A FIELD GUIDE TO MODEL-AGNOSTIC AI SYSTEMS

Harness Engineering:

Designing the System Around the Model

The durable advantage in enterprise AI is shifting from the model to the system built around it: the harness.

AI-CYBERSECURITY UPDATE / 25 JUNE 2026

JUNIOR WILLIAMS

> **As foundation models converge, the durable advantage in enterprise AI is shifting from the model to the system built around it: the harness. That system is what a company owns and governs, and what lets you swap the model underneath as better or cheaper options appear.**

The risk is no longer hypothetical. In June 2026, a U.S. export-control order forced Anthropic to disable Claude Fable 5 worldwide, three days after the model launched, with no deprecation window and no migration path. Applications that called it directly broke, while teams routing through a swappable layer changed a configuration and kept running. **A model can vanish on someone else's schedule,** which is exactly why the system around it has to be yours.

Enterprise AI decisions still tend to begin with which model to use, but that rarely determines success. Rankings shift every few months, and the leading models are close enough that the choice between them isn't what separates a working agent from a failed one. What does is the system around the model: the instructions it operates under, the data it can reach, the tools it can call, the limits on what it can do, and the checks that catch its mistakes.

On its own, a language model only generates text. It can't see a company's data, act in its systems, verify its own work, or carry a decision from one session to the next. Everything that turns a model into a working agent sits outside it. As LangChain puts it, an agent is a model plus a harness: "*if you're not the model, you're the harness.*" Designing that layer deliberately is the discipline of context engineering.

The diagram below splits the system in two. One box is the model. The surrounding eight are the harness: system prompts, context loading, retrieval, third-party tools, a code sandbox, skills, subagents, and loop control. Each does distinct work, and each shapes whether the agent's output can be trusted. Build it well and a further consequence follows, one most leaders haven't accounted for: **the model underneath becomes a component you can replace.**

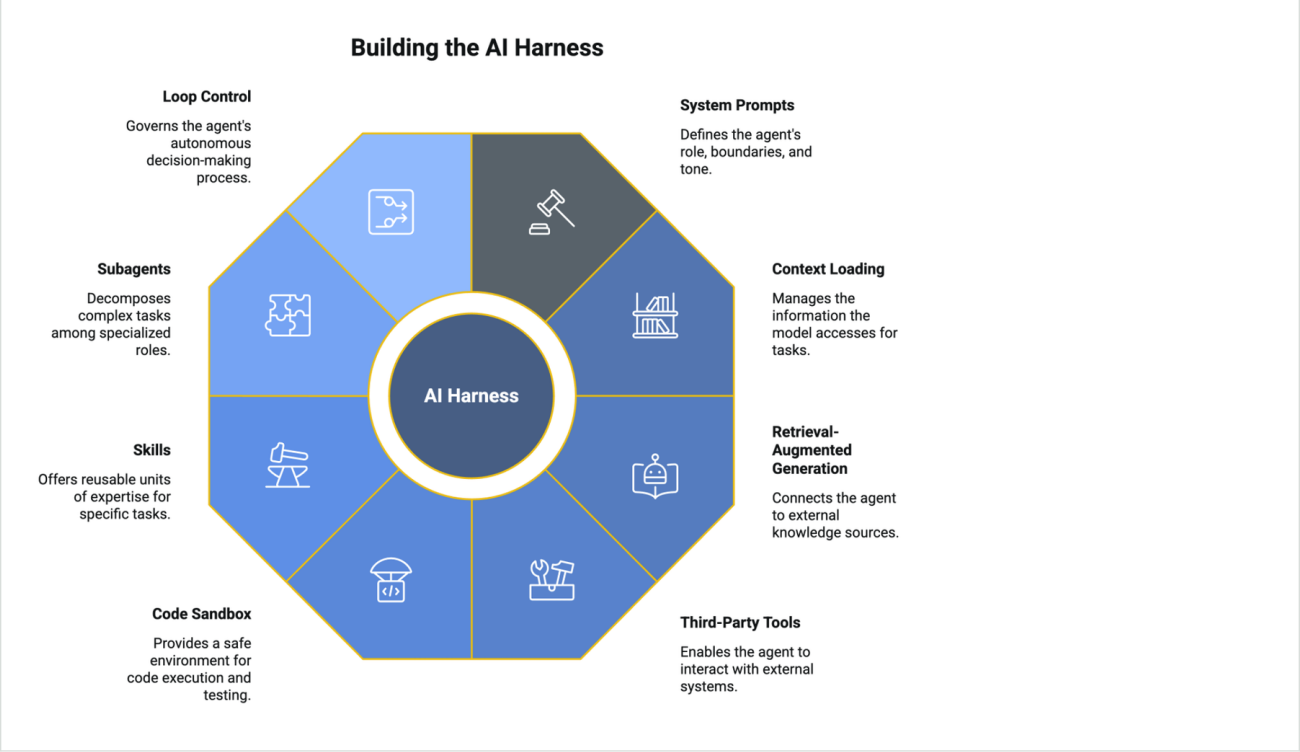

FIGURE 01 / THE HARNESS: EIGHT COMPONENTS AROUND THE MODEL.

System Prompts

The system prompt defines the agent's role, its boundaries, the actions it can't take without approval, and how to handle instructions that are ambiguous or in conflict. It's where policy moves from a document beside the workflow to a constraint operating inside it.

