, while a poorly engineered one loads everything into context and pays 30,000 tokens for a trivial task.

> DEFINITION
>
> Whoever controls the agent's context controls its behavior; whoever controls its intent controls its strategy; whoever controls its specifications controls its scale (arXiv 2603.09619). Context, intent, and specifications are not optional refinements on top of infrastructure. They are the stack, and organizational maturity is the minimum of the four layers, never the best one.

What this playbook covers, through the GitHub Copilot lens

This playbook is written for platform engineering leads and AI-native tech leads in enterprise environments, people with 5 to 15 years of systems experience who are frustrated by the gap between vendor promises and production reality. Chapters 1 to 4 cover one layer each. Chapter 5 integrates the layers into a single architecture. Chapter 6 covers harness engineering, the operating discipline named in early 2026 that describes how the four layers compose into a production-grade harness and

how that harness is governed across the agent's lifecycle. Starting from scratch, read 1 through 4 in order, then 5, then 6. Already running Kubernetes, start at Chapter 2 or 3. Already operating agents, start at Chapter 6.

The stack is not abstract, and this playbook grounds it in the tool most teams already have: GitHub Copilot. Agent mode is the stack in miniature. Custom instructions files behave like hot memory, repository-level instructions and curated AGENTS.md files are warm context curation, and MCP servers are the cold retrieval layer with governed access. Model choice in the Copilot picker, across Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, and the GPT-5.x family, is a platform-layer decision with direct cost consequences: since June 1, 2026, Copilot bills consumption in GitHub AI Credits, where one credit corresponds to one cent of usage, so an unmanaged context layer shows up on the invoice within a week. Each chapter uses these Copilot mechanisms as running examples alongside vendor-neutral patterns.

Five actions require no new budget: catalogue the agents already in production, measure the real cost of one agent for at least a week, write the first CONSTITUTION.md for the most visible agent, name an owner for each of the four layers, and schedule a review in 90 days to measure intent debt. The agent cemetery exists because teams optimized for pilots, not production. This playbook inverts that priority: read it, apply it, and stop building agents you will have to bury.

References

Primary sources for the agent cemetery, the layer model, and context engineering.

- MIT NANDA, The GenAI Divide: State of AI in Business 2025, the study behind the 95% figure that frames the agent cemetery.
- Gartner, Press release on task-specific AI agents in enterprise applications, the 40% by 2026 forecast, up from less than 5% in 2025.
- Context Engineering: From Prompts to Corporate Multi-Agent Architecture, arXiv 2026, the source of the context, intent, and specifications control principle.
- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, arXiv 2026, the benchmark asymmetry between bug fixes and real feature work.

- Anthropic, Effective context engineering for AI agents, practitioner guidance on treating context as an engineered artifact.
- Model Context Protocol, the open protocol behind the cold retrieval layer and Copilot's MCP support.

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 LAYER 01, CLOUD AND INFRASTRUCTURE

The cloud and infrastructure foundation: where agents run, and at what cost.

Before an agent can know anything or optimize anything, it has to run somewhere. Layer 01 is the substrate: compute, networking, identity, and environments, sized and governed for agentic workloads, feeding every layer above it.

The deployment gap: why the foundation layer exists

MIT's State of AI in Business 2025 found that 95% of enterprise GenAI pilots deliver no measurable value, while 78% of organizations report using AI in at least one function. The funnel is predictable: of 42 POCs funded in a typical $50M enterprise AI budget, one generates measurable return, 2.4% of the investment. Organizations with a structured approach to the four layers reached a 62% deployment rate, an order of magnitude above that norm. The difference is not the model or the tooling: it is layer discipline, and the discipline starts at the bottom.

Deployment cadence tells the same story: about 47% of organizations deploy AI models only occasionally and only 7% reach daily deployment. The gap is not technical, it is infrastructural: most teams have compute but lack the orchestration, observability, security boundaries, and automation that turn a model artifact into a production system. Layer 01 answers one question: where do agents run, and at what cost. For a GitHub Copilot shop the question is concrete. Agent mode in VS Code feels serverless, but the Coding Agent executes in cloud environments, MCP servers run as real services, and everything Copilot builds eventually lands on this substrate.

The substrate: Kubernetes, Azure building blocks, and agent-shaped extensions

Kubernetes is the de facto runtime for AI in production: two out of three organizations running AI at scale use it as their primary orchestration platform. On Azure that typically means AKS for the agent runtime, Microsoft Entra ID with managed identities for workload identity, virtual networks with private endpoints so agents reach data and tools over private paths, and separate development, staging, and production environments so an experiment never touches production data. The consolidation is accelerating: in March 2026 the CNCF nearly doubled the number of certified Kubernetes AI platforms, over 70% more certifications. That reflects two simultaneous facts: everyone is converging on the same platform, and standard Kubernetes needs deliberate extensions to run agentic workloads properly.

