THE FULL AGENTIC ENGINEERING PLATFORM · PLAYBOOK

Six disciplines decide whether agents ship or join the cemetery.

Most agentic pilots die in production, not for lack of ambition or budget, but for lack of engineering. This playbook lays out the full stack in six disciplines, Cloud, Platform, Context, Intent, Harness, and Token Economy, framed on GitHub Copilot. Since June 1, 2026, when GitHub AI Credits made cost per work performed, that discipline pays between 30 and 70 percent less on the bill.

AUTHOR

Paula Silva

ROLE

AI-Native Software Engineer

CONTACT

in/paulanunes

READING TIME

~ 45 minutes, end to end

6
ENGINEERING DISCIPLINES

6
CHAPTERS

8
OPTIMIZATION PATTERNS

30-70%
TOKEN COST REDUCTION

CHAPTERS

Chapter 00

The agent cemetery and the six disciplines

Why agentic pilots die in production, the three debts that compound, the six-discipline stack in sequence, and the June 2026 GitHub AI Credits change that makes discipline pay.

Foundations →

Chapter 01

Cloud foundation and platform engineering

Kubernetes for long-lived agents, MCP as the protocol layer, golden paths and SPIFFE identity, and the CPE four-plane architecture with supply-chain security.

Substrate →

Chapter 02

Context engineering

The concave utility curve and context rot, memory tiers with copilot-instructions.md and MCP, six compounding techniques, and prompt caching.

Cognition →

Chapter 03

Intent engineering

Intent debt and the Klarna case, the Spec Kit artifacts, Spec-Driven Development with its human gate and zero-cost hooks, and the planner-coder gap.

Strategy →

Chapter 04

Harness engineering

Agent equals model plus harness, the guides and sensors taxonomy, four production patterns, the seven-step playbook, and two paths on one Microsoft foundation.

Operating model →

Chapter 05

Token economy, FinOps, and adoption

Model routing in the Copilot picker priced in AI Credits, FinOps for AI, eight optimization patterns, Azure AI Foundry, and the 30, 60, 90 day roadmap.

Funding →

READING TIPS

KEYBOARD SHORTCUTS

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

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 THE CEMETERY PROBLEM

The agent cemetery and the six disciplines.

Most agentic pilots never reach production. The gap is not ambition or budget. It is engineering. An agent that works in a demo dies in production because the team skipped layers: cloud, platform, context, intent, operating model, and token economy. This chapter names why pilots die, lays out the six disciplines in sequence, and states the two facts that make the discipline pay for itself.

Why pilots die in production: three debts that compound

Agentic pilots do not die because the model is weak. They die because the team shipped without a coherent engineering foundation, and three debts compound underneath the demo. The first is technical debt, the familiar kind: patches, fragile dependencies, brittle tests. A large-scale study by Liu and colleagues analyzed 304,362 AI-generated commits across 6,275 repositories and found higher shares of requirement debt and test debt than in human-written code. AI assistance does not remove debt. It changes the shape of the debt.

The second debt is cognitive, and it is more dangerous because review does not catch it. Developers accept generated code without understanding why it works. Storey calls this cognitive surrender. The codebase becomes orphaned knowledge: it runs, but nobody on the team can modify it safely because nobody understands the reasoning behind it. The productivity gain on day one becomes a maintenance liability on day ninety.

Intent debt: the failure that looks like success

The third debt is intent debt, the newest and most insidious. The agent was built for metric X, under constraint Y, with hidden assumption Z. When production conditions shift, the agent keeps optimizing for the wrong thing. It is not broken. It was never

fully specified. Intent debt is the reason an agent can pass every test and still do the wrong work, and it is why the specification, not the code, is the artifact that has to be reviewed first.

The six-discipline stack, in sequence

A stack that survives production covers six disciplines, and the order matters. Cloud comes first: Kubernetes and the Model Context Protocol give agents a substrate that holds state across long sessions instead of treating each call as a stateless web request. Platform comes second: golden paths, an internal developer platform, and Backstage templates bake security and FinOps controls into every agent by default, so governance is not a paper exercise. Context comes third: the smallest set of high-signal tokens the agent needs, assembled through memory, MCP, and retrieval, and precomputed in CI rather than reconstructed on every task.

Intent comes fourth: Spec Kit makes the goal legible through a constitution, a specification, and an implementation plan, so the agent optimizes for the outcome you actually want. The operating model comes fifth, the Harness discipline: cadence, ownership, evidence, and review that turn the framework from slides into shipped software, delivered through Three Horizons for Microsoft-aligned teams or Open Horizons for the CNCF stack. Token economy comes sixth: FinOps for agents, with budgets per session, agent, and repository, and Azure AI Foundry for model routing. Skip any layer and the pilot joins the cemetery.

The June 2026 fact: billing moved from intent to work performed

On June 1, 2026, GitHub Copilot stopped charging per request. Premium Request Units retired. GitHub AI Credits took their place, and the conversion is simple: 1 AI Credit equals US$0.01. Input, output, and cached tokens are now accounted at the published API rate of each model in the Copilot picker, from Haiku 4.5 for trivial work to Sonnet 5 for code and Opus 4.8 for planning, alongside Claude Fable 5, Gemini 3.1 Pro, and the GPT-5.x models. Code completions and Next Edit Suggestions stay included in the plan and consume no credits.