Left unmanaged, the prompt accumulates as an open-ended list of instructions and degrades as it grows: directives contradict one another, and no one can say which still apply. Treated as architecture, it's version-controlled, tested against known failure cases, and written so its constraints are **enforced, not just stated.** When an agent can act, the system prompt is the first place you encode risk tolerance, and *the easiest place to encode it badly.*

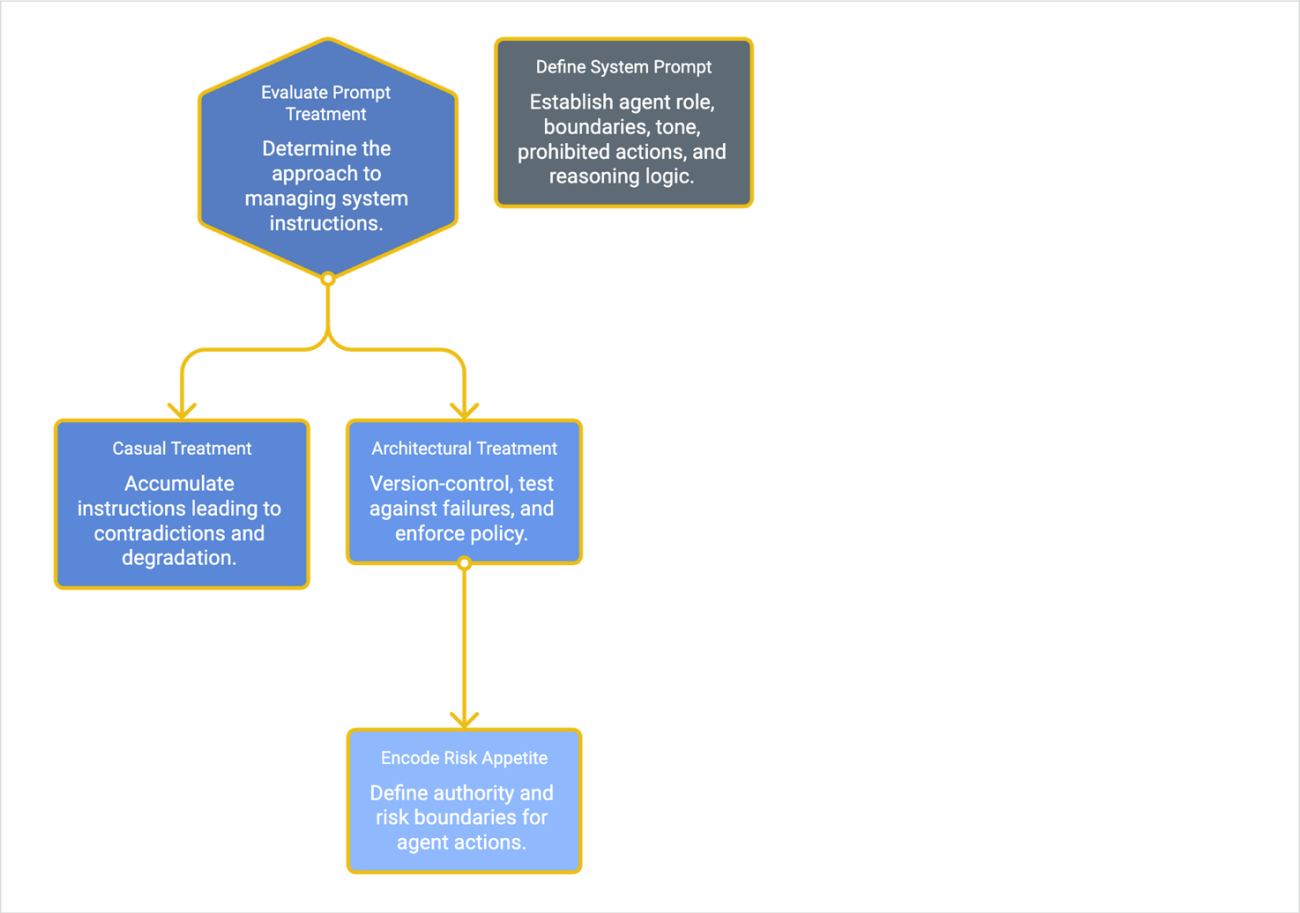

FIGURE 02 / SYSTEM PROMPTS - FROM AN ACCUMULATING INSTRUCTION LIST TO ENFORCED ARCHITECTURE.

Context Loading

A model's context window is finite, and every token in it carries a cost. Context loading is the decision about what information enters that window for a given task: in what order, and at what level of detail.

This is where many agent projects quietly fail. Too little, and the agent reasons from gaps it can't see. Too much, and the signal is buried, costs rise, and accuracy falls as the model loses track of what matters. Good context loading is disciplined curation under a fixed budget: the conventions, current task, recent state, and history the model needs, and nothing more. *The hard part is deciding what to leave out.*