The extensions exist because agents break the assumptions of traditional ML infrastructure. A training job runs to completion and an inference service answers stateless requests; an agent runs for unknown durations, adjusts resources on the fly, calls external services with variable latency, and executes code it did not write. The CNCF Kubernetes AI Conformance Program v1.35 names three capabilities that make this safe: in-place pod resizing, so resources adjust without a restart when an agent shifts from planning to compute-intensive tool use; workload-aware scheduling, which breaks the circular waits that trap FIFO schedulers when agents coordinate; and sandboxing, hard OS-level or VM-level isolation that holds even against hostile agent code. The Cloud Native Agentic Standards of March 2026 add least privilege, MELT observability with metrics, events, logs, and traces, and fault tolerance with retries and circuit breakers.

The four-container pod

At the pod level, a production agentic workload runs four containers, not one: the agent runtime with the model loop and decision logic; an MCP sidecar that terminates MCP connections, enforces RBAC and token rate limits, and emits the audit trail; a context sidecar serving the hot memory layer, the skills resolver, and CONSTITUTION.md; and a telemetry sidecar with the OpenTelemetry collector, decision spans, and MELT metrics. This topology is the first platform engineering decision that determines whether the agent will scale.

Cost anatomy, identity, and the protocol boundary

The cost anatomy of a production agent surprises most teams. In measured production baselines, 42% is compute, the part everyone tracks. The remaining 58% is invisible until the invoice: 28% inference tokens, 15% memory and vector database, 10% network and MCP proxies, and 5% observability. Tracking only compute leaves the TCO estimate off by a factor of two. GitHub Copilot makes the token vector explicit: since June 1, 2026, premium model usage bills as GitHub AI Credits at 1 credit = $0.01, and every agent mode session or Coding Agent task, whether on Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, or GPT-5.x, is a metered line item. Treat those credits as infrastructure spend.

Identity and mediation complete the layer. Every agent carries a distinct SPIFFE SVID, a cryptographic workload identity, and every resource permission is a policy in OPA: a summarizer agent gets read access and a daily quota of 50K tokens; an SRE incident agent gets break-glass access with an approval gate on writes. Mixing those permissions in one agent, the one-big-agent anti-pattern, means the first incident compromises everything. Access flows through the Model Context Protocol: agents never touch databases or APIs directly, they speak MCP to resource servers that enforce policy, log access, and rate-limit. MCP governs 10,000+ active servers, with 97 million monthly SDK downloads in early 2026, under the Agentic AI Foundation at the Linux Foundation, 170+ member organizations as of March 2026, and its scope is deliberately narrow: observability belongs to OpenTelemetry, workload identity to SPIFFE/SPIRE, agent coordination to A2A v1.0, and declarative control to GitOps. The MCP servers you configure for Copilot agent mode today speak the same protocol your production agents will.

> DEFINITION
>
> 42% of a production agent's cost is compute. The other 58%, inference tokens, memory, network, and observability, stays invisible until the invoice arrives. Track all five vectors or your TCO estimate is off by a factor of two.

From IaC to autonomous operation: how the layer feeds the stack

Infrastructure as code is table stakes; the pattern that makes LLM-generated IaC reliable is the verifier in the loop. TerraFormer, from Georgia Tech and AWS, runs terraform plan as feedback for the model and achieves a 15.94% improvement on the IaC-Eval benchmark, outperforming models 50 times its size. The same loop works in Copilot agent mode: encode Terraform and pipeline conventions in .github/copilot-instructions.md, let the agent generate, validate with plan, iterate. The supply chain is the dominant risk on this layer: of 295 GitHub Security Advisories referencing LLM components between January 2025 and January 2026, supply chain accounts for 44%, excessive agency for 20%, and prompt injection for 18%. No single scanner covers every category: combine at least two, such as zizmor plus actionlint, at a median scan of 2.71 seconds per workflow.

The trajectory of the layer is autonomy. Early adopters hit 95% automated provisioning in 2025, it becomes standard for leading companies in 2026, manual infrastructure management turns into a competitive liability by 2027, and full autonomous operation is the enterprise baseline by 2028. The reference architecture for that end state is Cognitive Platform Engineering, four planes on Kubernetes, Terraform, and OPA, data, intelligence, control, and experience, closed by a Sense-Reason-Act loop that senses workload patterns, reasons about configuration, and acts before anyone files a ticket. This is how Layer 01 feeds the stack: platform engineering inherits identity and quota primitives, context engineering inherits MCP mediation and the memory substrate, and intent engineering inherits the audit trail that makes drift measurable. Infrastructure is not plumbing, it is policy: every scheduling decision, network path, and isolation boundary enforces a choice about what is permitted, visible, and safe.

References

Sources for the deployment gap, the substrate requirements, and the standards stack.

- The GenAI Divide: State of AI in Business 2025, MIT NANDA, the 95% no-value pilot figure behind the deployment gap.

- CNCF Platforms Whitepaper, CNCF, the platform engineering foundation the agentic extensions build on.
- Model Context Protocol, the protocol boundary between agents and the infrastructure they depend on.
- Context Engineering: From Prompts to Corporate Multi-Agent Architecture, arXiv 2026, the multi-agent layered architecture this foundation layer supports.
- Effective context engineering for AI agents, Anthropic, the layer above that this substrate serves.