The change is structural, not cosmetic. Under Premium Request Units, cost was per intent: one request, one charge, regardless of how much work it triggered. Under GitHub AI Credits, cost is per work performed. An agent-mode session that fans out across a repository costs what it actually consumes. That makes token discipline a financial control, not a nicety. Every technique in this playbook, from precomputed context in CI, to model routing in the Copilot picker, to prompt caching, now shows up directly on the bill.

The numbers that justify the effort

The case for the platform rests on evidence, not a vendor deck. Deloitte surveyed 3,235 respondents across 24 countries and found that 75 percent of enterprises plan agentic AI within two years, yet only 34 percent report using AI to deeply transform the business. The gap between planning and transformation is engineering debt, and it is predictable. A causal study from Carnegie Mellon using the AIDev dataset found that ungoverned coding agents add 39 percent to cognitive complexity and 18 percent to static-analysis warnings, and that this debt accumulates persistently instead of self-correcting.

The discipline pays for itself. Teams that treat tokens as a finite resource, apply context engineering, route to the smallest model that passes the eval, and instrument cost pay between 30 and 70 percent less than teams that run GitHub Copilot as a black box. On agentic workloads specifically, the applied optimization patterns compound to a reduction in the same range. The difference between the cemetery and the platform is not talent. It is applied discipline, in sequence, across six layers.

> DEFINITION
>
> The whole playbook in one sentence: pilots die from three compounding debts, six engineering disciplines in sequence keep agents alive in production, and since June 1, 2026, when GitHub AI Credits made cost per work performed, that discipline pays between 30 and 70 percent less on the bill.

References

Primary sources for the failure pattern, the six-discipline stack, and the two facts that justify the platform.

- Deloitte, State of AI in the Enterprise 2026, the survey behind the gap: 75 percent of enterprises plan agentic AI, only 34 percent report deep transformation.
- Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild, the technical-debt evidence across 304,362 AI-generated commits in 6,275 repositories.
- AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development, the causal measurement of how coding agents change engineering output.
- GitHub Docs, Usage-based billing for GitHub Copilot: GitHub AI Credits, the June 2026 billing change: 1 AI Credit equals US$0.01, cost per work performed.
- CNCF, Platforms Whitepaper, the reference for the platform discipline and golden paths.
- Spec Kit, toolkit for Spec-Driven Development, the tooling behind the intent discipline: constitution, specification, and implementation plan.
- Anthropic, Effective context engineering for AI agents, foundational guidance for the context discipline.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 SUBSTRATE AND GOVERNANCE

Cloud foundation and platform engineering.

The first two layers of the stack decide whether an agent ever reaches production. Cloud infrastructure gives agentic workloads a place to run. Platform engineering decides what those workloads are allowed to do. This chapter covers both: how Kubernetes changes when the workload is a long-lived agent instead of a stateless request, why the Model Context Protocol collapses integration cost, how golden paths and workload identity turn governance from a document into enforced code, and how the CPE four-plane architecture and supply-chain controls hold the agentic stack together.

Kubernetes for agentic workloads

Kubernetes has become the default runtime for AI in production. Two of every three organizations running AI at scale use it as their primary orchestration platform. It did not win because it was built for machine learning. It won because it already solved distributed, stateful workloads at enterprise scale. The Cloud Native Computing Foundation then extended that base for AI. In March 2026 the Kubernetes AI Conformance Program certified more than 70% additional platforms, a signal that standard Kubernetes needs deliberate extensions to run agents safely.

The workload profile is what changes. A training job runs to completion. An inference service answers a request and returns. An agent does neither. Agentic sessions live for hours, hold KV cache across steps, and fan out to subagents. Three infrastructure capabilities address this directly. In-place pod resizing lets an agent shift from lightweight planning to compute-heavy tool use without a restart, so the pod stays scheduled and only its reservation changes. Workload-aware scheduling understands coordination between agents and breaks the circular waits that trap standard FIFO placement. Sandboxing gives each agent operating-system-level or VM-level isolation, a boundary that holds even when the agent runs code it did not write. Around compute sit the operational requirements: least-privilege containers,

scoped secrets, and the MELT observability stack of metrics, events, logs, and traces, so the decisions an agent makes stay auditable instead of opaque.

MCP as the protocol layer

Without a shared protocol, every agent integrates every tool on its own terms. That is N times M point-to-point integrations, and the count grows with both agents and tools. The Model Context Protocol collapses it to N plus M. Each agent speaks MCP once, each tool exposes an MCP server once, and any agent can reach any server. MCP carries three primitives: tools that produce side effects, resources that expose data, and prompts that shape interaction. GitHub Copilot in agent mode consumes MCP servers the same way, which means the integration you build for the platform is the integration Copilot uses.