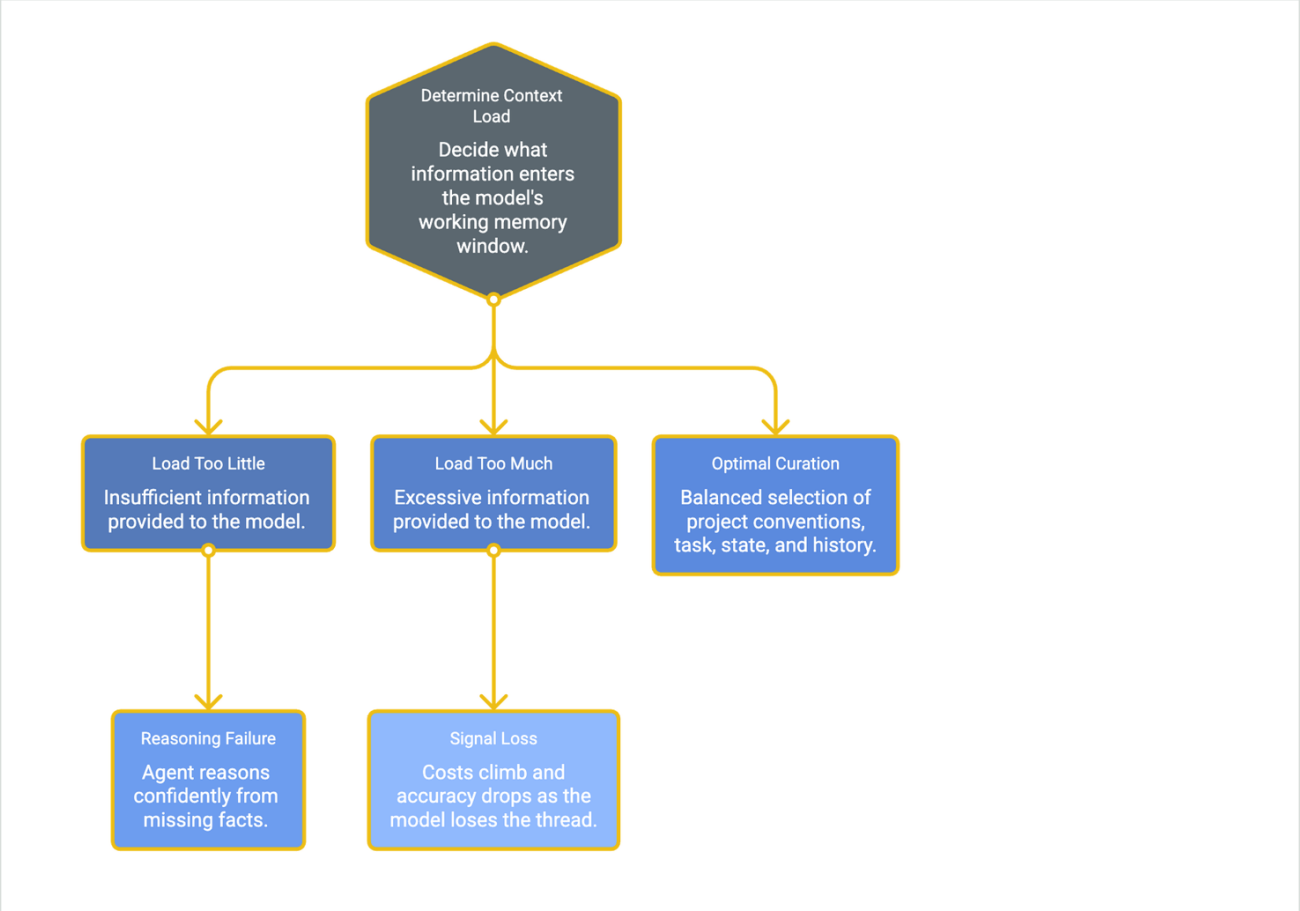

FIGURE 03 / CONTEXT LOADING - DISCIPLINED CURATION UNDER A FIXED BUDGET.

Retrieval-Augmented Generation

No context window can hold an enterprise's knowledge, and most of it changes faster than a model can be retrained. Retrieval-augmented generation closes the gap, connecting the agent to an external store (documents, policies, tickets, code, past decisions) and supplying the relevant passages as they're needed.

Retrieval is also where the agent meets a question every regulated business eventually asks: where did this answer come from? When the retrieval layer carries source metadata through to the response, the agent can attribute an answer to a specific document and date instead of just asserting it. When it doesn't, you get plausible passages with no reliable path back to a source. **A retrieval layer is working only when every answer can name its source.**

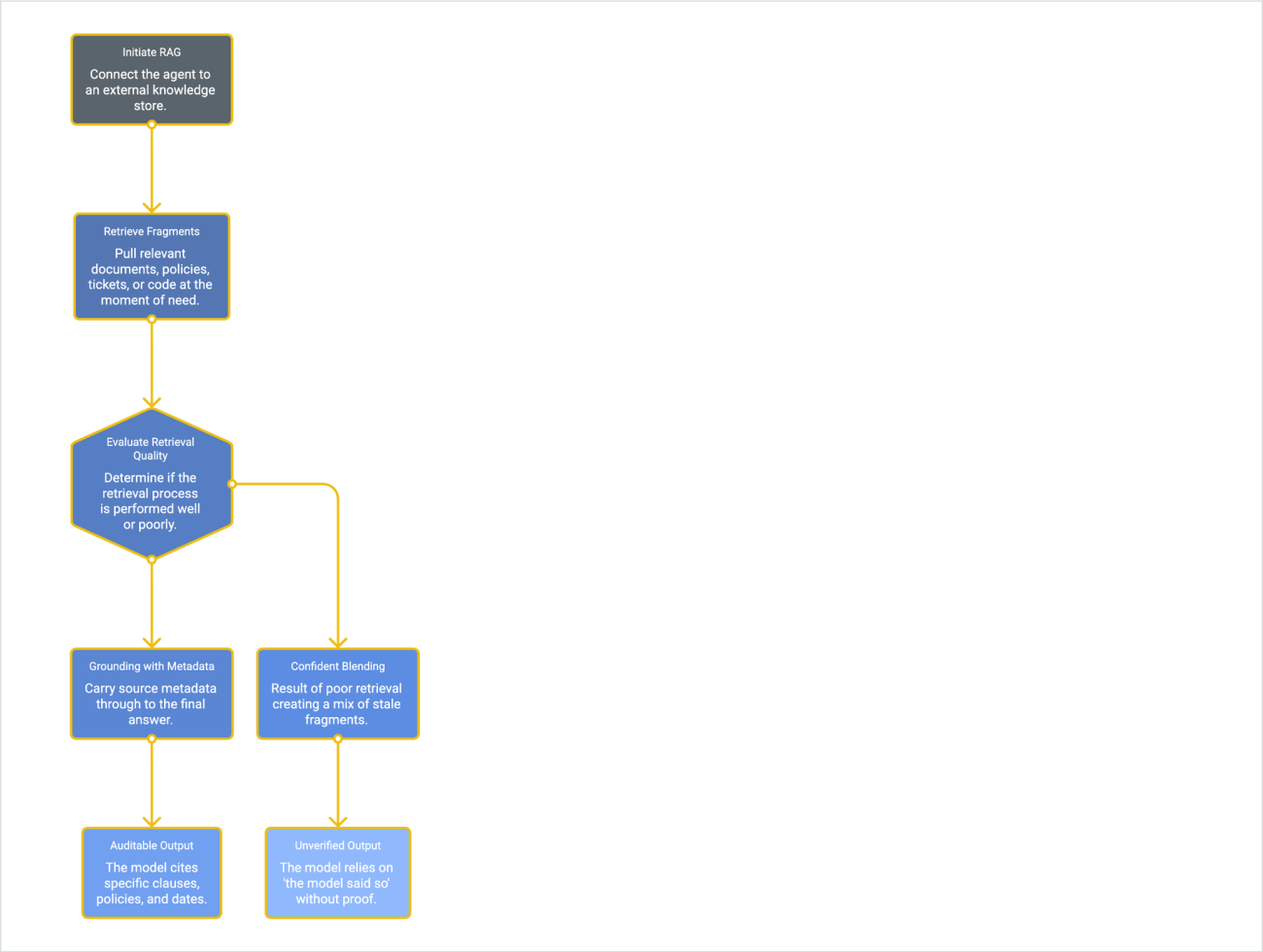

FIGURE 04 / RETRIEVAL-AUGMENTED GENERATION - GROUNDING ANSWERS IN AUDITABLE SOURCES.