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 LAYER 02, PLATFORM ENGINEERING

Platform engineering: the governance layer of agent-driven delivery.

Agents do not fail in production because the model is weak. They fail because nothing governs what they can touch. Golden paths, guardrails, policies, and review gates are the layer that turns GitHub Copilot agent mode from an operational risk into governed delivery capacity.

Platform engineering became the governance layer

MIT's State of AI in Business 2025 found that 95% of enterprise GenAI pilots deliver no measurable value. That is a governance problem, not a model problem, and platform engineering is where the industry went to solve it. The DORA 2025 assessment found roughly 90% of enterprises already operating internal platforms, and Gartner projects that 80% of large software engineering organizations will run dedicated platform teams by 2026. The role changed along the way: platform teams stopped being internal tools builders and became governance architects.

The trigger is the arrival of agents as operators. When GitHub Copilot agent mode opens a pull request or generates Terraform, it makes infrastructure decisions with autonomy and needs the same governance you would apply to a human operator. The results justify the investment: Spotify's platform team deployed agents that produced more than 1,500 merged pull requests with 60 to 90% time savings on migrations, and Thomson Reuters delivered 15x productivity gains and 70% automation across critical workflows. The structural delta is small: an agentic internal developer platform has seven components, and only two differ from a standard IDP, an Agent API alongside the human portal and per-agent RBAC alongside policy-as-code. Without those two, the platform is good for humans and invisible to agents.

DEFINITION

The operating principle of this chapter, captured at KubeCon EU 2026: agents amplify what is good in your ecosystem and amplify what is bad. Clean golden paths and strong guardrails get executed flawlessly at scale; inconsistent conventions and weak cost controls get replicated across thousands of deployments. Platform hygiene is a precondition for agents, not an optimization.

The four pillars: golden paths, guardrails, safety nets, review gates

The CNCF framework identifies four pillars that platform teams must implement to govern autonomous agents: golden paths, guardrails, safety nets, and manual review workflows. In the GitHub Copilot ecosystem each pillar maps to a concrete mechanism: custom instructions, policy checks on pull requests, automated response, and protected review gates.

Golden paths: from intent to infrastructure

Golden paths define the blessed way to accomplish common tasks: deploy a service, provision a database, request secrets. Hand-written paths do not survive agent velocity, so mature platforms make them generative: the platform team reviews and approves a path template once, and agents reuse it thousands of times. In Copilot terms, the golden path lives where the agent already looks, repository custom instructions that encode team standards and templates exposed through MCP, so agent mode consumes the approved blueprint instead of improvising one. The platform team stops writing Terraform all day and starts writing validation schemas.

Guardrails: from reactive scanning to proactive enforcement

Guardrails stop bad outcomes before production. The historical model, scanners flagging violations after deployment, does not work at agent scale. Policy-as-code inverts it: platform teams describe high-level requirements, PCI-DSS compliance, no hardcoded secrets, cost per service under US$ 10,000 per month, and those requirements become deterministic, executable policies that validate agent-

generated code before execution. One policy definition covers thousands of agent-driven deployments, and human reviewers focus on defining policy and handling exceptions, not on case-by-case validation.

Safety nets: autonomous response

Safety nets catch what slips past the guardrails. When a new CVE drops, the platform generates runtime guardrails and deploys them in minutes instead of the weeks a coordinated patch across hundreds of deployments used to take. Autonomous incident response detects anomalies, triggers agent-driven troubleshooting, and resolves incidents without human intervention, while predictive SRE acts before users feel the impact.

Review gates: strategic friction

Not everything should be automated. Platform teams define manual review gates for high-risk decisions: payment system changes, database migrations, infrastructure quota expansions. These gates slow agents down intentionally. The auditor agent assembles logs, architecture diagrams, and audit trails into a complete compliance package with a risk score, so human reviewers spend their time deciding, not gathering information.

Agents as first-class platform citizens

A mature platform treats the agent as a user persona with its own identity: every agent carries a distinct SPIFFE SVID, a cryptographic workload identity, and every resource permission is a policy in OPA. A summarizer agent gets read access to the production database and a daily token quota; an SRE incident agent gets emergency break-glass access with an approval gate on write operations. Mixing those permissions in one agent, the one-big-agent anti-pattern, means the first incident compromises everything. The same discipline answers the vibe coding tension: a developer asks Copilot agent mode to deploy a service, receives plausible-looking infrastructure code, and submits it without validation. The platform must assume every agent-generated artifact carries risk: mandatory policy validation before merge, cost impact analysis on every deployment, automated rollback when thresholds are violated.

MCP is the interface that makes this scale. Instead of building custom APIs, the platform team exposes capabilities as MCP servers, service catalog, infrastructure provisioning, secret management, deployment workflows, and maintains a discoverable registry of them. When a new agent joins, it discovers capabilities through the registry and operates within the constraints it finds: the agent onboards itself. Since June 1, 2026, Copilot premium usage bills in GitHub AI Credits, one credit equals US$ 0.01, consumed in proportion to the tokens processed, which makes per-agent budgets enforceable in dollars. Routine golden-path execution can run on a cheaper picker model such as Haiku 4.5, while Claude Fable 5 or Opus 4.8 stay reserved for the complex work that justifies their cost.