The architectural value is the boundary, not the wire format. An agent does not touch a database or an external API directly. It speaks MCP to a server that mediates access, and a platform-owned MCP gateway centralizes authentication, scope, and telemetry in front of those servers. Policy, rate limits, and content classification live at the gateway, enforced without changing agent code. MCP is deliberately narrow. It connects agents to data, tools, and services, and it does not try to own observability, workload identity, or access control. That restraint is what keeps it composable with OpenTelemetry for traces and SPIFFE for identity. In early 2026 the protocol governed more than 10,000 active servers with 97 million monthly SDK downloads, and it moved under the Agentic AI Foundation at the Linux Foundation, which passed 170 member organizations by March 2026.

Golden paths and identity

Platform engineering is now table stakes. The DORA 2025 assessment reported that roughly 90% of enterprises already operate an internal platform, and Backstage, the open-source internal developer platform, holds close to 89% share among adopters. The reason is governance. Without golden paths, every team rebuilds the same controls, and every agent ships with its own authentication, telemetry, and secret handling. With golden paths delivered as Backstage scaffolds, IDP templates, and

MCP server registries, security and cost controls are compiled in. Teams are compliant by default and opt out only with a stated reason.

A .github/copilot-instructions.md file is a golden path in the same sense. It encodes the conventions Copilot should follow so the agent starts inside the guardrails instead of guessing at them. The four pillars of platform control, golden paths, guardrails, safety nets, and manual review gates, apply to agents exactly as they apply to people. Agents become first-class platform citizens with their own RBAC, their own resource quotas, and their own cost budgets.

Identity and policy as code

Identity is the other foundation. Agents need SPIFFE workload identities, not shared service accounts. Each agent carries a SPIFFE ID that propagates from the caller to the MCP server to the data source, and tokens are minted per session with narrow scope and short TTL, so an agent can never exceed what the requesting user could do. Policy as code closes the loop. Rules written for OPA or Cedar live in repositories, version alongside the code they govern, and pass peer review like any other infrastructure artifact. The audit log a reviewer needs is queryable without engineering help.

The four-plane architecture and supply-chain security

Cognitive Platform Engineering gives the stack a reference shape. It organizes the platform into four planes built on Kubernetes, Terraform, and OPA. A data plane runs agent workloads and MCP servers on GPU-backed pods. A control plane handles orchestration through the Kubernetes API, GitOps, and Argo. A cognition plane carries routing, policy, and intent artifacts such as Spec Kit specifications and the project CONSTITUTION. An observability layer collects spans, logs, and cost data through OpenTelemetry.

The planes close a sense-reason-act loop. The platform senses workload patterns, reasons about configuration, and acts through the control plane, so an agent that shifts from planning to code generation is met with resources before a developer files a ticket.

Securing the agentic supply chain

The security surface is wider than application security. An analysis of 295 GitHub Security Advisories that reference LLM components found supply-chain vulnerabilities in 44% of them, ahead of excessive agency at 20% and prompt injection at 18%. The controls are known and cheap. Combine at least two GitHub Actions scanners, since no single scanner covers every weakness class, and the median scan costs 2.71 seconds per workflow, so there is no performance excuse to skip it. Generate SLSA provenance and an SBOM in the pipeline, sign artifacts, and treat every agent-generated change as untrusted until policy clears it. Billing reinforces the discipline. Since June 1, 2026, GitHub Copilot charges in GitHub AI Credits at one credit per US$0.01, metered on the input, output, and cached tokens each model consumes, so a wasteful agent shows up as a line item and a well-scoped one does not.

> DEFINITION
>
> Layers one and two in one sentence: Kubernetes gives agents a runtime built for long-lived, stateful sessions, and platform engineering gives that runtime its rules. Substrate before autonomy, protocol over point-to-point, identity per session, policy as code, and supply-chain checks in every pipeline.

References

Primary sources for the cloud foundation, the protocol layer, golden paths, and supply-chain security across the agentic stack.

- CNCF, Platforms Whitepaper, the reference definition of internal platforms and golden paths.
- Cognitive Platform Engineering: 4-Plane Reference Architecture, the four-plane model on Kubernetes, Terraform, and OPA.
- Model Context Protocol: Design Patterns and Security Considerations, protocol structure and the platform-owned gateway pattern.
- LLM-Enabled OSS Vulnerabilities in GitHub Security Advisories, the supply-chain share of agentic-stack advisories, 44% of 295.

- Unpacking Security Scanners for GitHub Actions, why two scanners beat one, at 2.71 seconds median per workflow.
- GitHub Docs, Usage-based billing for GitHub Copilot: GitHub AI Credits, how token consumption is metered since June 1, 2026.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

02 THE COGNITION LAYER

Context engineering: what the agent knows.

Context engineering is the discipline of assembling the smallest set of high-signal tokens that maximizes the quality of an agent's output. It is not filling the window with everything that might help. It is the opposite: curation done with surgery. In GitHub Copilot agent mode, the context an agent reads decides both the answer and the bill. This chapter covers the utility curve, the memory tiers, six compounding techniques, and prompt caching.

The concave utility curve: more context is not better

The common intuition is that more context always helps the model. More information, better response. That intuition is wrong. The relationship between the amount of context and the quality of the output is concave with a peak, not monotonically increasing. Adding irrelevant tokens has a neutral effect at first, then a negative one. This is context rot. Three mechanisms drive it. Signal dilution comes first: for a question about one module, injecting fifty files from another module forces the model to filter noise before it can focus. The attention window is finite: even models with a one million token window apply an attention gradient, so tokens in the middle receive less attention than tokens at the start and the end, and dilution pushes relevant material into the low-attention zone. Cost rises without benefit: more tokens always cost more money, and if quality does not increase, the spend is pure waste.

