

Why Google's New Harness² Open Source Release Changes How I Think About AI Agents

5 min read · 3 days ago



Nikhil

Follow



Listen



Share



More

Moving past the endless retraining by fixing the operating system around our AI agents -> this is what google is trying to do.

Why Google's New Harness² Open Source Release Changes How I Think About AI Agents



Google new Opensource Harness2 to reach RSI

For the past year, our conversations with other developers about AI agents have followed the exact same script. An agent fails at a complex task, like reconciling a messy spreadsheet or pulling specific legal clauses from a folder of PDFs.

Our immediate reaction is always to blame the model. We wait for the next frontier release, or we spend weeks setting up a fine tuning pipeline, hoping that changing the model weights will magically fix a broken workflow.

When Google Research quietly dropped the open source code for **Harness2** on Github a few days ago under an Apache 2.0 licence.

After spending time reading through their October 2026 research paper (Harness2: Recursive Agent Harnessing for an Open World) and inspecting the repository, I am convinced we have been solving the wrong problem. Most of the time, our AI agents do not need a smarter base model. They need a better working environment.

What is Google Harness2

In Simple language, Harness² is an open source Python framework from Google Research that helps AI agents improve at unfamiliar tasks by automatically upgrading their operating setup while keeping the underlying AI model frozen.

To understand why I find this so practical, we first need to clarify what a 'harness' actually is. When you deploy an AI agent, the large language model is just the brain in a jar. The harness is the entire system wrapping that brain. It includes the instructions you give it, the tools it can access, how it stores memory, and how it delegates subtasks.

Usually, engineers build a harness once by hand and leave it static. When the agent hits a real world problem that the developer did not anticipate, the static harness breaks. Harness² solves this by treating the harness itself as something that can be tested, rewritten and improved automatically during execution.

The Eight Editable Components Around the Model

Important aspect in the google-research/harness2 repository is how specific the architecture is. Many frameworks stop at automated prompt tweaking. If you have ever built a production agent, you know that changing a few words in a system prompt rarely fixes a deep workflow issue.

Harness² exposes eight distinct components around the frozen model that it can read and modify:

1. System Prompt: this defines the main role and objective for the harness
2. Guardrails: This sets the strict behavioral rules and boundaries

3. Memory: Controlling how the agent stores and recalls context across steps.
4. Tool guidance: Explains when and how to use specific tools that are available
5. Subagents: manages how work is split and delegated to helper agents.
6. Skills: Packages reusable procedures for specific jobs.
7. Scripts: Provides executable code helpers for routine actions.
8. Plugins: Connects external extensions into the tool loop.

By giving framework permission to edit all eight areas, you allow the system to fix the actual root cause of a failure. If an agent keeps crashing because it reads a massive file all at once, Harness² does not just tell the prompt to “be careful”. It can write a validation script, adjust the tool guidance, or spin up a subagent to chunk the data first.

How the Propose, Probe, and Compose Loop Works.

The core engine inside Harness² runs on a three step cycle that the authors call **Propose, Probe and Compose**. What makes this genuinely clever to me is that it works without needing a human answer key (ground truth feedback) during the task.

Suppose you ask an agent to adjust a set of financial records and it struggles.

How Harness² steps in

- First during the **Propose** stage, an improved model looks at the task and generates alternative versions of the harness. One version might add a new verification skill. Another version might tighten the guardrails around how numbers are extracted.
- Second, in the **Probe** stage, the framework runs these different candidate harnesses on the exact task using the exact same frozen model (such as Gemini 3.8 flash or GLM 5) and the same executor (like OpenCode or Codex). Instead of checking the final answer against a grading rubric that does not exist yet, it observes the behavioral differences between the runs. Did one version inspect the directory before editing files while the other blindly overwrote data? That execution trace is the evidence.
- Third, during the compose stage, Harness² compares those execution traces,

picks the modifications that actually helped the agent behave more sensibly, and combines them into a refined harness for the final run.

You can run this loop in parallel recursion (testing multiple wide ideas at one) or sequential recursion (stacking refinements step by step to go deeper).

Why the Dual Loop Memory Matters

If Harness² had to run that entire testing loop from scratch on every single ticket without learning anything permanent, it would be too slow and expensive for my taste.

Thankfully, the researchers built a second loop called harnessing level recursion. Every time the agent finishes a task and finds a harness recursion. Every time the agent finishes a task and finds a harness upgrade that worked, it saves the winning setup and the evidence into a persistent improvement library, including a local EXPERIENCE.MD file.

When a brand new task arrives tomorrow, the agent still start from a clean base harness so old modifications do not pollute the new job. However, the improver reads the library of past lessons to propose smarter upgrades right out of the gate. In my view, this closely mirrors how human software teams work, we keep a clean main branch, but we bring out past debugging experience to every new sprint.

Looking at the Numbers

On Workbuddy Bench (which tests office, web and coding work), adding Harness² to an OpenCode executor running GLM 5 jumped from average score from 64.2 to 75.7. That is 11.5 point gains without touching a single weight inside the model. With Gemini 3.5 Flash, the average score climbed from 72.4 to 77.5, while also cutting down the average number of steps the agent needed to finish the job.

The test is also done on **JobBench** for professional workflows and LAB for complex legal tasks, showing steady gains across board.