The code review reality check

The industry narrative about autonomous code review does not match the data. A study of 19,450 pull requests found that code review agents operating alone achieve a 45.20% merge rate against 68.37% for human reviewers, and 60.2% of PRs reviewed only by agents show signal quality between 0 and 30%. An analysis of 278,790 review conversations across 300 projects explains why the two are complementary: AI comments run dense on defects, 29.6 tokens per line of code against 4.1 for humans, humans add understanding and knowledge transfer, and AI-generated code takes 11.8% more review rounds. The pipeline follows directly: Copilot Code Review does the first pass, a human does the second, and no agent review is ever the sole gate.

Merge is not the finish line either. An analysis of 1,210 merged agent-generated bug-fix PRs using SonarQube found that merge success does not reflect post-merge quality, with one agent averaging 8.3 static analysis issues per merged PR. A study of 33,000 PRs across five agents found that documentation and CI tasks merge best, performance and bug fixes worst, and that the most frequent failure mode is reviewer abandonment: unclear, oversized, or poorly described PRs simply get dropped. The platform response is specific: a mandatory post-merge quality gate, PR size limits and description templates for agent-generated code, and post-merge quality metrics instead of merge rate. Organizations that get this layer right report 2 to 5x productivity improvements across engineering teams.

DEFINITION

Before scaling agents, six elements form the minimum viable platform: a complete service catalog, golden paths designed for AI workloads, an MCP server registry, per-agent RBAC, observability for agent trajectories that captures intent and reasoning, and cost governance per agent in AI Credits. Missing any of the six is a blocking issue for agent deployment.

References

Sources for the platform engineering layer, the four pillars, and the code review research in this chapter.

- CNCF Platforms Whitepaper, CNCF TAG App Delivery, the reference definition of internal platforms, golden paths, and platform capabilities.
- Model Context Protocol, the open protocol through which agents discover the capabilities a platform team exposes, including Copilot agent mode.
- The GenAI Divide: State of AI in Business 2025, MIT NANDA, the report behind the 95 percent figure for pilots without measurable value.
- From Industry Claims to Empirical Reality: Code Review Agents, Kennesaw State and Quanta, 19,450 pull requests, the merge rate and signal quality gap between agents and humans.
- Human-AI Synergy in Agentic Code Review, Queen's University, 278,790 review conversations across 300 projects, why AI and human review are complementary.
- Beyond Bug Fixes: Code Quality After Agentic Merges, University of Saskatchewan, 1,210 merged PRs analyzed with SonarQube, why merge rate is not a quality proxy.
- Where Do AI Coding Agents Fail?, Drexel University, 33,000 PRs across five agents, reviewer abandonment as the dominant failure mode.

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 LAYER 03

Context engineering: curate what the agent sees.

Instruction files, repository context, MCP retrieval, and token budgets: the discipline of giving GitHub Copilot agent mode the smallest set of high-signal tokens that maximizes the outcome.

What the agent sees is an engineering decision

Context engineering is the discipline of structuring the complete information payload an agent receives at inference time. Where prompt engineering optimized a single interaction, context engineering asks what the agent actually needs to know, in what order, at what cost, and how it is managed over time. Every request GitHub Copilot agent mode sends carries instruction files, repository excerpts, tool definitions, MCP outputs, and conversation history, and since June 1, 2026 all of it is metered in GitHub AI Credits, 1 credit per $0.01. Anthropic frames the goal precisely: find the smallest possible set of high-signal tokens that maximizes the likelihood of the desired outcome.

More context is not better context. Researchers analyzed 18 leading LLMs and documented severe, unpredictable degradation as context expands, the phenomenon known as context rot. In controlled tests, the same 4K tokens of high-relevance material often outperformed 100K tokens of mixed-relevance material. The implication is direct: curating what the agent sees beats expanding what it could see, and the optimal compression point must be measured per model and workload, not assumed.

Four components that must stay coherent

The discipline rests on four tightly coupled pieces. Composition selects and formats what goes in: facts, instructions, tool definitions, examples. Retrieval supplies context at runtime through semantic search and dynamic tool outputs. Memory manages state across time. Protocols standardize how agents receive and publish

context, primarily MCP. In Copilot terms, composition is your instructions files, retrieval is workspace search plus MCP servers, memory is what persists between agent sessions, and protocols are the MCP configuration itself.

Instruction files are infrastructure, not prompts

The .github/copilot-instructions.md file loads into every request, which turns its size into a recurring bill. Run the math: a 2,000-token instructions file, 1,000 developers, 10 requests per day each. That is 20 million tokens per day before a single line of task work, roughly $60 per day at $0.003 per 1,000 input tokens, or 6,000 AI Credits. Path-scoped .instructions.md files with the applyTo pattern cut that cost by approximately 68%: an instruction about handling payment failures loads only for work in the payments subsystem. Combined with lazy-loaded skills, the same day drops to 6.4 million tokens, about $19.20, and the difference compounds to roughly $15,000 per year at this modest scale.