The rule to apply before adding anything

Anthropic states the rule plainly: use the smallest set of high-signal tokens that maximizes the probability of the desired outcome. Not all possible context. Not the context that looked relevant at first glance. The minimum that sustains the correct result. Before injecting context, ask three questions. Is this information necessary for this specific task? Can it be retrieved on demand instead of injected in advance? Am

I adding it because it helps, or because it feels like a guarantee? The guarantee answer is usually the signal that context rot is being introduced.

Memory tiers and MCP: guides the agent reads first

Context lives in two tiers, and confusing them is expensive. The context window is the working memory of a single request. It is volatile, costly, and finite, from two hundred thousand to one million tokens depending on the model, and every turn rereads all of it. Persistent memory is information stored outside the context, in a vector store, a key-value store, or a database, and retrieved on demand. It accumulates indefinitely without inflating the window, storage costs cents per gigabyte, and retrieval costs only when consulted. The test question decides where a fact belongs: is this in active use in this interaction, or is it potentially useful reference? Active use goes in the context window. Potentially useful reference goes in persistent memory. Retrieval runs over the Model Context Protocol (MCP), the standard interface an agent uses to reach a store, a tool, or a resource without a bespoke integration for each one.

Instruction files are human-curated context

The concrete form of curated context in the developer flow is the instruction file. AGENTS.md at the repository root and .github/copilot-instructions.md are read automatically at the start of a session, before any task. A study of AGENTS.md measured the effect: a human-curated file reduces runtime by 28.64% and token usage by 16.58%. An LLM-generated file performs 3% worse than no file at all. Human curation is mandatory. The file .github/copilot-instructions.md is the GitHub Copilot form, read by Copilot Chat and the Copilot coding agent and rendered in the VS Code interface. Keep these files lean, fifty to two hundred lines, and update them in pull requests when conventions change. They encode where things live, the conventions the codebase follows, and what the agent must never touch.

Six compounding techniques that cut recurring token cost

Six techniques compound on top of curation, each measured. Compaction is the model compressing its own session history into a structured summary that keeps decisions, files touched, and errors while discarding failed attempts and filler. Measured on a coding benchmark, it delivered a 22.7% average token reduction, up to 57% in very long sessions, with 0% accuracy loss. Tool-result clearing addresses the largest token consumer in long sessions: a single file read can inject thirty thousand tokens, and old tool results stay in context and are paid for on every later call unless cleared. Keep the last five to ten results and replace the rest with a short marker.

E-mem subagents isolate context by fan-out: a lead dispatches subagents that each explore a slice and return only a summary, which reached the same task quality with 70% fewer tokens. Persistent memory moves stable facts out of the window; one study distilled 4,182 conversations into a compact layer with 11x compression while keeping retrieval intact. SkillReducer trims skill files programmatically, a 39% reduction in body and a 48% reduction in description. The sixth technique, prompt caching, is the largest single lever and gets its own section. These techniques stack: compaction shrinks history, clearing removes stale results, subagents isolate exploration, persistent memory offloads reference, and SkillReducer thins the definitions the agent always reads.

DEFINITION

Six techniques that compound: compaction (22.7% average token reduction, 0% accuracy loss), tool-result clearing (keep the last five to ten results), E-mem subagents (70% fewer tokens), persistent memory (11x compression on retrieval), SkillReducer (39% smaller bodies, 48% smaller descriptions), and prompt caching (up to 90% input savings). Curate the smallest high-signal set, then reuse it.

Prompt caching: reuse the KV cache from a stable prefix

Every model call runs the attention mechanism over all tokens, which is the most expensive work of the request, and stores the intermediate result, the attention KV cache, in GPU memory. On a later request with the same token prefix, the provider retrieves that KV cache instead of recomputing it. The model skips the work of processing those tokens again. In conversational workloads this reduces input cost by up to 90% without changing a single line of prompt. The cache operates on a prefix hash, byte by byte, so any change to any byte invalidates the cache from that point forward.

The discipline is one rule: stable content first, dynamic content later. Put the system prompt, the tool definitions, and static context at the front and mark them cacheable; keep the user message and anything that changes at the end. A changed system prompt invalidates the whole prefix, so version it deliberately. Under GitHub AI Credits, cached tokens are billed at each model's published rate, where one credit equals one cent, so a stable prefix is not only faster, it is a direct line item saved on every repeated call. When routing across the Copilot picker, the same prefix discipline holds whether the call goes to Haiku 4.5 for a trivial step, Sonnet 5 for code, or Opus 4.8 for a plan.

References

Primary sources for the utility curve, the instruction-file evidence, the retrieval standard, and the billing model.

- Anthropic, Effective context engineering for AI agents, the source of the rule: the smallest set of high-signal tokens that maximizes the probability of the desired outcome.
- GitHub Docs, Customizing GitHub Copilot with instruction files, how .github/copilot-instructions.md is read and applied in agent mode.
- Spec Kit, toolkit for Spec-Driven Development, the pattern for human-curated specification artifacts that ground custom agents.