Third-Party Tools

A model that can only produce text is *a chatbot*; a model that can call tools is *an operator*. Tools (APIs, databases, search, ticketing systems, internal services) are how an agent acts beyond the conversation, in the systems where work is recorded and changed.

Every tool the agent can call is also a permission you've granted. The harness governs what that access means in practice: which actions are read-only, which need human approval, which are forbidden, and what gets logged each time a tool runs. Without that engineering, a helpful assistant becomes *an unmanaged path into production.* **Each tool connection is both an attack surface and a record-keeping obligation,** and the surrounding system is where both are designed.

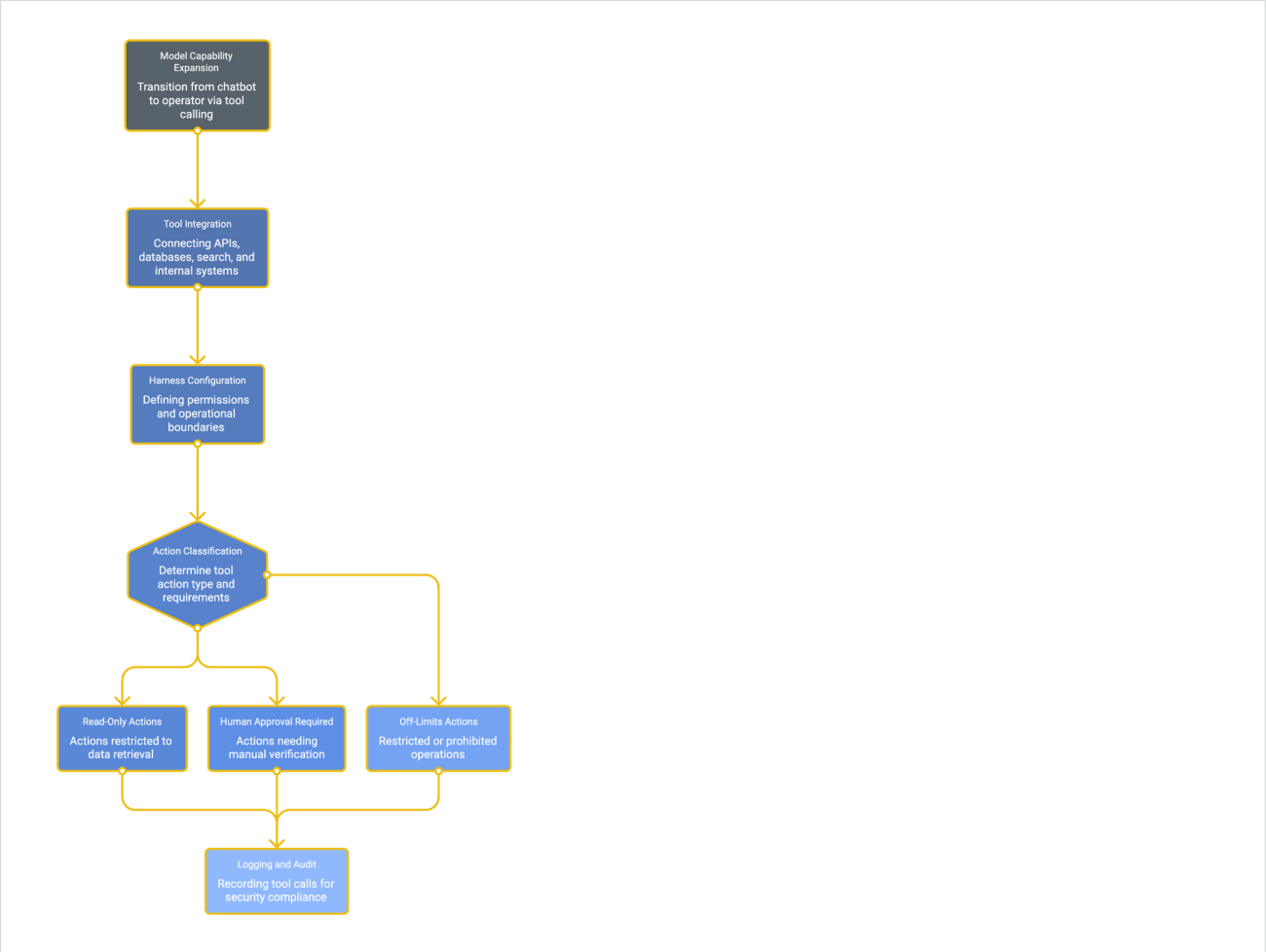

FIGURE 05 / THIRD-PARTY TOOLS - PERMISSIONS, APPROVALS, AND LOGGING.

Code Sandbox

Capable agents write and run code (to transform data, test an approach, automate a step, or check their own output), and they need somewhere to do it that doesn't touch production. The code sandbox is that isolated environment: an agent can run code, fail, and retry without effects leaking outward.

The sandbox does two things at once. It contains risk, keeping generated code away from systems it should never reach. And it produces a feedback signal: code that executes returns results the agent can observe and correct against. An agent that runs its work in a contained environment learns from what actually happens; the same capability without containment is *a direct route into systems it was never meant to change.*