The strongest evidence is about who writes these files. A controlled experiment across 47 internal repositories showed that human-curated AGENTS.md files reduced runtime by 28.64% and token consumption by 16.58%. LLM-generated equivalents added 3.02% overhead, worse than having no file at all: auto-generated specifications document the obvious and omit the critical, the tacit team knowledge only humans can identify. The conclusion: instruction files are engineering documentation, peer-reviewed, versioned, treated as production code, with a named owner and a curation cadence.

> DEFINITION
>
> A context file has exactly two states: curated signal or accumulating noise. The test is ownership. If nobody reviews changes to copilot-instructions.md in pull requests, the file is drifting toward noise, and every developer pays for that drift in AI Credits on every request.

Memory tiers and retrieval discipline through MCP

A well-engineered agent loads by tier, not by default. Hot memory, about 660 tokens, is always in context: agent identity, inviolable conventions, safety hooks, updated only on deploy. Warm memory, about 2,000 tokens per activated skill, loads on demand: specialist instructions, per-repository AGENTS.md, domain schemas, curated weekly by tech leads. Cold memory, about 1,500 tokens per query, is never preloaded: the corporate knowledge base, production data, and code, retrieved through MCP servers with per-agent access control applied. The result stays within roughly 4,160 tokens per invocation, while a poorly engineered agent dumps everything into hot memory and pays 30,000 tokens for a trivial task.

The tiering is not theoretical. One production deployment applied it to a 108,247-line C# backend over 12 weeks, spanning 283 agent sessions and 2,801 user prompts. Tokens per task fell 32%, from 18K to 12.2K per invocation, p95 latency fell 47%, from 9.2 to 4.9 seconds, and task completion rose 18%. Skills apply the same principle through lazy loading: descriptions of 50 to 100 tokens stay visible, full bodies load only on invocation; with roughly 70% of installed skills unused in a session, that alone cuts effective context cost by 70%.

MCP context has smells; audit it

Every MCP server enabled in Copilot contributes tool descriptions to the context, and those descriptions are prone to what researchers call tool description smells. A tool described as 'processes transactions' invites the model to hallucinate the missing details; a spec without error documentation fails silently when the real tool throws an exception. Auditing descriptions for vagueness, missing parameter constraints, and absent error handling is unglamorous, essential context engineering. The skill ecosystem shows neglect at scale: of 55,315 public skills surveyed, 26.4% lack a routing description entirely, making them undiscoverable, and over 60% of body content is non-actionable filler. Stripping that noise compressed descriptions by 48% and bodies by 39% while improving functional quality by 2.8%.

Context budgets and hygiene, measured in AI Credits

Treat the context budget like a latency budget: an explicit number per invocation, tracked in AI Credits. Budgeting is not blind shrinking, though. The compression paradox is documented: aggressively compressing context reduced input tokens by 17% and increased total session cost by 67%, because the model burned far more output tokens reconstructing meaning. The governing idea is semantic density. Tokens that carry real information, descriptive names, type annotations, well-written docstrings, are investments; boilerplate and ceremony are pure waste. Instructions files should use rich, clear natural language rather than abbreviations, and cuts should remove noise, never signal.

Hygiene means resisting two failure modes over time. Brevity bias discards domain-specific insight in the name of compression. Context collapse is worse: in one documented run, an agent context of 18,282 tokens achieved 66.7% accuracy; rewritten for brevity, it fell to 122 tokens and 57.1%, below the 63.7% no-context baseline. The alternative is deliberate curation: the ACE approach, generation, reflection, and curation over an evolving context, delivers +10.6% accuracy over static-context baselines on agent benchmarks and +8.6% on financial analysis tasks.

Match context depth to the model in the picker

Workload-to-model matching is context engineering, and the Copilot model picker is where it happens. Fast action work, quick edits and tool executions, runs lean on Sonnet 5 or Haiku 4.5: optimize for latency, not context depth. Extended reasoning over complex state, compliance reviews, architecture decisions, belongs to Claude Fable 5 or Opus 4.8, where 50K to 100K tokens per invocation can be a justified spend. Compaction and subagent handoffs are Haiku 4.5 territory: minimal context, maximal specificity. Multimodal parsing of PDFs, diagrams, and tables needs a vision-capable pick such as Gemini 3.1 Pro or a GPT-5.x model. A thinking workload on a fast model underdelivers; an action workload on Opus 4.8 overprovisions AI Credits.

References

Sources for the curation discipline, the memory tiers, and the token economics.

- Effective Context Engineering for AI Agents, Anthropic, the smallest set of high-signal tokens framing.
- Agentic Context Engineering, arXiv, evolving contexts, brevity bias, and context collapse.
- Codified Context: Infrastructure for AI Agents in a Complex Codebase, arXiv, the three-tier production numbers from a 108K-line C# backend.
- A Survey of Context Engineering for Large Language Models, arXiv, the research map of composition, retrieval, memory, and protocols.
- Model Context Protocol, MCP, the open protocol behind tool and data retrieval in Copilot agent mode.

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 LAYER 04, INTENT ENGINEERING

Intent engineering, expressing what to build.