- GitHub Docs, Usage-based billing and GitHub AI Credits, how cached and uncached tokens are metered, where one AI Credit equals one cent.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

03 THE STRATEGIC LAYER

Intent engineering: what the agent should do.

Context engineering makes an agent accurate. It does not make the agent aim at the right target. That is the gap intent engineering closes. A GitHub Copilot agent in agent mode can hold the whole repository in context and still optimize for an outcome the organization never chose. This chapter is the strategic layer of the stack. It shows how to encode what the agent should do as versioned artifacts: CONSTITUTION.md for values, SPECIFICATION.md for testable requirements, and IMPLEMENTATION_PLAN.md for scope. Then it shows why the interface between planning and coding is where most multi-agent failures begin.

Intent debt: full context, no target, still the wrong outcome

I keep coming back to the Klarna case from Q3 2025 because it shows exactly what breaks. Klarna deployed an agent with everything you would want from an infrastructure perspective: detailed customer data, product specifications, pricing rules, execution history. The context was comprehensive. Within weeks, the agent was optimizing for metrics the organization never wanted to prioritize. By the time anyone caught the misalignment, the system had already incurred operational and reputational cost. The lesson is direct: context engineering cannot prevent an agent from optimizing for the wrong thing. You can give an agent all the context in the world, but if you never tell it what to optimize for, it will optimize for something.

Intent debt accumulates silently. The objectives, the constraints, and the reasoning behind them live in a Slack thread from eight months ago, an email buried in a support channel, the head of one engineer. Each is a point of failure in a system meant to run autonomously at scale. Every generation of AI-assisted work without explicit intent artifacts widens the gap between what the system was meant to do and what it does. The topic article names three debts that compound together:

technical, cognitive, and intent. Intent debt is the most insidious because the agent is not broken. It was never fully specified.

> DEFINITION
>
> A well-contextualized agent with no written intent is an arrow with excellent sighting and no target. Context tells the agent how the world works. Intent tells it what to aim at.

The Spec Kit artifacts: values, requirements, scope

Intent engineering materializes as three artifacts, and Spec Kit makes them legible. CONSTITUTION.md is a human-first document of principles, guardrails, and non-negotiable organizational values. It is written for humans to read and structured for machines to enforce. A constitution states a hierarchy of what matters most when trade-offs arise, for example that cost optimization never takes precedence over data privacy. It is not a checklist of every decision. In a GitHub Copilot repository, those same principles surface to agent mode through the .github/copilot-instructions.md file, so the values travel with the code.

SPECIFICATION.md encodes requirements in a machine-parseable, testable form. It translates a constitutional principle into a condition the agent can verify before it acts. If the constitution says preserve transparency, the specification says: when a system limitation is discovered, escalate to a human reviewer before proceeding. IMPLEMENTATION_PLAN.md breaks the work into atomic tasks and marks which can run in parallel. The plan becomes a control boundary. The agent knows exactly what it may modify, and anything outside the plan is out of scope by default.

> DEFINITION
>
> Three files, three jobs. CONSTITUTION.md fixes the values. SPECIFICATION.md fixes the testable requirements. IMPLEMENTATION_PLAN.md fixes the scope. Together they are the backbone of Spec-Driven Development.

Spec-Driven Development: the gate, the hooks, the defect drop

Spec-Driven Development organizes agent work into phases, with a hard control between them. First, a human and a planner draft CONSTITUTION.md and SPECIFICATION.md in Ask mode, with no file writes. Then they refine the specification into an IMPLEMENTATION_PLAN.md. Then comes the human approval gate. The agent cannot enter agent mode, where it holds write permissions, until a human signs off on the plan. This is not a soft boundary. It is enforced by the orchestration layer. The gate moves human judgment upstream, into the specification, where a misalignment is still cheap to fix.

Hooks: enforcement at zero AI Credits

Hooks enforce the plan at zero token cost. A preToolUse hook is a shell command that intercepts a write and blocks it when the target file is not in IMPLEMENTATION_PLAN.md. It runs in the orchestration layer, not in the model, so it consumes no GitHub AI Credits. The agent proposes, the hook enforces, the human decides whether to amend the plan. Research on constitutional SDD found that it reduces security defects by 73% compared with unconstrained AI-assisted development. The mechanism is not a perfect specification. It is that explicit review of the plan before implementation surfaces the misalignment while it is still fixable. Route the planning phase to a stronger model in the Copilot picker, such as Opus 4.8 or Claude Fable 5, and the scoped implementation to Sonnet 5.

> DEFINITION
>
> The human gate between specification and implementation is the most important control in the workflow. Hooks make it cheap to keep: a shell conditional in the orchestration layer, zero model tokens, zero AI Credits.

The planner-coder gap: the specification is the interface

A study of multi-agent system failures found that 75.3% of them originate in the gap between the planner and the coder. When the planner writes an ambiguous or incomplete specification, the coder generates insecure or incorrect code. This is not only a quality problem. It is a security vector. An attacker who can influence what the planner produces, through prompt injection or context manipulation, can make the coder generate vulnerable code without ever touching the coder.