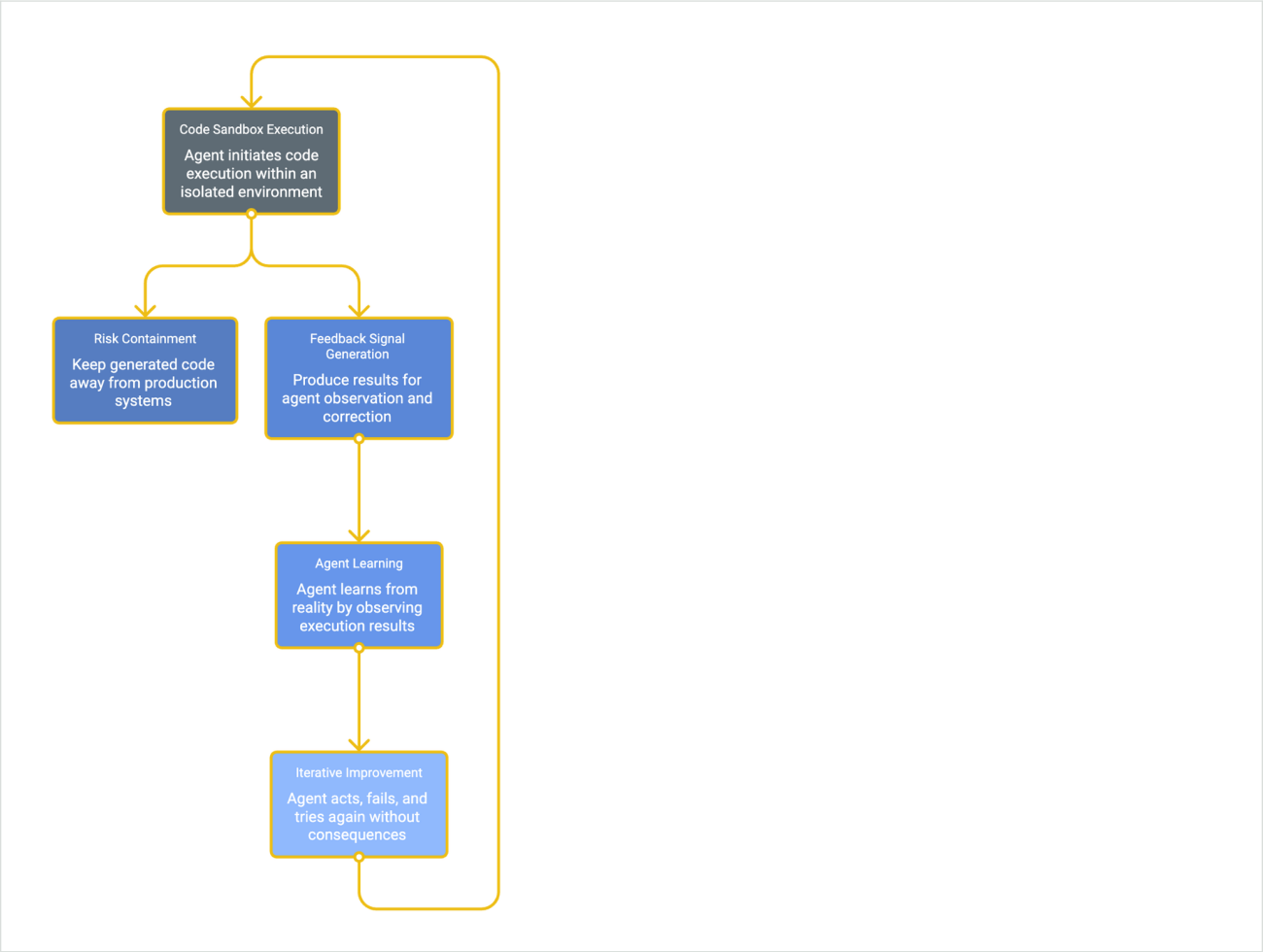

FIGURE 06 / CODE SANDBOX - CONTAINED EXECUTION THAT ALSO PRODUCES A FEEDBACK SIGNAL.

Skills

Skills are reusable units of capability (a procedure, a template, a domain-specific method) that an agent loads when a task calls for it, rather than carrying at all times. They keep the agent's working context lean while making specialized competence available on demand.

Skills also make expertise portable and governable. A procedure written once can be versioned, reviewed, and reused across many agents, turning know-how that would otherwise live inside a single prompt into an asset the organization can manage. **A reviewed, versioned skill can be audited, retired, and shown to behave consistently wherever it runs.** *The same procedure improvised in a prompt each time can be none of those things.*

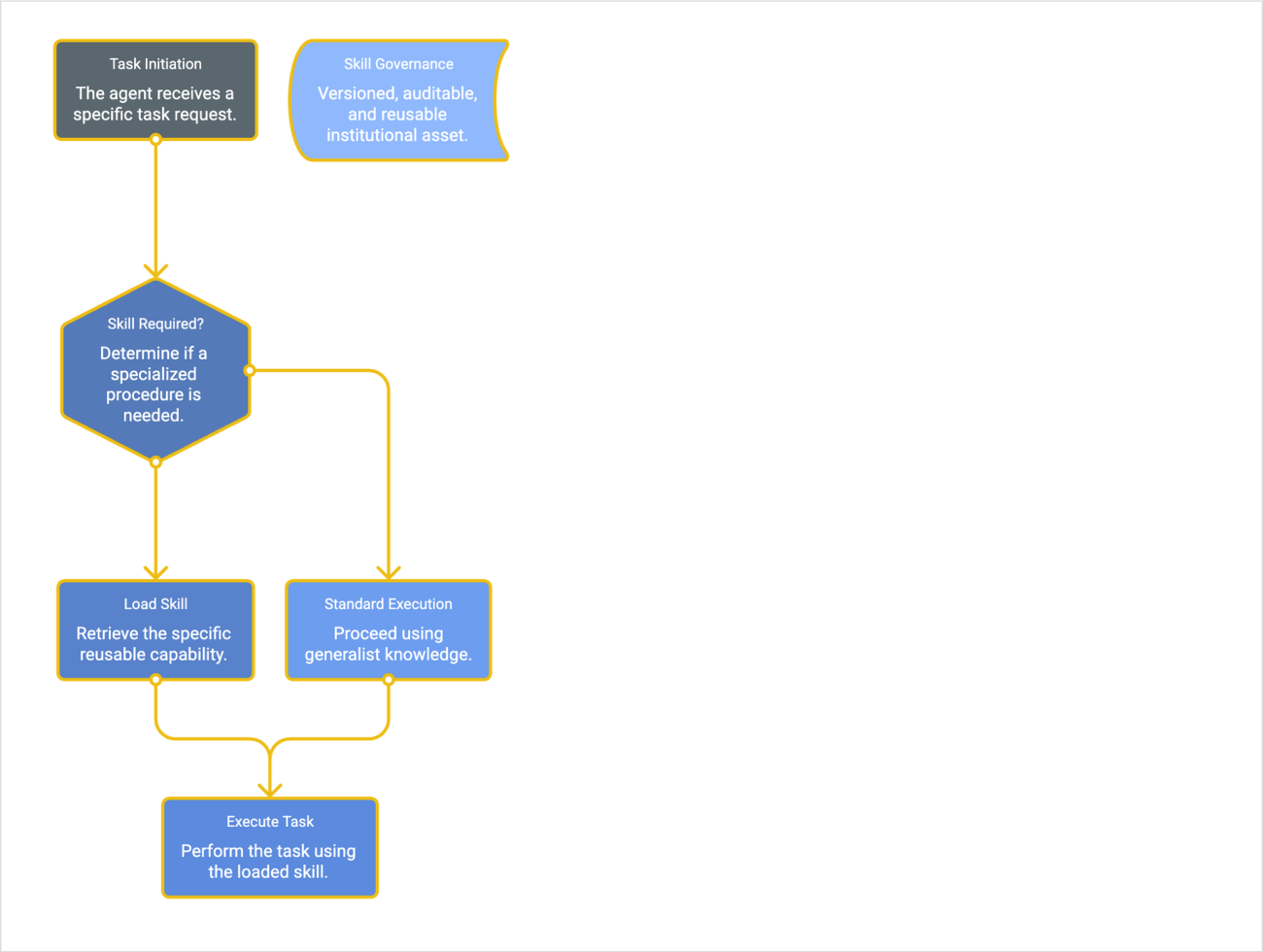

FIGURE 07 / SKILLS - REUSABLE, GOVERNABLE CAPABILITY LOADED ON DEMAND.

Subagents

Complex work resists being held in a single context. Subagents divide a task across specialized roles (one to plan, one to build, one to check), each working in its own clean context.

The most consequential version separates the agent that does the work from the agent that judges it. Anthropic's engineering team, describing a system for long-running application development, called this separation "*a strong lever*": an agent asked to assess its own output tends to respond by "*confidently praising the work* - even when, to a human observer, the quality is obviously mediocre." The approach borrows from adversarial training, where a generator produces and an evaluator pushes back, and the tension improves the result. It's the familiar principle that **the author of a change shouldn't be the only check on it.**

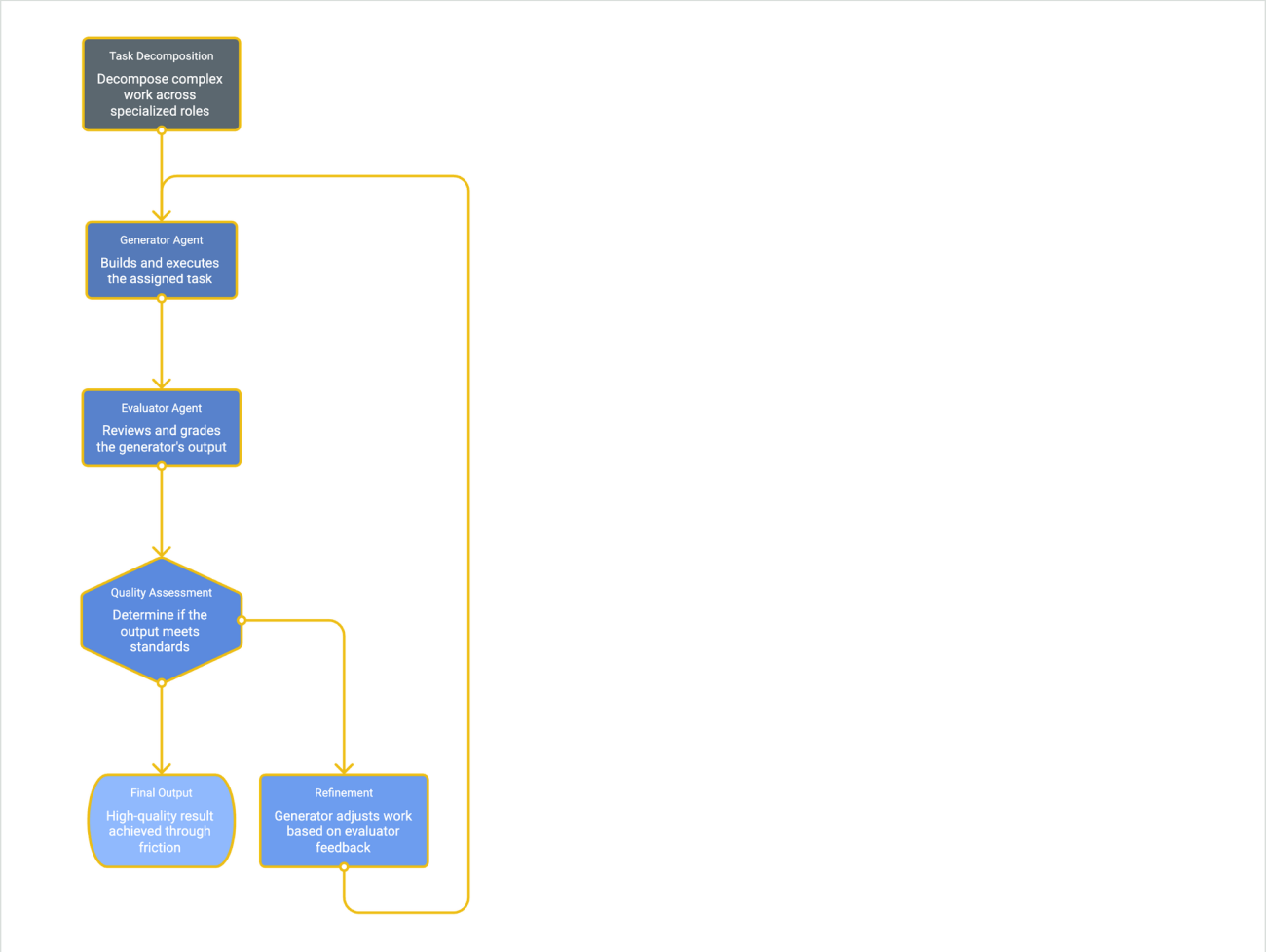

FIGURE 08 / SUBAGENTS - SEPARATING THE GENERATOR FROM THE EVALUATOR.

Loop Control

An agent operates in a loop: take a goal, act, observe the result, decide whether to continue. Loop control governs that cycle (when to proceed, stop, retry, or hand off to a person) and guards against the two common failure modes: stopping before the work is done, and running indefinitely while burning budget.

This is the difference between a demo and a system an organization can fund. Loop control is where stopping conditions, retry limits, approval gates, and escalation rules are set. Without them, an agent is either too cautious to finish or too unconstrained to be trusted with anything that matters.

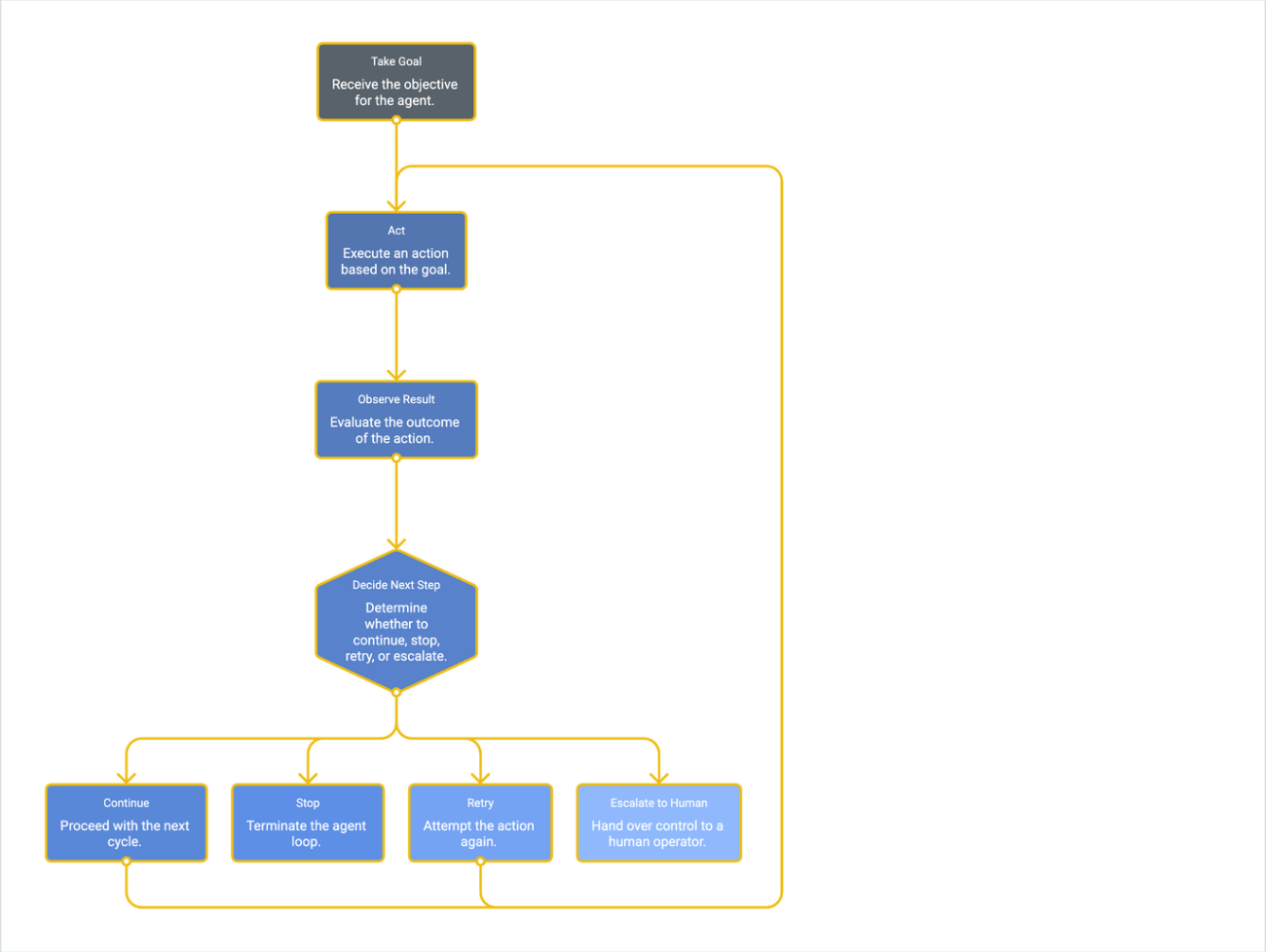

FIGURE 09 / LOOP CONTROL - GOVERNING THE ACT - OBSERVE - DECIDE CYCLE.

Why the Harness Makes Strategy Model-Agnostic

Only one of the nine boxes is the model; the other eight hold the engineering, the knowledge, and the controls. The model still does the reasoning, but it no longer decides whether the system can be trusted, which is what makes it a component you can swap.

That swap is already routine. In the public cloud, changing the model is mostly configuration: run the same harness against several frontier models and compare accuracy, latency, and cost on the real task. Retrieval, sandboxing, skills, permissions, and logging carry over untouched. Only two parts need work: the system prompt, which models follow differently, and the tool-calling layer, which may need a per-model adapter. **Changing the model is configuration; confirming the change is safe is engineering,** so a serious comparison reruns the same evaluation suite against each candidate.

The same portability lets the workload move: from a hosted model, to an open-weight one in a private cloud, to a model on your own hardware. The architecture stays the same; only where inference runs and who can see the data change. The costs are real: open-weight models still trail on long-horizon tool use, structured output, and instruction-following over long context, and self-hosting trades a per-token bill for infrastructure to run. The system doesn't make these moves free; *it makes them evaluable, taken on your terms.* Sensitive work can come in-house, inference can sit where the economics favour it, no single vendor's outage or price hike can take a capability offline, and the ability to switch becomes real leverage.

This argument has limits. The model behaves like a commodity only within a range: above a baseline of capability, below the hardest problems. For novel reasoning and work that has to be right the first time, the strongest model still wins. The supporting parts (vector stores, sandboxes, tool-calling, orchestration) are commoditizing too. The durable advantage was never the mechanics but what you put into them: your policies, your sourced knowledge, your permission boundaries, your evaluation criteria, your rules for when an agent must stop. That inverts the usual framing, where the model is the asset and everything around it is integration work. Frontier models are converging and competing on price; **what a company owns and keeps when the model underneath changes is the layer around it.**

Where to Start

The eight components don't have to be built at once - and shouldn't be. **Controls come before capabilities:**

1. System prompt, loop control, and tool permissions: the control plane that makes an agent safe to authorize at all.
2. Context loading and retrieval, so the agent is grounded in current, auditable information before it gets consequential work.
3. Sandbox, skills, and subagents, scaling capability on a foundation that can already be governed.

A minimum viable harness is the control plane plus grounding; everything beyond that adds leverage.

The Work, and the Test

None of this is glamorous. A harness is configuration, documentation, retrieval pipelines, permission rules, sandboxes, evaluation agents, and stopping conditions: infrastructure, in other words. The agent is the visible part, but this is where an organization's standards become executable, and it is **what separates an AI pilot from a system a business can run on.** It is a standing capability the organization funds as ongoing engineering work, *not a project with an end date.*

An agent's readiness for production depends less on the model's raw capability than on the harness around it. A short set of questions serves as a readiness check:

- Can the agent see the right context without seeing too much?
- Can it use tools without bypassing approval?
- Can it ground its answers in sources you can audit?
- Can a separate evaluator challenge its work?
- Can a human reconstruct the run after something goes wrong?
- Can the whole system be pointed at a different model next quarter without being rebuilt?

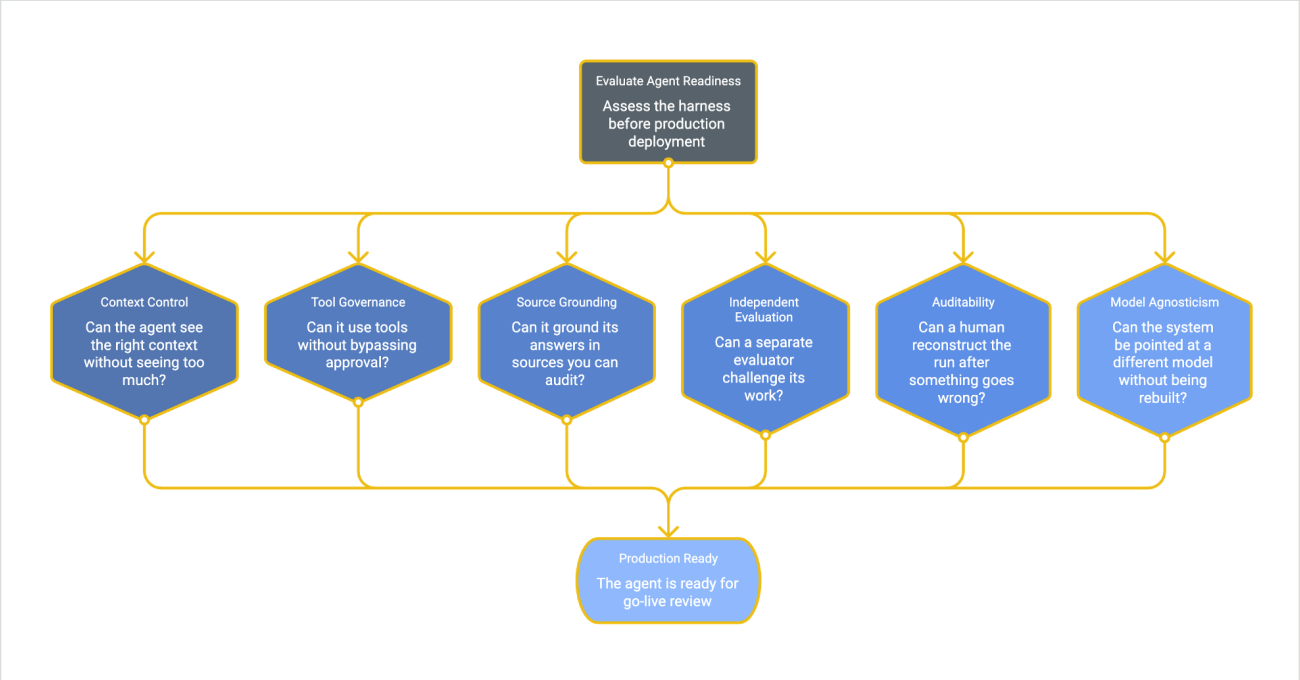

FIGURE 10 / A HARNESS READINESS CHECK BEFORE PRODUCTION.

Organizations that internalize this will spend less effort pushing models to perform and more on engineering the conditions in which good output is the default. The best available model is still worth using, and worth changing when a better one appears. But the advantage that accumulates, **the part a competitor can't acquire by signing the same vendor's contract, is the harness built around it.**

Sources

Birgitta Böckeler, "**Harness engineering for coding agent users**," Martin Fowler, April 2, 2026.
https://martinfowler.com/articles/harness-engineering.html

Vivek Trivedy, "**The Anatomy of an Agent Harness**," LangChain, March 10, 2026.
https://www.langchain.com/blog/the-anatomy-of-an-agent-harness

Ryan Lopopolo, "**Harness engineering: leveraging Codex in an agent-first world**," OpenAI, February 11, 2026.
https://openai.com/index/harness-engineering/

Prithvi Rajasekaran, "**Harness Design for Long-Running Application Development**," Anthropic Engineering, March 24, 2026.
https://www.anthropic.com/engineering/harness-design-long-running-apps

Hailin Zhong and Shengxin Zhu, "**AI Harness Engineering: A Runtime Substrate for Foundation-Model Software Agents**," arXiv, May 13, 2026. https://arxiv.org/abs/2605.13357v1

About the Author

Junior Williams is a **cybersecurity and enterprise architecture leader** focused on *AI, governance, privacy, and secure-by-design implementation*. He advises Canadian enterprises and public-sector organizations on technology strategy, cyber risk, compliance, and responsible AI adoption. He is the founder of **Junior Williams Consulting Inc.** and an incoming **Industry Fellow** with the Rogers Cybersecure Catalyst Fellowship program at Toronto Metropolitan University for 2026 - 2027, with applied research and standards work spanning *ISO/IEC mirror committees, CSA Group, the AIUC-1 Consortium,* and AI governance initiatives.

JUNIOR WILLIAMS - 2026 BIKE RIDE FOR BRAIN HEALTH

SOURCE / LINKEDIN ARTICLE

Book an intro call