Context tells the agent what is true. Intent tells it what to want. This chapter turns organizational purpose into artifacts that GitHub Copilot agents can read, verify against, and be blocked by.

Context without intent optimizes for the wrong thing

The Klarna case from Q3 2025 shows what happens when this layer is skipped. The agent had everything context engineering can provide, from detailed customer data to pricing rules and execution history, yet within weeks it was optimizing for metrics the organization never wanted to prioritize, and by the time anyone noticed, the misalignment had already produced substantial operational and reputational cost. Context engineering makes agent decisions more accurate for a given objective, but it cannot say what the objective should be. A well-contextualized agent without intent is a pilot with weather data and fuel specifications but no flight plan.

The four-level pyramid

The source hierarchy has four levels. Level 1, prompt engineering: one instruction for one model in one interaction; it does not scale across sessions or teams. Level 2, context engineering: information payloads, memory tiers, and MCP retrieval, covered in the previous chapter. Level 3, intent engineering: goals, values, and trade-off hierarchies, answering what the system exists to achieve and which constraints are non-negotiable. Level 4, specification engineering: a machine-readable corpus of policies, quality standards, and implementation constraints for coherent multi-agent operation at scale. Most organizations operate at levels 1 and 2; the distance to 3 and 4 is where intent debt compounds.

Intent debt, a debt of clarity

Intent debt is not a debt of effort but of clarity: it accumulates when objectives, constraints, and the reasoning behind them live in a Slack thread from eight months ago or in the head of one engineer instead of in durable artifacts. Three mechanisms turn it into risk: knowledge concentration, where departures leave undocumented constraints behind; optimization drift, where unstated objectives push humans and agents toward measurable proxies; and compounding absence, where each new agent adds divergence with no force pulling the system back. When agents make consequential decisions unsupervised, that debt is operational risk.

Three artifacts, and prompts as contracts

Intent engineering materializes as three repository artifacts. CONSTITUTION.md is a human-first document of principles and non-negotiable values, written for humans and structured for machines to enforce. Five mandatory sections: Identity, with scope and limits; Values and trade-offs, an ordered hierarchy of the form prefer X to Y, always; Hard constraints, the things the agent never does; Protocols for recurring situations; and Escalation hooks, which define when the agent stops and calls a human. Around 200 lines, committed to Git.

SPECIFICATION.md in EARS notation

The specification translates constitutional principles into machine-testable acceptance criteria using EARS, the Easy Approach to Requirements Syntax from ISO/IEC/IEEE 29148, created for Rolls-Royce in 2009. The vague requirement of resilience to network failures becomes: when a network call fails 3 times in 60 seconds, the system shall open the circuit breaker; while the circuit is open, it shall serve cached responses for up to 30 seconds. Requirements stop being prose and become conditions an agent can verify before executing.

IMPLEMENTATION_PLAN.md and the Copilot repository

The plan breaks work into atomic tasks with [P] markers for what can run in parallel and doubles as a control boundary: the agent knows exactly what it may modify. In GitHub Copilot the trio lives in the repository as versioned contracts: custom instructions in .github/copilot-instructions.md carry constitutional principles into

every request, reusable prompt files in .github/prompts/ encode the specification rituals, and a custom agent reads CONSTITUTION, SPECIFICATION, and IMPLEMENTATION_PLAN before writing any code. A prompt that matters is not disposable chat text; it is a reviewed file with an owner and a diff history.

> DEFINITION
>
> A well-contextualized agent with no written intent is an arrow with excellent sighting and no target. Intent engineering supplies the map, the destination, and the trade-offs the agent is authorized to make.

Spec-driven development, the executable workflow

Spec-Driven Development organizes agent-assisted work into phases with approval gates: human and planner draft CONSTITUTION.md and SPECIFICATION.md in Ask mode, with no file writes; the specification is refined into an IMPLEMENTATION_PLAN.md; a human approval gate follows; and only after explicit sign-off does the agent enter Agent mode, with writes scoped to the plan and code review gating the merge. The gate is a technical boundary enforced by the orchestration layer, not a soft convention, and constitutional AI research reports that SDD reduces security defects by 73% versus unconstrained AI-assisted development.

The planner-coder gap

A study of multi-agent system failures found that 75.3% originate in the gap between planner and coder: ambiguous or incomplete specifications become insecure or incorrect code. The gap is also a security vector: an attacker who influences the planner through prompt injection can make the coder generate vulnerable code without ever touching it. SPECIFICATION.md is the interface between planning and coding; making it explicit, machine-parseable, and human-reviewed closes the surface where three quarters of failures start.

Modes, models, and AI Credits

The workflow maps onto the Copilot model picker. Plan in Ask mode with a premium model such as Claude Opus 4.8, or Claude Fable 5 for the hardest architectural trade-offs; implement in Agent mode with Claude Sonnet 5 as the workhorse. Planning makes few calls, so it costs few GitHub AI Credits, the billing unit since June 1 2026, where 1 credit equals $0.01; execution makes many. Two disciplines protect intent and budget: hooks that block writes outside the plan run in the orchestration layer and consume zero credits, and extended thinking stays in planning, because on iterative implementation it adds 43% to token cost and degrades quality by 30% across 4,018 SWE-Bench trajectories.