This is the strongest empirical case for treating SPECIFICATION.md as real engineering, not paperwork. The specification is the interface between planning and coding. When that interface is explicit, machine-parseable, and human-reviewed, you close the gap where three-quarters of failures begin. When it is implicit, a vague task or a casual prompt, you leave that failure surface open. The same evidence base shows the model is a larger driver of outcome than the framework, so invest in the specification and in model selection in the Copilot picker before you invest in framework choice. Intent engineering is the strategic layer. Harness engineering, the next chapter, is the operating model that runs it.

> DEFINITION
>
> 75.3% of multi-agent failures start at the planner-coder interface. The specification is that interface. Make it explicit, machine-parseable, and human-reviewed, and you close three-quarters of the failure surface.

References

Primary sources for intent debt, the Spec Kit artifacts, Spec-Driven Development, and the empirical evidence on cost, complexity, and the specification interface.

- GitHub, Spec Kit: a toolkit for Spec-Driven Development, the toolkit that makes CONSTITUTION.md, SPECIFICATION.md, and IMPLEMENTATION_PLAN.md concrete, version-controlled artifacts.

- GitHub Docs, Usage-based billing for individuals: GitHub AI Credits, how agent work is metered under GitHub AI Credits, one credit equal to one US cent since June 1, 2026, and why orchestration-layer hooks draw zero credits.
- arXiv, Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild, large-scale evidence that AI-generated code shifts debt toward requirements and tests, the raw material of intent debt.
- arXiv, AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development, the measured impact of coding agents, including the cognitive complexity that ungoverned agents add to engineering organizations.
- Deloitte, State of AI in the Enterprise 2026, the transformation gap that intent debt widens: three in four enterprises plan agentic AI, roughly a third report deep transformation.
- Anthropic, Effective context engineering for AI agents, the discipline intent engineering sits on top of, context makes the agent accurate, intent tells it what to aim at.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

04 EXECUTION AND OPERATIONS

Harness Engineering: the operating model.

An agent is a model plus a harness. The model reasons. The harness is everything around it: tools, memory, state, gates, sensors, and guides. Swap the model in the GitHub Copilot picker and the agent still behaves the way its harness lets it. This chapter turns the stack into an operating model: the guides and sensors taxonomy, four production patterns, seven implementation steps, and two accelerators on one Microsoft foundation.

Agent equals model plus harness

The formula that named the field in early 2026 is compact: an agent is a model plus a harness. The model provides reasoning capability. The harness is everything else, the tools, memory, state, context, validation, security, lifecycle, gates, sensors, and guides. In GitHub Copilot the model is the pluggable part. You choose it in the picker, Claude Fable 5, Opus 4.8, Sonnet 5, Haiku 4.5, Gemini 3.1 Pro, or GPT-5.x, and the harness stays constant. Beren Millidge framed it in 2023: a raw model is a CPU with no operating system, and the harness is the OS that makes it useful.

Martin Fowler and Birgitta Böckeler gave the harness its clearest taxonomy in March 2026. A harness divides into guides, which steer the agent, and sensors, which verify it. Guides are the files the agent reads first: .github/copilot-instructions.md, AGENTS.md, versioned system prompts, architectural constraint documents, and rich tool descriptions. Sensors observe what the agent did. Deterministic sensors are cheap and fast: custom linters, type checkers, pre-commit hooks, structural tests, and CI gates that block a merge when any check fails. Inferential sensors are LLM-based: code review as a judge and semantic pull request analysis. A study of AGENTS.md files found that human-curated files cut runtime by 28.64% and token usage by 16.58%, while LLM-generated files performed 3% worse than no file at all. The discipline is curation, not file size.

> DEFINITION
>
> Agent equals model plus harness. The model is the pluggable component you select in the GitHub Copilot picker. The harness is what your team builds and owns: guides that steer the agent and sensors that verify it. Build an agent and what you built is a harness pointed at a model.

Production patterns

Four patterns recur in production. The ReAct loop is the simplest: the agent reasons, picks a tool, observes the result, and iterates until it hits the goal or exhausts its budget. Use it for short tasks with a clear success check: issue triage, summarization, report generation. The initializer plus coding agent pattern handles work that spans many sessions. An initializer runs once and creates the structure: an init script, a progress file as the session-to-session log, a first commit, and a feature list of trackable items. A coding agent then runs on each later session: it reads progress, picks the next feature, implements, tests with Playwright, updates the log, and commits. State lives in versioned files, so long-term memory becomes legible and auditable.

The Planner-Generator-Evaluator pattern, from Anthropic in March 2026, splits work across three roles. The Planner expands a short prompt into a specification and deliberately leaves implementation details open. The Generator implements in sprints and first signs a sprint contract with the Evaluator: a shared definition of done. The Evaluator drives the application through Playwright like a real user and fails the sprint if anything breaks. Research found that 75.3% of multi-agent failures originate at the planner-to-coder interface, and the Evaluator makes it explicit. The fourth pattern is continuous garbage collection. Background agents scan the repository on a cadence for cross-layer imports, duplication, and drift, then open small refactor pull requests that review in under a minute and auto-merge. OpenAI treats technical debt as a high-interest loan, paid down continuously rather than in painful bursts.

