NEWSLETTER

The Architecture of Reliable Agents: Harnessing, Local Inference, and Open-Source Control

April 21, 2026

Agents are rapidly moving beyond single-prompt interactions into multi-day, multi-session workflows. But as context windows remain discrete, bridging the gap between sessions requires more than just a bigger model—it demands rigorous harness engineering. From Anthropic's two-agent patterns to Martin Fowler's feedforward/feedback frameworks, the industry is converging on a new standard for reliable agentic coding. Meanwhile, local inference is catching up, with Qwen3.6 enabling complex tool-calling on consumer hardware. Here’s how the stack is evolving to support long-running, autonomous agents.

01

The Two-Agent Pattern and the Memory Bridge

The core challenge of long-running agents is that they must work in discrete sessions, and each new session begins with no memory of what came before. Anthropic's research into this problem revealed that even frontier models fail when asked to one-shot complex applications across multiple context windows. Out of the box, agents tend to attempt to build entire apps in a single pass, often running out of context mid-implementation. A second failure mode occurs later in the project, where an agent looks around, sees partial progress, and prematurely declares the job complete. Effective harnesses for long-running agents (November 2025)

Anthropic solved these failures with a structured two-agent pattern. The first agent, an initializer, creates a comprehensive environment scaffold, including a 200+ item feature list and a `claude-progress.txt` log. By using JSON for the feature list rather than Markdown, they found that models are significantly less likely to accidentally overwrite or corrupt the structure during subsequent edits. This initializer sets the stage for a dedicated coding agent that is strictly instructed to work incrementally, one feature at a time.

The true innovation lies in the "memory bridge" between sessions. Anthropic found that asking the coding agent to commit clean, merge-ready code to git and update the progress log is critical. This allows the next agent instance to instantly understand the state of the project without guessing what happened in the previous context window. The result is a system that behaves less like a fragile prompt-executor and more like a persistent, shift-based engineering team.

> FURTHER READING

Anthropic's Autonomous Coding Quickstart — November 2025

02

The Engineering Discipline: Feedforward, Feedback, and the Steering Loop

To move from experimental scripts to production-grade agent systems, we need a rigorous mental model for what surrounds the model. Martin Fowler's April 2026 framework introduces the concept of the agent harness as a cybernetic governor, combining **feedforward controls** (guides that prevent bad outputs) and **feedback controls** (sensors that enable self-correction). Without both, you get an agent that either repeats the same mistakes or encodes rules without ever knowing if they worked.

Fowler categorizes these controls into two execution types: computational and inferential. Computational controls are deterministic and fast, running on the CPU in milliseconds. Examples include linters, type checkers, and structural analysis tools that provide reliable, immediate feedback. Inferential controls, on the other hand, rely on semantic analysis and AI code review. They are slower and more expensive, but they provide the nuanced judgment necessary for complex architectural decisions.

The most critical practice in harness engineering is the "steering loop." Whenever an agent repeats a specific mistake, the harness must be iteratively improved to make that issue less probable in the future. This involves distributing quality checks across the development lifecycle: running fast computational checks before a commit is even created, and reserving expensive inferential checks for post-integration pipelines. By continuously refining the harness based on observed agent behavior, developers can drastically reduce review toil and increase system quality. Harness engineering for coding agent users (April 2026)

03

Open-Source Infrastructure for Multi-Day Agent Workflows

The open-source community is rapidly maturing the infrastructure required to support these long-running patterns without vendor lock-in. OpenHarness, which recently hit 10.7k GitHub stars, delivers a complete Python-based agent loop, toolkit, and multi-agent coordination system. It provides the essential plumbing for the control plane, including streaming tool-call cycles, API retry logic with exponential backoff, and a governance layer with path-level command rules.

A standout feature of OpenHarness is its personal agent, ohmo, which leverages "Auto-Compaction" to preserve task state across context compression. This allows agents to run multi-day sessions without manual intervention, automatically managing the trade-off between context window limits and long-term memory. ohmo can fork branches, write code, run tests, and open PRs autonomously, running on existing Claude Code or Codex subscriptions without requiring extra API keys.

OpenHarness also abstracts the complexity of provider integration, supporting Claude, OpenAI, Gemini, and local models through a unified interface. By standardizing the harness architecture—agent loop, context compression, and governance—OpenHarness allows developers to focus on agent behavior and swarm coordination rather than reinventing the infrastructure wheel. *Additional info:* OpenHarness & ohmo (date unavailable)

04

Local Agentic Coding on Consumer Hardware

For developers prioritizing privacy and cost, Qwen3.6-35B-A3B represents a significant leap in local inference capabilities. Released in April 2026, this multimodal hybrid-thinking model excels at agentic coding and tool-calling while running on as little as 17GB of unified memory using Unsloth's 3-bit quantization, or 23GB with 4-bit. This hardware accessibility makes sophisticated agent loops viable on consumer MacBooks and workstations without relying on expensive cloud APIs.

Unsloth's Dynamic 4-bit quantization uses real-world calibration datasets to maintain accuracy, ensuring that the model doesn't degrade when compressed for local deployment. The model supports a 256K context window and has been specifically optimized for nested object parsing, which is critical for reliable tool execution in complex coding environments.

Running these models locally also requires careful parameter tuning to maximize reliability. Unsloth provides specific configurations to prevent common agentic failures:

- Temperature of 0.6 for precise coding tasks to reduce hallucination.
- Top_p of 0.95 to balance creativity and focus.
- Disabled repetition penalty for coding to avoid recursive code loops.

While Ollama currently lacks support for Qwen3.6 GGUFs due to separate mmproj vision files, llama.cpp compatible backends handle the workload efficiently, making local agentic coding a practical reality. Qwen3.6 - How to Run Locally (April 2026)

05

The Fine-Tuning to Inference Pipeline and the Template Trap

Fine-tuning a model is only half the battle; deploying it reliably requires bridging the gap between training frameworks and inference layers like Ollama. Unsloth's ecosystem allows fine-tuning via QLoRA on as little as 3GB VRAM, but exporting to Ollama introduces a critical vulnerability: chat template mismatches. If the template used during inference doesn't exactly match the training template, the model will produce gibberish, rendering the fine-tuning useless.

Ollama's Modelfile system solves this by allowing developers to explicitly define the `TEMPLATE` (using Go syntax) and `ADAPTER` (LoRA) instructions. The `FROM` instruction specifies the base model, while `PARAMETER` controls inference settings like temperature and context window size. By auto-generating Modelfiles with the correct chat templates during export, Unsloth ensures that the EOS token and prompt formatting align perfectly between training and deployment.

This precision is vital for long-running agents that rely on consistent tool-calling formats. A mismatched template can cause the model to fail to recognize tool boundaries, leading to infinite generation loops or malformed JSON outputs. Ensuring that training and inference frameworks speak the exact same language is the difference between a functional custom agent and a broken one. *Additional info:* Modelfile Reference (date unavailable)

The convergence of structured harness patterns, open-source orchestration, and efficient local inference is transforming AI agents from novelty demos into reliable engineering tools. As the stack matures, the focus is shifting from raw model size to the architectural discipline of the

harness itself. The agents that will dominate the next era of software development won't just be the smartest—they'll be the most rigorously engineered.