Verify intent, not syntax

Agent-generated tests do not verify intent. In a six-LLM study on SWE-bench Verified, 74.4% of the traces from the strongest agent included tests written by the agent itself, but those tests act as observational feedback, closer to print statements than to assertions; GPT-5.2 reached 71.8% success while writing almost no tests. The concerns must stay separate: SPECIFICATION.md defines what must be tested, and a human reviews whether the generated tests actually test it. Under semantic changes, pass rates of LLM-generated tests drop from 79% to 66% and branch coverage from 76% to 60%; after significant refactoring, regenerate and re-review instead of trusting them.

The metrics of drift

Four signals reveal drift early. Scope creep rate per agent session measures how often the agent proposes work outside the plan. Percentage of edits outside IMPLEMENTATION_PLAN.md tracks what hook enforcement reveals and should trend toward zero. Human review time per agent-generated pull request is a leading indicator of under-specified plans. And cognitive debt indicators, such as onboarding time, how many people can explain why a constraint exists, and reluctance to modify specific files, expose intent that was never written down.

Two levers the artifacts do not replace

A study of 9,374 trajectories across 19 agents, 8 frameworks, and 14 LLMs found that the model drives outcomes more than the framework, and that successful trajectories follow the sequence understand, reproduce, fix, verify. The Copilot picker choice is therefore a first-order governance decision, and a hook that requires the agent to state its understanding before editing eliminates a whole class of failures. The second lever is human: AI use during learning reduces library-specific skill acquisition by 17%, and the patterns that preserve it, conceptual inquiry, code with explanation, and generation followed by comprehension, protect the skill intent engineering depends on: writing and evaluating specifications.

References

Sources for the four-level pyramid, the intent artifacts, the SDD workflow, and the verification data.

- Context Engineering: From Prompts to Corporate Multi-Agent Architecture, arXiv 2026, the four-level hierarchy from prompts to specification engineering.
- Storey (2026), arXiv:2603.22106, intent debt, its three risk mechanisms, and the drift metrics.
- Specification-Driven Development for AI-Assisted Software Engineering, SANER 2026, the CONSTITUTION, SPECIFICATION, and EARS artifact structure.
- Constitutional AI reduces security defects in specification-driven development, the 73% reduction versus unconstrained AI-assisted development.
- Planner-Coder Gap in Multi-Agent Systems, the 75.3% of multi-agent failures born between planner and coder.
- Rethinking the Value of Agent-Generated Tests, six LLMs on SWE-bench Verified, tests as observational feedback.

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 INTEGRATION AND HARNESS

Integration and harness engineering: run the stack as one system.

The four layers of the Context Platform Stack only pay off when they run as one delivery loop, and that loop only stays trustworthy when a harness of guides, sensors, and evals wraps every agent. This chapter assembles the loop, names the harness, and closes with a 90 day adoption sequence.

One delivery loop, four layers

The Context Platform Stack is not a sequential process, it is a dependency graph. Cloud and infrastructure answer where agents run and at what cost. Platform engineering adds governance, the internal developer platform, golden paths, RBAC, and cost controls that answer what agents can do and who can deploy them. Context engineering adds cognition, skills, memory tiers, and MCP servers that answer what the agent knows. Intent engineering adds strategy, CONSTITUTION.md and specifications that answer what the agent should optimize. The flow is bidirectional: intent constrains context, context determines which platform capabilities matter, and platform choices constrain infrastructure. Most failures happen at those feedback loops: infrastructure built without intent, context grown without governance, agents shipped without specification.

Skipping the loop leads to the agent cemetery, the repository of abandoned prototypes that almost every large enterprise accumulates. MIT's State of AI in Business 2025 found that 95% of enterprise GenAI pilots deliver no measurable value, while organizations that work the four layers as one system reached a 62% deployment rate in the engagements behind this playbook. Benchmarks explain part of the gap: Claude Opus 4.5 scores 74.4% on SWE-bench bug fixes but only 11.0% on FeatureBench, the ICLR 2026 benchmark of 200 feature level tasks, roughly 7 times harder than bug fixes. A POC that only demos bug fixes operates in the easy zone and overestimates production capability.

DEFINITION

Five early warnings that you are heading toward the cemetery: agents without SLOs or daily KPI tracking, no named owner for the agent's evolution, no golden path for deploying the next agent, context maintained by a single person, and intent drift that goes undetected for more than two weeks. Fix the discipline before scaling the fleet.

Route models by SDLC phase, and pay in AI Credits

Model choice is task and phase specific, not global, and the GitHub Copilot model picker is where the discipline lands. Specification and architecture require decomposition under uncertainty, so they route to a premium reasoning model, Claude Opus 4.8 or Claude Fable 5 with extended thinking, in ask mode. Feature implementation across 3 to 10 files routes to Claude Sonnet 5 in agent mode with hooks enabled. Docstrings, commit messages, and PR summaries route to Claude Haiku 4.5. Gemini 3.1 Pro and the GPT-5.x family cover the equivalent tiers for teams standardized on other vendors. The research behind the split is specific: extended thinking costs 43% more and degrades quality by 30% on iterative implementation tasks, so premium reasoning inside tight coding loops is both wasteful and worse.