The seven-step playbook

The seven steps are sequential and cumulative. First, choose the right use case. A good one has a verifiable success criterion, repetitive work that justifies the investment, material impact, and containable risk. Vague criteria, one-off work, and unbounded risk are where pilots die. Second, write the instruction file as a map, not an encyclopedia. For GitHub Copilot that is .github/copilot-instructions.md. Keep it near 100 lines and let it point to deeper sources in a structured docs directory. Third, instrument trajectories from day one: the full record of a session, prompts, tool calls, outputs, errors, and timings. Without it you cannot debug, and every captured session becomes evaluation data. Fourth, plant deterministic sensors before any LLM judge. A custom linter for architectural rules, a type checker on every pull request, pre-commit hooks, and a mandatory CI gate catch most failures cheaply.

Fifth, enforce architecture mechanically. Encode dependency direction as a linter rather than a paragraph. On the Microsoft stack, pair Bicep with Azure Policy, Roslyn analyzers, and branch protection in GitHub. This step separates pilots from production. Sixth, add self-verification loops: sprint contracts, test-first orchestration, Playwright browser automation, and loop detection middleware that aborts on repeated actions. LangChain moved from rank 30 to rank 5 on Terminal Bench 2.0 by adding sensors and self-verification with the model held constant. The missing piece is often a sensor, not a better model. Seventh, run the continuous improvement loop. Observe a failure in a trajectory, classify the root cause, have the agent write the fix under review, add a regression test, and audit future trajectories. Every failure becomes structurally impossible to repeat. Three engineers using this discipline shipped roughly a million lines of production code with no human-written code.

Two paths on one Microsoft foundation

The foundation under every engagement is constant. Azure is the cloud, GitHub Enterprise carries source control and DevOps, Entra Agent ID gives each agent a managed identity with scopes and an audit trail, and Azure AI Foundry with Microsoft Agent Framework is the AI platform. GitHub Copilot ships the harness primitives. The .github/copilot-instructions.md file is the repository guide; path-scoped .instructions.md and reusable .prompt.md files extend it. Custom chat modes hold

team personas, agentic workflows carry loops and gates, and MCP server configuration exposes tools. The GitHub Copilot coding agent handles multi-session work, and code review automation acts as an inferential sensor in pull requests. Billing runs on GitHub AI Credits since June 1 2026: one credit equals one US cent, and input, output, and cached tokens are metered at each model published rate, so cost tracks work performed rather than seats.

Two accelerators materialize the same harness. Three Horizons uses Red Hat Developer Hub on Azure Red Hat OpenShift for clients with prior Red Hat investment or strong support requirements. Open Horizons uses Backstage on Azure Kubernetes Service for Azure-native clients. Both implement the same discipline and reach the same Horizon 3 destination; only the internal developer platform and the cluster change. The failure-mode-to-harness mapping makes incident review mechanical: cross-layer imports point to a dependency-direction linter, premature completion to a sprint contract and a separate evaluator, an agent loop to loop detection middleware, a leaked secret to a pre-commit hook and a sandbox without secret access, and a destructive command to an approval-required policy. None of these is solved by a better prompt. Each needs a structural piece of the harness, and the agent mode primitives in GitHub Copilot supply most of them out of the box.

References

Primary sources for the harness formula, the guides and sensors taxonomy, the four production patterns, and the seven-step operating model.

- Mitchell Hashimoto, My AI Adoption Journey, origin of the discipline: engineer the harness so the agent never repeats a mistake.
- Ryan Lopopolo, OpenAI, Harness engineering in an agent-first world, the seven-step playbook, continuous garbage collection, and roughly a million lines of code with no human-written code.
- LangChain, Improving Deep Agents with harness engineering, rank 30 to rank 5 on Terminal Bench 2.0 by changing the harness with the model held constant.
- Birgitta Böckeler, Harness engineering for coding agent users, martinfowler.com, the guides and sensors taxonomy this chapter builds on.

- Anthropic, Harness design for long-running application development, the Planner, Generator, and Evaluator pattern for multi-hour autonomous sessions.
- Microsoft Agent Framework 1.0, the managed Agent Harness layer on the shared Azure and GitHub foundation.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 FUNDING AND CLOSE

Token economy, FinOps, and adoption: fund the platform, retire the cemetery.

The platform is built. The bill still arrives. On June 1, 2026, GitHub Copilot moved from charging per request to charging for the tokens each model actually consumes, so model choice, caching, and routing become line items a finance team can read. This chapter funds the platform: how to route models in the Copilot picker, how to run FinOps for AI, how to cut 25 to 45 percent over baseline, and how to adopt at enterprise scale.

Model routing in the Copilot picker: pay each model at its API rate

The Copilot picker exposes one model per interaction, and agent mode reads that choice on every task. Route by complexity. Haiku 4.5 handles trivia: classification, extraction, and low-nuance batch work. Sonnet 5 is the workhorse for code: debugging, multi-file refactors, and deep review. Opus 4.8 is reserved for plans: architecture and decisions that will be reused. Every premium call bills input, output, and cached tokens at the published API rate of that model, converted at 1 AI Credit equal to US$0.01. Bundled models included in the plan burn no credits.