Since June 1, 2026 this routing table is also a budget instrument, because every premium call in Copilot bills in GitHub AI Credits and one credit corresponds to one cent of model usage at list price. A team that reserves Opus 4.8 for specification and review, runs implementation on Sonnet 5, and pushes summaries to Haiku 4.5 pays for cognitive depth exactly where it buys quality. The maturity map tells you where to start: stage 0 is chaos and stage 4 is intent driven, with every layer active. Your stage is the minimum of the four layers, not the best one: DORA metrics without AGENTS.md curation is stage 1, and fifty agents without intent metrics is still stage 2. Each transition takes one or two quarters.

The harness: guides, sensors, and evals

The formula that crystallized in early 2026 is compact: an agent is a model plus a harness. The model provides the reasoning; the harness is everything around it, tools, memory, state, gates, and validation. Mitchell Hashimoto coined engineer the harness on February 5. On February 11, OpenAI's Ryan Lopopolo reported three engineers shipping roughly a million lines of production code with zero human written code and named the discipline harness engineering. On February 17, LangChain jumped from rank 30 to rank 5 on Terminal Bench 2.0 by changing only the harness, with the model held constant. By April 3, Microsoft Agent Framework 1.0 went GA with a layer literally called Agent Harness. The model is the pluggable component; the harness is the differentiator.

In March 2026, Martin Fowler and Birgitta Böckeler organized the harness into guides, which steer the agent, and sensors, which verify it. Anthropic's three agent pattern shows why the split matters: a Planner expands the prompt into a specification, a Generator implements in sprints, and an Evaluator clicks through the application like a real user before any sprint closes, with a shared definition of done agreed up front. Research attributes 75.3% of agent failures to the planner coder interface, and this pattern makes that interface explicit and verified.

Guides steer the agent

In GitHub Copilot the guides are concrete files: .github/copilot-instructions.md as the repository map, path scoped .instructions.md files, reusable .prompt.md files for recurring tasks, and MCP server configurations that define which tools exist. Keep the map around 100 lines pointing to deeper docs, an index rather than an encyclopedia. And curate it by hand: across 47 internal repositories, human curated AGENTS.md files cut runtime by 28.64% and token consumption by 16.58%, while LLM generated versions added 3.02% overhead.

Sensors verify the output

Deterministic sensors are cheap and catch most failures: custom linters that enforce dependency direction, type checkers, structural tests, pre commit hooks, CI gates that block merges, and schema drift detectors. Inferential sensors, LLM as judge review and Copilot code review automation on pull requests, catch the semantic judgment that linters cannot. LangChain's lesson: often the problem is not the model,

it is the missing sensor; loop detection and mandatory self verification drove their Terminal Bench jump. Evals close the set: a golden set of representative prompts runs in CI on every harness change and blocks regressions before they ship.

The 90 day adoption sequence

Hashimoto's definition is the operating discipline: every time the agent makes a mistake, engineer the environment so that mistake becomes structurally impossible to repeat. The loop has six steps: observe the failure in a captured trajectory, classify the root cause, decide the fix type, a new sensor, tool, instruction update, gate, or sub agent, implement the fix, add a regression test, and audit future trajectories to confirm the fix held. Trajectories captured from day one become a compounding asset: every session produces training data, eval data, and edge case documentation.

The sequence itself fits in one quarter. Days 1 to 30: pick one use case with a verifiable success criterion, write .github/copilot-instructions.md, instrument trajectories, define a minimum approval policy, and run the first experiment against a measured baseline. Days 31 to 60: plant deterministic sensors, encode the three to five most important architectural rules as mechanisms, and add self verification plus LLM as judge review on pull requests. Days 61 to 90: set up identity and audit for the agents in use, replicate the pattern to a second repository, and measure against the baseline. Measure beyond velocity too: a study of 2,989 developers identified six dimensions that DORA style metrics miss, self sufficiency, cognitive load, task completion, peer review quality, expertise development, and code ownership.

References

Primary sources for the integration loop and the harness engineering discipline.

- My AI Adoption Journey, Mitchell Hashimoto, February 5, 2026, the post that coined engineer the harness.
- Harness engineering: leveraging Codex in an agent-first world, Ryan Lopopolo, OpenAI Engineering Blog, February 11, 2026.

- Improving Deep Agents with harness engineering, LangChain, February 17, 2026, the Terminal Bench 2.0 case study.
- Harness engineering for coding agent users, Birgitta Böckeler, martinfowler.com, March 2026, the guides and sensors taxonomy.
- Harness design for long-running application development, Anthropic Engineering, March 2026, the Planner, Generator, Evaluator pattern.
- Microsoft Agent Framework Version 1.0, Microsoft, April 2026, the GA release with the Agent Harness layer.
- FeatureBench: Benchmarking Agentic Coding for Complex Feature Development, arXiv 2026, the benchmark behind the feature work versus bug fix gap.

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