Routing is the first-order cost lever, and it belongs in version control, not in each developer head. Set the default to Sonnet 5 for agent mode and edit mode, pin Opus 4.8 to plan mode, and drop to Haiku 4.5 for well-defined batch tasks. Commit that policy in .vscode/settings.json and state the rationale in .github/copilot-instructions.md so the whole team runs the same routing. The premium tier costs several times the workhorse for the same session, so a default of Opus 4.8 is money spent without a matching gain in quality.

FinOps for AI: Inform, Optimize, Operate

FinOps for AI is the discipline of instrumenting, measuring, optimizing, and operating token spend with the rigor applied to any other cloud resource. It runs in three phases. Inform captures per-request telemetry, model, tokens, user, and feature, through OpenTelemetry, exports it under the FOCUS schema, and surfaces it in Power BI or Grafana dashboards. Without data, optimization is guessing. Start with showback: show each team the cost it generated without charging it, because behavior changes once the number is visible.

Optimize identifies the specific waste: inefficient prompts, oversized models, missing cache, untrained teams. Each fix has an estimable payback that is measurable and attributable. Operate closes the loop with chargeback to teams, anomaly detection that flags a session running three sigma over the mean, and quarterly budget review. Maturity is measured in months, not days. Establish the baseline in month one, capture quick wins by month three, and institutionalize the improvement cycle by month six, where showback graduates to chargeback.

Eight optimization patterns, 25 to 45 percent over baseline

The numbers make the case. For a 500-developer organization at US$19 per seat, a baseline of US$9,500 per month, pilots without discipline saw 50 to 100 percent overage after the PRU transition. With discipline applied, the same organization reaches 25 to 45 percent reduction over baseline within 60 to 90 days, compounding to 40 to 70 percent on agentic workloads. For organizations above 5,000 developers, annual savings exceed US$1 million. The payback period is under 60 days.

Eight patterns deliver that reduction: dynamic routing by complexity, hierarchical caching at three levels, subagent fan-out with an E-mem hierarchy, plan-then-execute with a cheaper planning model, semantic cache using embedding similarity, tool composition over catalog growth, lean MCP with progressive disclosure of schemas, and Spec-Kit driven custom agent SDD. Five anti-patterns cancel the gains and must go: the largest model by default, an agent where a workflow suffices, the whole repository pasted into context, skipping prompt caching, and skipping the harness.

Azure AI Foundry and migration: PTU versus PAYG

Azure AI Foundry complements GitHub Copilot, it does not replace it. Copilot solves coding in the IDE. Foundry runs production workloads with governance, a model catalog of 1,800 plus models, content safety, RBAC, and audit inside the Azure tenant. It offers two billing models. PAYG bills per token at published rates with no commitment, which fits variable or spiky load. PTU reserves inference capacity with guaranteed throughput and consistent latency, which fits stable, high volume. The choice is per workload, not per organization.

The 30, 60, 90 day adoption roadmap

The heuristic is simple: PAYG below US$5,000 per month per model, PTU above US$30,000, and a benchmark in between. The roadmap follows the money. In the first 30 days, analyze the bill preview, set the paid-usage cap with alerts, route models in the picker, and turn on telemetry. From 30 to 60 days, apply routing and caching, curate copilot-instructions.md, and run the tech-lead workshop. From 60 to 90 days, institutionalize showback, wire the FOCUS export into dashboards, and move steady workloads onto Foundry where compliance or PTU economics justify it.

The closing argument: discipline, not talent

Seventy-five percent of enterprises plan agentic AI. Only 34 percent report deep transformation. The gap is not budget and it is not talent. It is applied discipline across the six disciplines: Cloud, Platform, Context, Intent, Harness, and Token Economy. The token economy is the discipline that keeps the other five funded, because a platform nobody can afford to run is a platform nobody runs.

Under PRUs, cost was per intent, and a four-hour agentic session cost the same as a quick question. Under tokens, cost is per work performed, and every optimization shows up on the invoice. Teams that treat tokens as a finite resource, with budgets per session, agent, and repository, pay 30 to 70 percent less for the same output. That margin is the difference between the cemetery and the platform, and it is bought with practice, not headcount.

DEFINITION

Token economy in one sentence: price every model call at the published API rate in AI Credits, route the smallest model that passes the eval, and fund the platform with the savings. Applied discipline, not talent, separates the cemetery from the platform.

References

Primary sources for the billing change, the FinOps discipline, the Azure AI Foundry options, and the adoption evidence.

- GitHub Docs, Usage-based billing and GitHub AI Credits, the billing change: 1 AI Credit equals US$0.01, with tokens billed at the published API rate of each model.
- GitHub Copilot, Available AI models, the model picker that every routing decision targets.
- FinOps Foundation, FinOps Framework, the Inform, Optimize, and Operate phases applied to AI spend.
- FOCUS, FinOps Open Cost and Usage Specification, the standard schema for unified cost export across providers.
- Microsoft Learn, Azure AI Foundry, the governed platform for production workloads and the model catalog.
- Microsoft Learn, Provisioned throughput (PTU), the reserved-capacity option weighed against pay-as-you-go.
- Deloitte, State of AI in the Enterprise, the adoption gap: most enterprises plan agentic AI, far fewer report deep transformation.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps