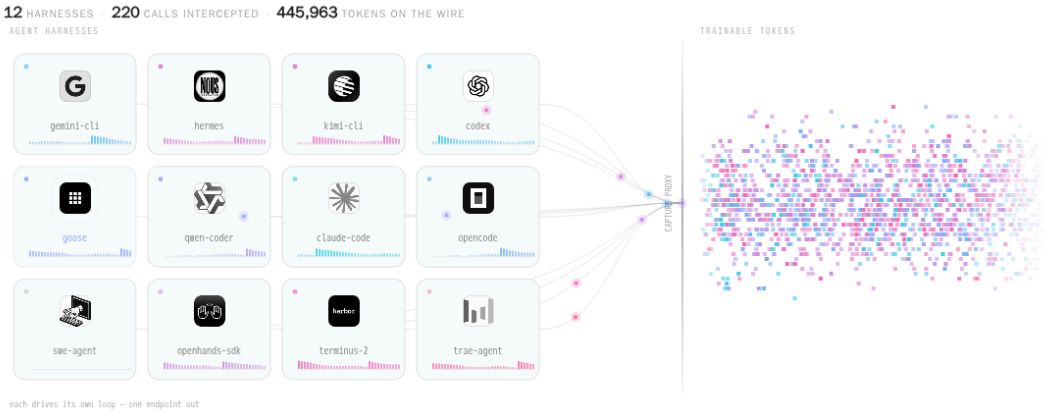
The ultimate guide to multi-harness RL



The same model performs differently in every agent harness. Now you can train it with RL inside the ones people actually use, like Claude Code and Codex, and improve it across all of them.

AUTHOR

Adithya S Kolavi

AFFILIATION

Hugging Face

PUBLISHED

September 24, 2026

Table of Contents

Introduction

If you use AI to write code, you have probably used the same model through more than one tool: Claude in Claude Code, in Cursor, in Pi, or in Cline. And you may have noticed it doesn't behave the same way in each. It plans differently, reaches for different tools, and finishes tasks in one that it gets stuck on in another.

The difference can be measured, and it comes from the program wrapped around the model, called an agent harness. It runs the loop, decides which tools the model gets, writes the context the model reads, makes sense of what the model sends back, and decides when to stop. Change the harness and you change what the model sees and what it is allowed to do, so the results change too.

What exactly counts as a harness?

This matters most for the open-weight models you run yourself. A model that was never trained in your harness can struggle there. It calls tools the harness doesn't have, or writes output the harness can't read.

The fix is to train the model inside those harnesses, and this article introduces an open framework that makes it possible: `pick a model, pick a harness, pick a sandbox, and train`. We walk through how it works, and what happened when we used it to train two small open models, Qwen3.5-2B and LFM2.5-2.6B, across several harnesses at once.

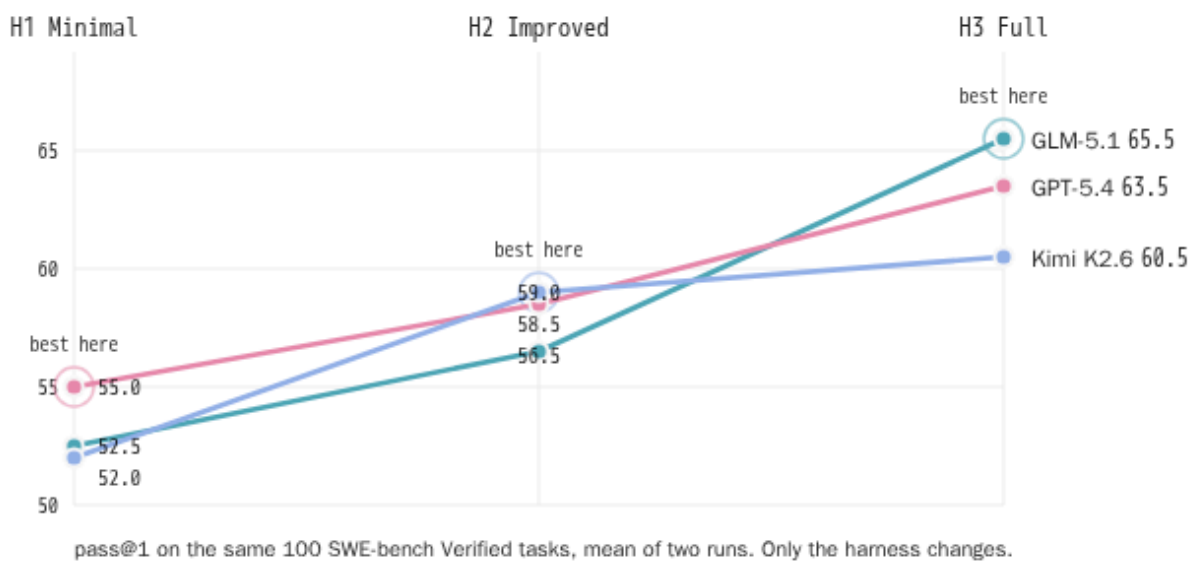
Benchmark scores now come with a harness attached

Model cards have started saying which harness a score came from. GLM-4.7 promises "significant improvements on complex tasks in mainstream agent frameworks such as Claude Code, Kilo Code, Cline, and Roo Code" (Z.ai, 2025). Kimi K2 reports Terminal-Bench (Merrill et al., 2026) twice, 25.0 under Terminus and 30.0 under Moonshot's own framework (Kimi Team, 2025). MiniMax M2 names a harness for almost every benchmark it lists (MiniMax, 2025). DeepSeek-V3.2's thinking mode wouldn't run under Terminus at all, so its Terminal-Bench number had to come from a different harness (DeepSeek-AI, 2025).

The harness can even change which model comes out ahead. A 2026 paper on harness disclosure took three models within three points of each other on a public coding leaderboard, GLM-5.1, GPT-5.4 and Kimi K2.6, and ran them on the same 100 SWE-bench Verified (Jimenez

et al., 2024) tasks under three harness configurations, with everything else held fixed (Zhang et al., 2026).

Same models, same tasks, three harnesses



pass@1 on the same 100 SWE-bench Verified tasks, mean of two runs. Only the harness changes.

LEGEND

GLM-5.1 GPT-5.4 Kimi K2.6

THE THREE CONFIGURATIONS

H1 no compression, verbose tools, no retries H2 compressed context, minimal tools, retries

H3 H2 plus self-checks, drift checks, rollback

Three models that sit within 3 points of each other on a public coding leaderboard, run across three harness configurations under matched conditions: same task subset, same container, same 50-step budget, same timeout. Each model is best under a different configuration. Changing the harness moves GLM-5.1 by 13.0 points; changing the model inside a fixed harness moves scores by 2.5 to 5.0. The authors count six ranking reversals across the nine model-pair/harness-pair comparisons, and harness-induced variance 7.8 times model-induced variance. Source: Table 2, Stop Comparing LLM Agents Without Disclosing the Harness.

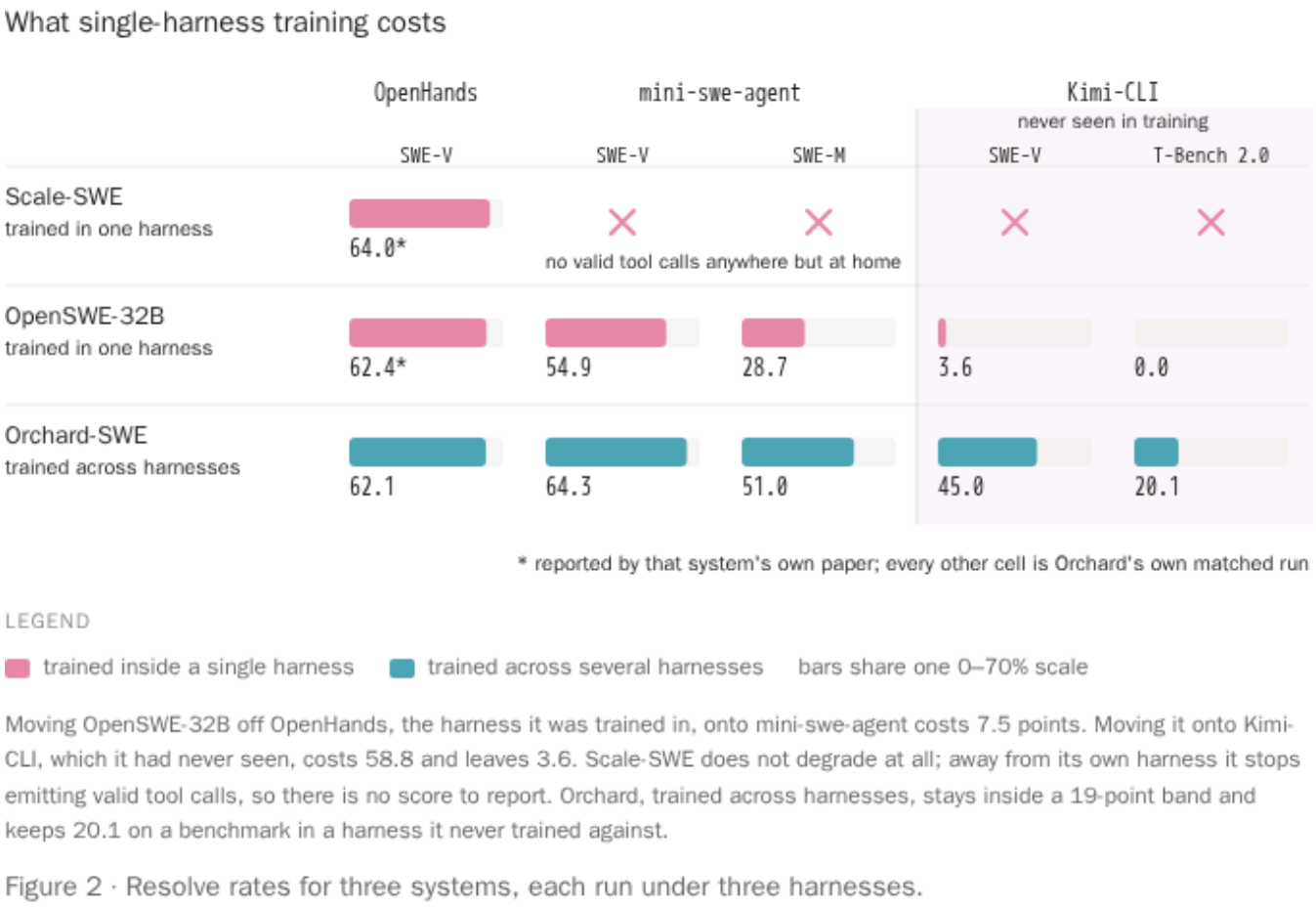
Figure 1 · Each of the three models is the best model under a different harness configuration.

Each model comes out on top under a different harness. Swapping the harness moved GLM-5.1 by 13 points, while swapping the model inside the same harness moved the score by only 2.5 to 5.

You can see the same thing outside a controlled test. Claude Opus 4.5 scores 45.9% on SWE-bench Pro (Deng et al., 2025) on Scale's SEAL leaderboard and 55.4% inside Claude Code, with the same weights (Zhang et al., 2026).

When a model leaves the harness it was trained in

Those are frontier models, and they work in every harness they were tested in. A model trained inside just one harness can do far worse when you move it. The Orchard paper measured this (Peng et al., 2026). It took two open coding models, each trained in a single harness, and ran them under three harnesses, including one neither had seen, alongside Orchard's own model, trained across harnesses.



Moving OpenSWE-32B off OpenHands, the harness it was trained in, onto mini-swe-agent costs 7.5 points. Moving it onto Kimi-CLI, which it had never seen, costs 58.8 and leaves 3.6. Scale-SWE does not degrade at all; away from its own harness it stops emitting valid tool calls, so there is no score to report. Orchard, trained across harnesses, stays inside a 19-point band and keeps 20.1 on a benchmark in a harness it never trained against.

Figure 2 · Resolve rates for three systems, each run under three harnesses.

Moving OpenSWE-32B from OpenHands, where it was trained, to Mini-SWE-Agent costs 7.5 points. Under Kimi-CLI, which it had never seen, it drops 58.8 points, to 3.6% on SWE-bench Verified, and scores zero on Terminal-Bench 2.0. Scale-SWE stops producing valid tool calls anywhere but its home harness. Orchard's model stays between 45.0 and 64.3 on SWE-bench Verified across all three.

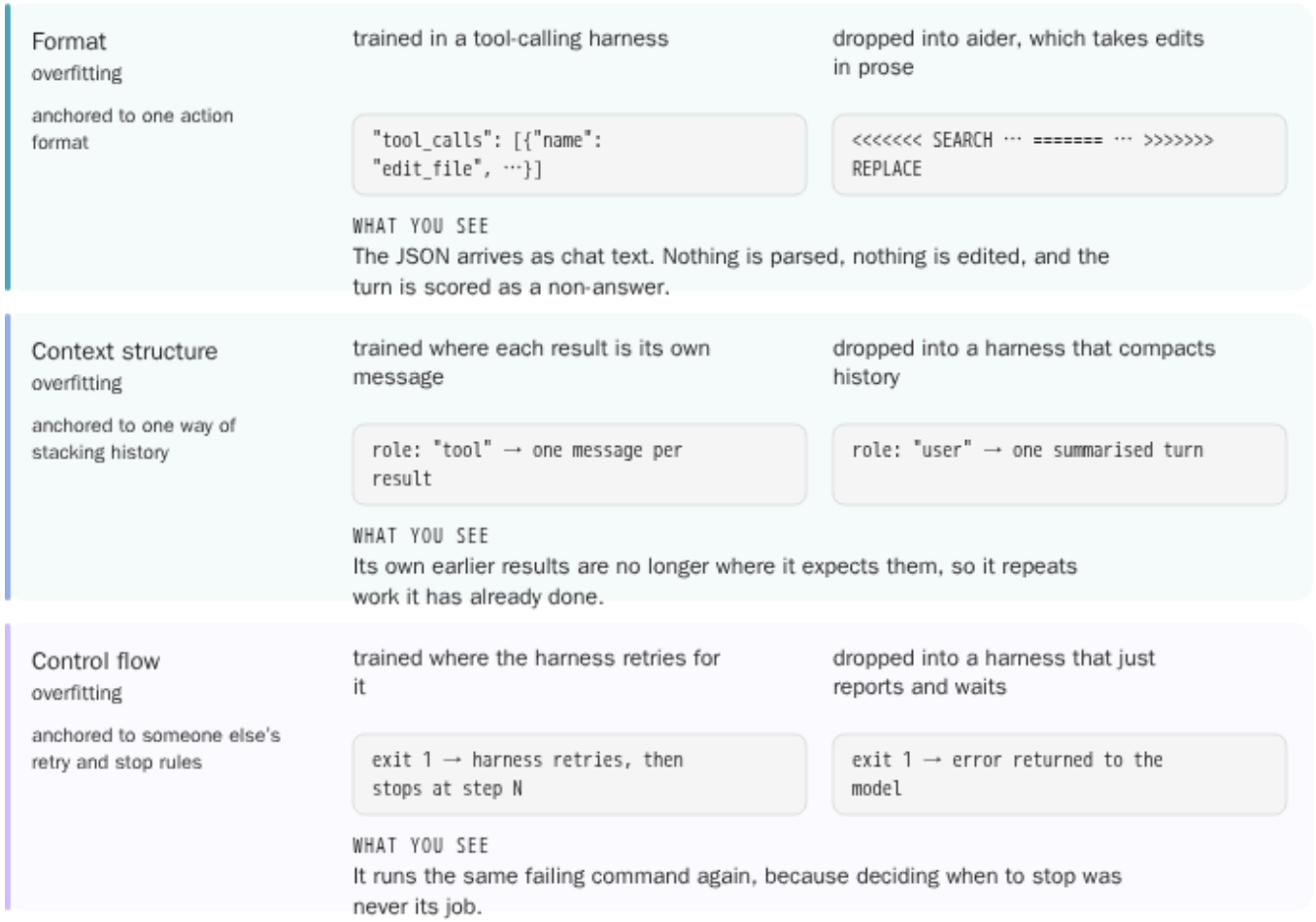
The paper calls these a degraded resolve rate, where the model still works but solves less, and a catastrophic format failure, where its output stops being usable at all. Each row of that figure is one fixed model, and each benchmark uses the same tasks under every harness, so the drop comes from the harness.

The KwaiKAT team puts the cause well (KwaiKAT Team, 2026):

> *"In agentic RL, if training relies solely on a single fixed harness, the model often learns not 'how to solve the task' but 'how to solve the task under that particular harness's interface conventions.'"*

They split it into three kinds of overfitting: to the harness's action format, to the way it lays out context, and to its control flow, the retries and stop conditions the model learns to lean on.

Three ways a model overfits to its harness



The three modes are the KAT-Coder team's; the examples are the form each one takes on the wire. All three have the same cure and it is not three separate patches: vary the harness during rollouts, and none of the three has a fixed target to anchor to.

Figure 3 · What the model anchored to, and what breaks when that changes.

Frontier labs now train across harnesses on purpose

KwaiKAT's answer, which they call harness scaling, is to vary the harness during training. poolside's Laguna report does this (poolside, 2026):

> *"To encourage generalization across diverse agent harnesses, our training data includes trajectories from external frameworks such as OpenHands, OpenCode, and Mini-SWE-Agent."*

That is 1.3 billion tokens of supervised trajectories from harnesses poolside doesn't own, with each one's own habits left in. Their reinforcement learning then runs inside their own harness, `pool`.

Kimi K3's RL environment builds harnesses such as Claude Code and Codex from composable modules, so the model trains on many harness configurations instead of one (Kimi Team et al., 2026). Qwen3-Coder-Next generates its agentic coding data in six different harnesses (Qwen Team, 2026), and MiMo-V2.6 trains on a pool of task-adapted mini-harnesses (LLM-Core Xiaomi, 2026).

OpenForgeRL checked whether that variety costs peak performance. It trained the same model on one harness and on three, and the three-harness model scored higher even on the single harness's home turf, 48.5 to 46.0, and almost twice as high under Codex (Yu et al., 2026).

The claims, in the papers themselves

poolside | Kimi | Qwen | Xiaomi MiMo | KwaiKAT | Orchard | OpenForgeRL | Agent Lightning

LAGUNA M.1/XS.2 Technical Report — §4.3.3, p20 · arXiv:2605.27605

directly.

4.3.3 Multi-harness Training

To encourage generalization across diverse agent harnesses, our training data includes trajectories from external frameworks such as OpenHands [90], OpenCode[2], and Mini-SWE-Agent[3]. Specifically, we augment the supervised fine-tuning mix with 1.3B tokens of multi-harness agentic trajectories spanning a broad range of software engineering tasks. We intentionally preserve native harness behaviors during data collection — including custom subagent spawning, context compaction, planning scaffolds, and reminder systems — so the model sees a broad distribution of interaction patterns. These frameworks vary substantially in trajectory length, tool use, subagent orchestration, and harness-specific quirks, all of which improved generalization in internal evaluations.

A frontier lab budgets for this. 1.3B tokens of supervised trajectories collected across OpenHands, OpenCode and Mini-SWE-Agent, with each harness's native quirks deliberately left intact.

Unretouched crops from the official PDFs; the only edit is the crop. Pick a paper to stop the rotation.

Figure 4 · Eight 2026 reports, at the page cited.

What is missing

None of this needs a frontier-sized budget. Agent Lightning takes Qwen3.5-9B from 41.8% to 56.4% on SWE-bench Verified with roughly 6,000 training examples (He et al., 2026). The same paper names the hard part. The harness owns the loop, so the trainer only sees requests and responses going past, and turning those back into training samples is still an open problem.

An open, reliable way to do it is still missing. It has to capture everything RL training uses from harnesses you don't control, and plug into whichever trainer you already use.

We built this on OpenEnv (Meta PyTorch & Hugging Face, 2025), around a capture proxy that sits between the harness and the model. Every call the agent makes passes through the proxy, which records the prompt and completion tokens as the model saw and produced them, with their log probabilities. It speaks the API formats coding agents use, so Claude Code, Codex and OpenCode run unmodified, and a custom harness only has to point at it. Because it all sits behind OpenEnv's open environment interface, the same setup can feed different trainers. This article uses TRL (von Werra et al., 2020).

On top of that, the Harbor integration (Kolavi, 2026) serves Harbor's containerized tasks as OpenEnv environments, so the harness and the sandbox become settings you choose per rollout.

What is Harbor?

What is a harness?

The introduction gave the one-line version: the harness is the program that runs the agent. This chapter is about where its edges are, because every later decision in the article turns on one of them.

The tightest definition in the literature comes from a position paper on benchmark disclosure, which calls it "the software layer between the model and the task that constructs the context the model sees, mediates its tool calls, validates its outputs, and decides when to retry, escalate, or stop" (Zhang et al., 2026). From the model's side the harness is the interface. From the trainer's side it is an executable you do not control, which is the fact the whole article has to work around.

It helps to say what a harness is not, because several neighbouring things get called by the same name in casual writing. Hugging Face's agent glossary is the fuller version of this vocabulary; what follows is the short form this article needs. The model is the next-token distribution, stateless, with no loop and no memory between calls. The sandbox is the isolated place actions actually run, a container or a VM or a tmux session. A benchmark is a fixed task set plus a protocol for comparing results on it. None of those is the harness.

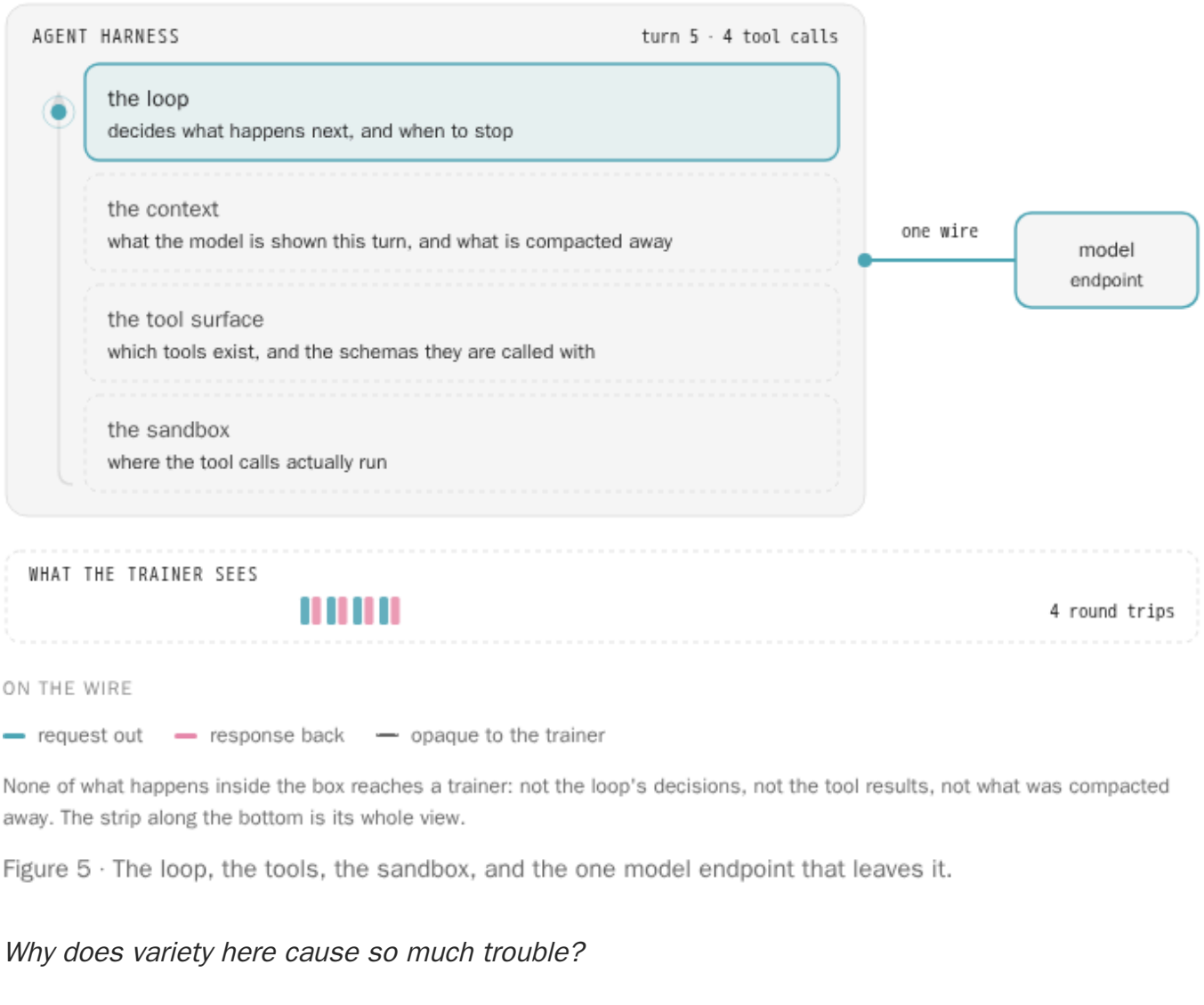
The one worth dwelling on is the RL environment, because the difference is direction rather than content. An environment owns state, transition and reward: it receives an action and returns an observation. A harness executes the loop. A harness pulls, an environment is pulled, and that is the whole distinction the rest of this article turns on.

> *Note: the previous guide (Kolavi et al., 2026) uses "harness" in a narrower sense, meaning the trainer-side interaction layer, the tool surface a trainer drives. This article uses it in the sense the 2026 literature settled on, meaning the whole deployed agent program that owns the loop. Same word, nearly opposite locus of control, so it is worth keeping the two straight when reading across the two articles.*

The cleanest statement of the boundary is not a paper but a plugin contract. OpenClaw's agent-harness SDK defines a harness by negation: it is "the low level executor for one prepared OpenClaw agent turn," and explicitly "not a model provider, not a channel, and not a tool registry." The harness owns session runtime and resumption, native tool execution, event streaming and auth bootstrap. The platform around it owns provider and model selection, transcript files, tool policy and schema normalization.

That split is the whole opening for this article. The harness decides *which call to make*. It does not, in the end, decide *where that call goes*.

Inside a harness



None of what happens inside the box reaches a trainer: not the loop's decisions, not the tool results, not what was compacted away. The strip along the bottom is its whole view.

Figure 5 · The loop, the tools, the sandbox, and the one model endpoint that leaves it.

Why does variety here cause so much trouble?

Because the variety is not cosmetic. Harnesses differ in the API dialect they speak, meaning the request and response format they use to talk to a model server: OpenAI chat-completions, OpenAI Responses, Anthropic Messages, Google Gemini. They differ in whether they use tool calling at all, in how they compact context, and in how much of the control flow they take away from the model. Those are exactly the axes the KAT-Coder team identified as the ones a model overfits to (KwaiKAT Team, 2026).

The harness landscape

Harness	Anthropic Messages	OpenAI Responses	Google generateContent	chat completions	plain text	Tool calls
Claude Code	●					yes
Codex		●				yes
OpenCode	●	●	●	●		yes
Goose	●		●	●		yes
SWE-agent				●		yes
Mini-SWE-Agent		○		●	○	one tool
OpenHands SDK				●		yes
Aider				●		no
Qwen Code			●	○		yes
Kimi CLI				●		yes
Terminus 2				●		no
Hermes			*three APIs, switchable*			yes

LEGEND

● speaks it by default ○ also supports it

The model API a harness speaks decides whether a capture proxy can sit in front of it without changes to the harness.

Figure 6 · The model APIs each harness speaks. Hover a harness for its maker and what sets it apart.

Mini-SWE-Agent's own README gives scaffold overfitting as a reason to use it, so harness authors knew about the problem before any tech report named it. Aider is the far end of the range. It does not use tool calling at all and asks the model to write edit blocks in prose, so a model trained on structured tool calls has to switch to a different output format entirely.

In practice the failure often shows up as an error rather than a lower score. A model trained in one harness calls a tool by that harness's name for it, with that harness's argument names, and a harness that spells them differently rejects the call before the tool ever runs.

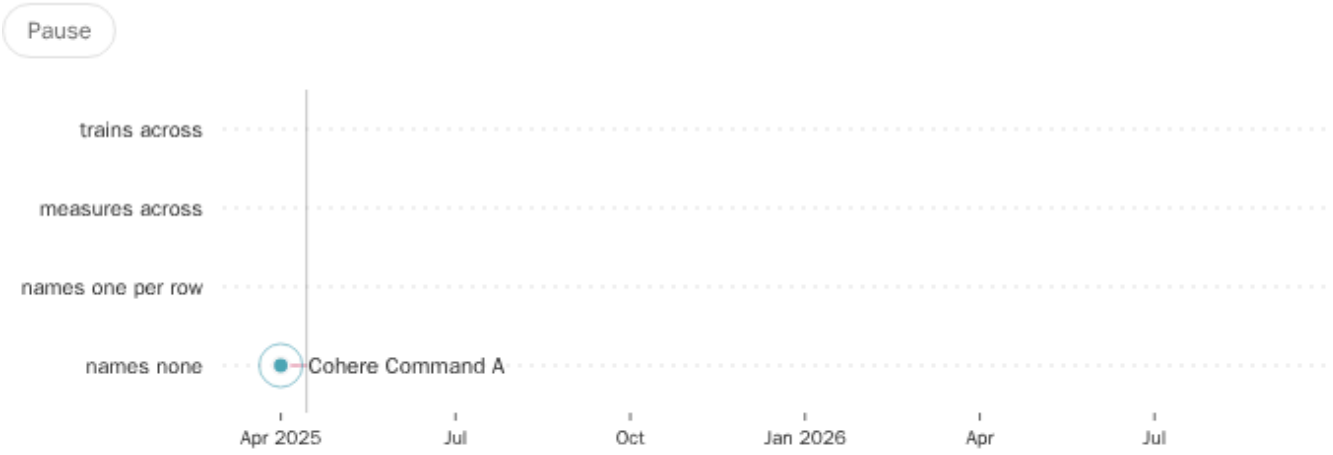
What harness overfitting looks like

The task, the reasoning and the answer are all fine. The call fails because the model learned one harness's spelling of its tools.

Figure 7 · One tool call, accepted by the harness the model trained in and rejected by one it never saw.

Model reports only recently started saying which harness a number came from. In 2025 most named none or one per benchmark. By 2026 every report on the timeline below trains across several.

From naming no harness to training across several



Cohere Command A · Apr 2025 names none
The words harness and scaffold appear zero times.

Figure 8 · Recent model releases over time, by the most each report says about harnesses. Hover or tap a dot for details.

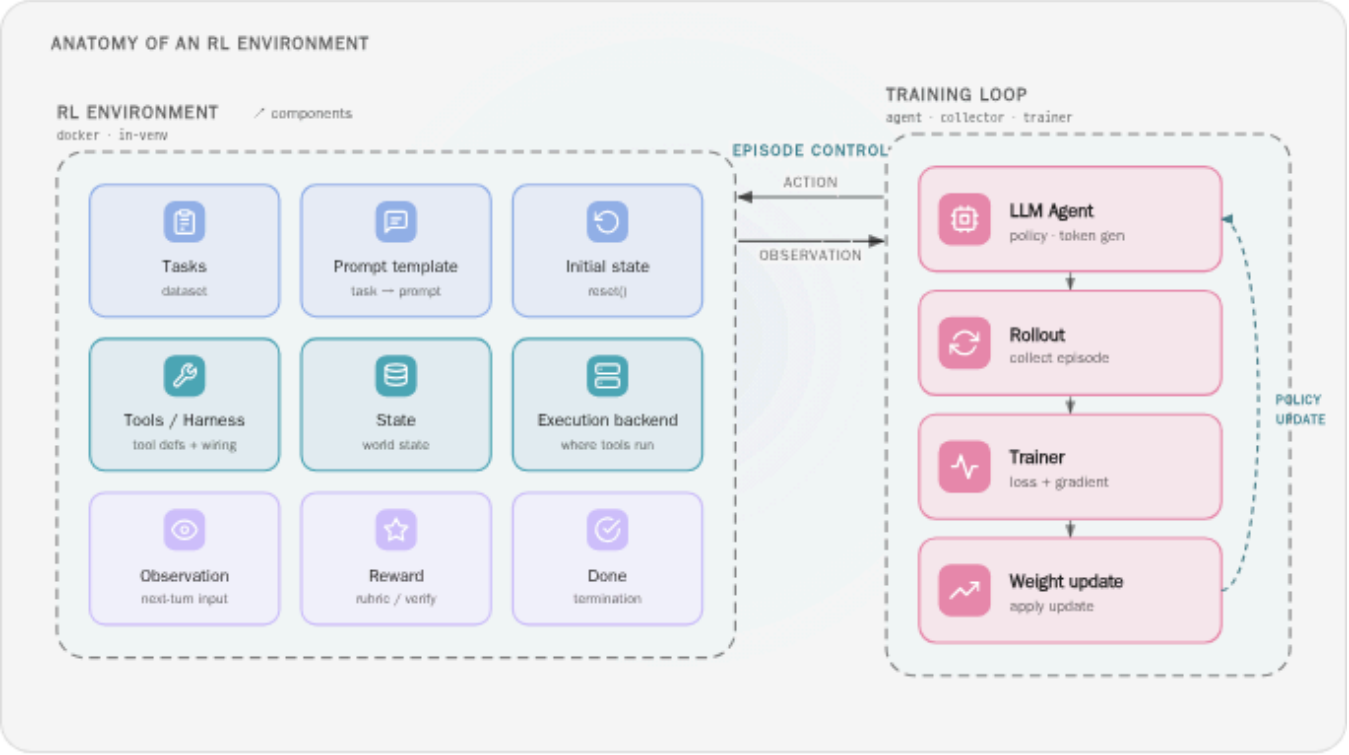
The list of harnesses keeps growing. Harbor 0.22.0 ships adapters for more than 40 of them, and OpenEnv has validated ten end to end, so a model trained on a few will keep meeting ones it never saw.

What are RL environments?

Technical overview

An RL environment is the stateful thing a policy acts on. It takes an action, updates its internal state, returns an observation, and at some point returns a reward. In the LLM era the actions are tool calls, the observations are tool results, and the reward comes from a grader that looks at the finished trajectory. What separates it from a benchmark is that a benchmark is a fixed task set plus a comparison protocol, administered on top of environments.

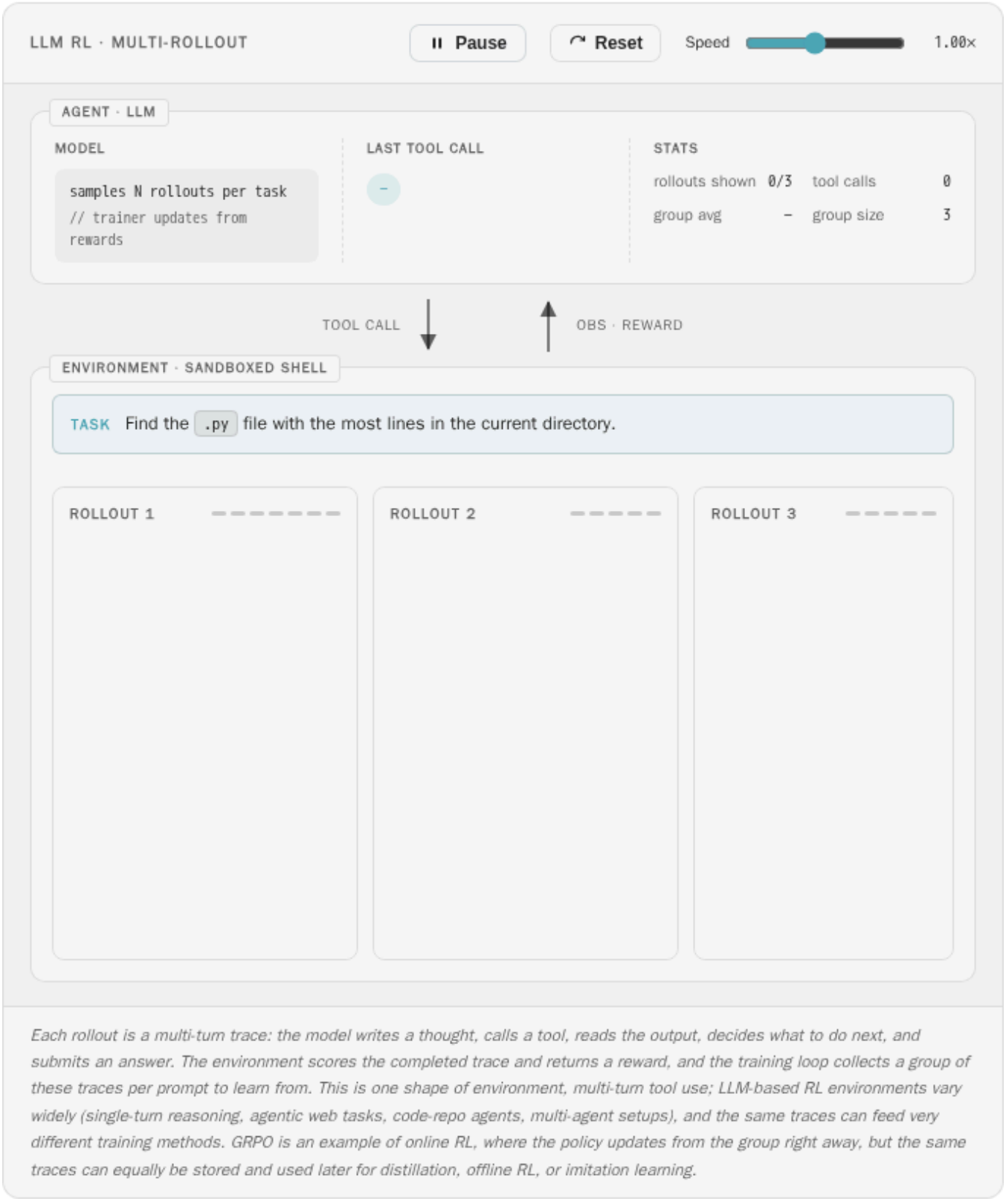
Drawn out, the two halves look like this. On one side a training loop that samples, scores and updates. On the other an environment made of far more parts than the word suggests.



The pieces inside that box are worth naming once, because the rest of this article keeps reaching for them: a task set, a prompt template, tools, observations, an execution backend, world state, a reward rule and a termination condition. Every environment has all of them, whether or not its framework gives you a name for each. Note the box labelled tools and harness, which is the previous guide's narrower use of the word, meaning the tool surface rather than the agent program. The previous guide (Kolavi et al., 2026) tabulates all of these component by component, with an example of each, so there is no need to repeat it here.

And this is the shape a single rollout takes once the model is a language model and the actions are tool calls: write, run, read the result, decide what to try next, and eventually submit

something a grader can score.



Each rollout is a multi-turn trace: the model writes a thought, calls a tool, reads the output, decides what to do next, and submits an answer. The environment scores the completed trace and returns a reward, and the training loop collects a group of these traces per prompt to learn from. This is one shape of environment, multi-turn tool use; LLM-based RL environments vary widely (single-turn reasoning, agentic web tasks, code-repo agents, multi-agent setups), and the same traces can feed very different training methods. GRPO is an example of online RL, where the policy updates from the group right away, but the same traces can equally be stored and used later for distillation, offline RL, or imitation learning.

That is the whole definition this article needs, and all three figures are lifted from The ultimate guide to RL environments rather than redrawn, so the vocabulary carries over intact. That guide spends a full chapter on the anatomy, walks the five-stage spine from tasks through harness and reward to rollout collection and training, and compares how six frameworks slice it. If

anything above feels thin, that is where the depth is. This chapter only covers what changes once the agent, rather than the trainer, is driving.

White-box vs black-box RL environments

Who actually drives the rollout?

This is the distinction the rest of the article turns on, and it is not about what the task is. It is about which side of the boundary owns the loop.

In a white-box environment the trainer owns it. The trainer samples an action, calls `env.step()`, reads the observation, samples again. The environment is passive, it waits to be called, and every token the policy produced is already in the trainer's hands because the trainer is the one that produced it. This is how the environments in the previous guide work, and it is what TRL's `GRPOTrainer` expects when you hand it an environment factory.

In a black-box environment the agent owns it. A harness starts inside a sandbox, runs its own loop, calls its own tools, compacts its own context, and stops when it decides to. The trainer is outside that box. It sees one thing: a sequence of calls arriving at a model endpoint.

Microsoft's Agent Lightning team gave the two modes names, and the sentence is the cleanest statement of the problem I have found (He et al., 2026):

> *"In traditional agentic RL, the training engine owns the environment interaction loop. In harnessed agentic RL, the harness owns this loop, while the training engine observes only a sequence of LLM request-response pairs."*

The reason this is tractable at all is a single structural fact. Agents differ wildly on the inside, but every LLM-based agent has to talk to a model, and the model API is the one interface that is guaranteed to exist outside the agent. The policy is the model. Everything from the API boundary outward, harness state and environment state alike, is latent.

> *Note: the KAT-Coder report (KwaiKAT Team, 2026) uses white-box and black-box to mean something else, namely how a harness organises its trajectories. Mini-swe-agent is white-box, with a simple loop and no trajectory compression. Claude Code, Codex, OpenClaw and OpenHands are black-box, because they compress and reorganise their context. That is a property of the harness. The sense used here is a property of the training setup, meaning who owns the loop. The two often coincide and they are not the same axis.*

The previous guide already drew a version of this split, framed as whether the trainer or the environment drives the episode, and it worked through where each of six frameworks sits on it. Every one of them sat on the left-hand side.

What follows is that same axis pushed one step further out. There, the thing that might own the loop was another RL framework. Here it is an agent binary that was never written with training in mind.

Two things change when the loop moves, and they are worth separating because they have different consequences. The first is control: who makes each call, and in what order. Watch the tails of the arrows rather than the boxes.

Figure 11 · Who initiates each call, under both architectures.

The second is what the trainer is left holding afterwards. Control and visibility are not the same problem: a trainer could in principle be told about events it did not cause, and the reason it is

not is that the harness has no obligation to report them. Here is one rollout written out three times, once as it happened and once as each architecture records it.

What the trainer is left holding

The retry is the row to notice. The model emitted a malformed call, the harness quietly repaired it, and the black-box record has no idea it happened.

Figure 12 · One rollout, as it happened and as each architecture records it.

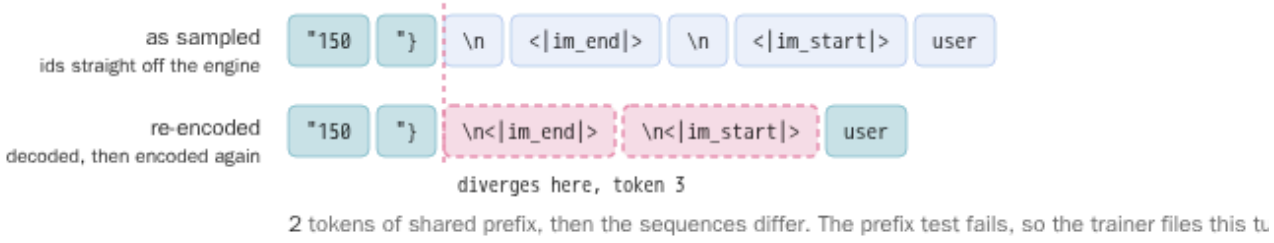
Why rewards are not enough

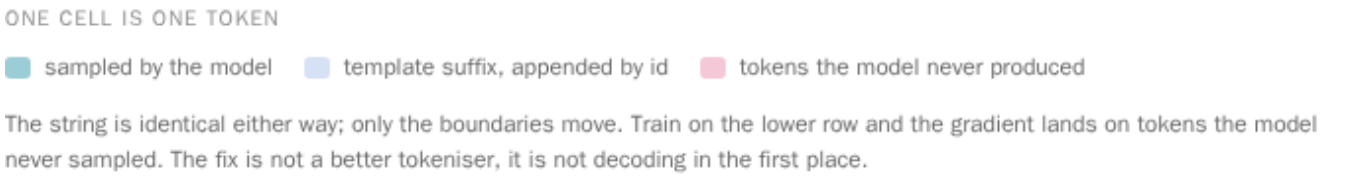
It is tempting to think a reward and the text of a trajectory should be sufficient, and it is worth being precise about why they are not. An on-policy policy gradient (Williams, 1992) needs two things per token: which token was sampled, and the probability it was sampled with. A harness that hands back text and a score gives you neither.

Re-tokenising the text guesses at the first, and it can guess wrong, because decoding is not injective. TRL's write-up on getting this right states the rule plainly: "in RL, you optimize on the exact tokens the model produced," and the failure is that "decode a sequence, re-encode the text, and you can land on different tokens," after which "the gradient ends up on a sequence the model never sampled" (Gallouédec & Rasul, 2026). Nothing errors. The loss just spikes now and then for no reason you can see.

The seam between turns is where this usually bites, because that is where a template appends something to what the model wrote. Tool-call serialisation changes whitespace on the way through, and some harnesses quietly repair malformed JSON, which hides the model's actual mistake from the reward. Recomputing log probabilities in the trainer gives you the current policy's numbers but not the sampler's, so the importance ratio silently defaults to one when it is not one, and an estimator you believe is on-policy is a biased off-policy estimator instead.

Why re-tokenising is a guess





The string is identical either way; only the boundaries move. Train on the lower row and the gradient lands on tokens the model never sampled. The fix is not a better tokeniser, it is not decoding in the first place.

Figure 13 · The same characters, tokenised two different ways.

Fixing either requires the sampler to hand back token ids and per-token log probabilities, and then never letting go of them: the rule is not to re-encode anything you decoded, but to keep the sampled ids and append the template's suffix by id concatenation. That is workable because the property it depends on, a chat template that extends token for token when a tool message is appended, is common rather than rare: eighteen of nineteen models tested satisfy it unmodified (Gallouédec & Rasul, 2026). It is also the real reason every serious multi-harness system intercepts at the model endpoint rather than at the text boundary.

There is one honest caveat to close on. Not everyone agrees the token-level machinery is necessary. OpenForgeRL builds the same proxy architecture and reconstructs training samples from the prompt and response pairs the proxy collects. The paper never mentions token ids or log probabilities, and it still reports the strongest multi-harness results published so far (Yu et al., 2026). This article takes the token-faithful side, and the question is not settled.

OpenEnv

OpenEnv is a standard interface for RL environments (Meta PyTorch & Hugging Face, 2025). It offers Gymnasium-style `reset`, `step` and `state` over a client and server transport, with typed actions and observations, packaged as Docker and publishable to the Hub. It began as a Meta PyTorch and Hugging Face collaboration and now lives at `huggingface/OpenEnv`, steered by a committee of twelve organisations, under BSD-3-Clause.

In this article OpenEnv is the shared interface that the harness, the environment and the trainer all connect to. It does not train anything or define rewards. Two pieces added in OpenEnv 0.5.0 make it possible to train inside a black-box harness. The capture proxy records what RL training needs from any agent that calls a model API. The Harbor integration (Kolavi, 2026) serves Harbor's containerised tasks as OpenEnv environments, with the harness and the sandbox chosen per rollout. Both install with `pip install "openenv[harbor]"`.

One rollout, end to end

The figure follows one rollout of a SmolDataEnvs task, three turns long, through every part of the system. Switching the harness changes only how Harbor wires it to the proxy, the format of its model calls and replies, and the names of its tools.

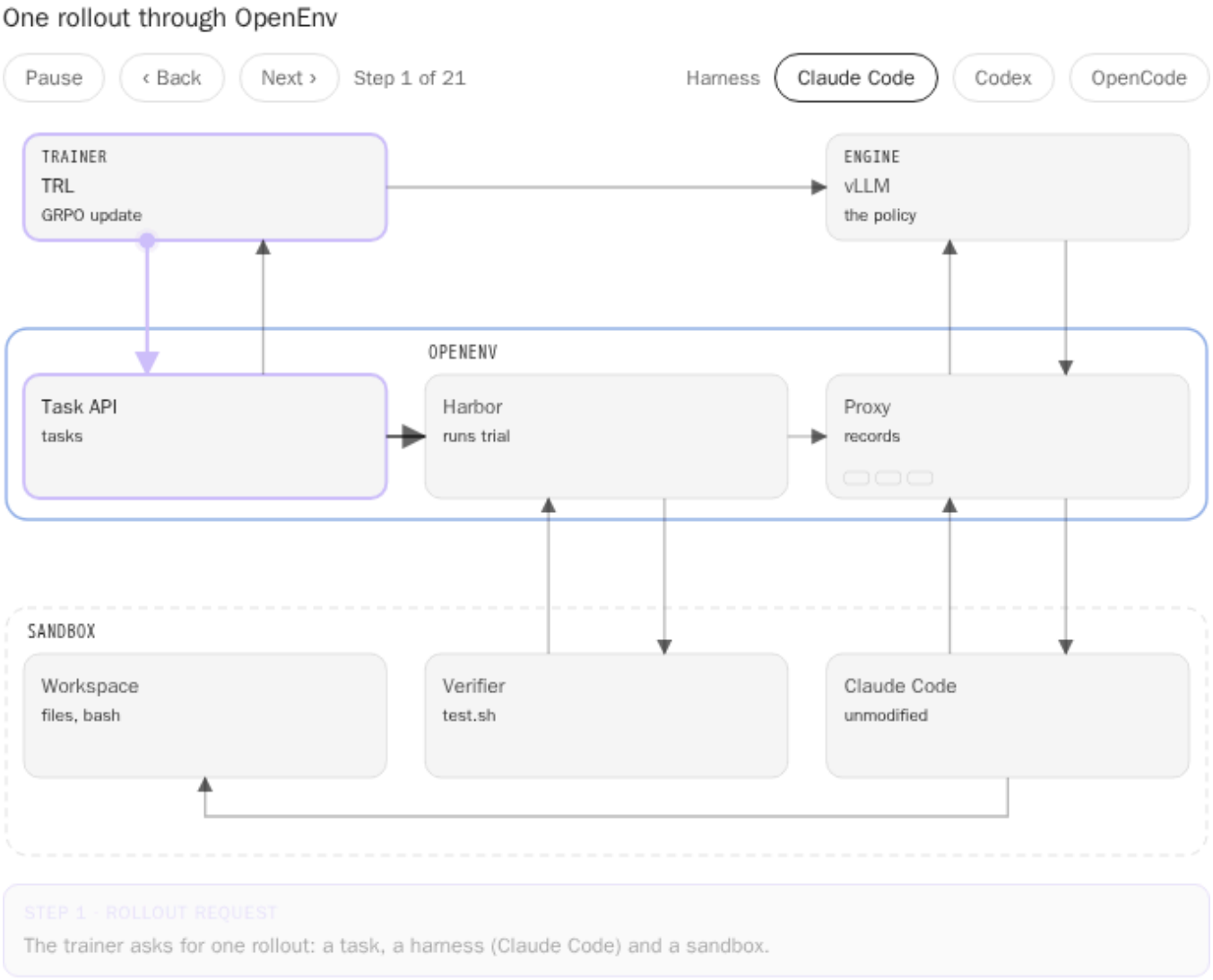


Figure 14 · One three-turn rollout, from the trainer's request to the weight update, step by step.

In our runs the trainer and vLLM (Kwon et al., 2023) had one GPU each, the OpenEnv server ran next to them or on a Hugging Face Space, and the Harbor runs used E2B sandboxes. The harness inside the sandbox never talks to vLLM or to the trainer. It reaches the model only through the capture proxy, and the only credential it holds for that is a session id.

The capture proxy

A harness finds the proxy the way it would find any model provider, through a base URL and an API key. The API key is a capture session id minted for that rollout, which is how one proxy on one port serves every concurrent rollout. A call carrying a key the proxy never registered gets a 401.

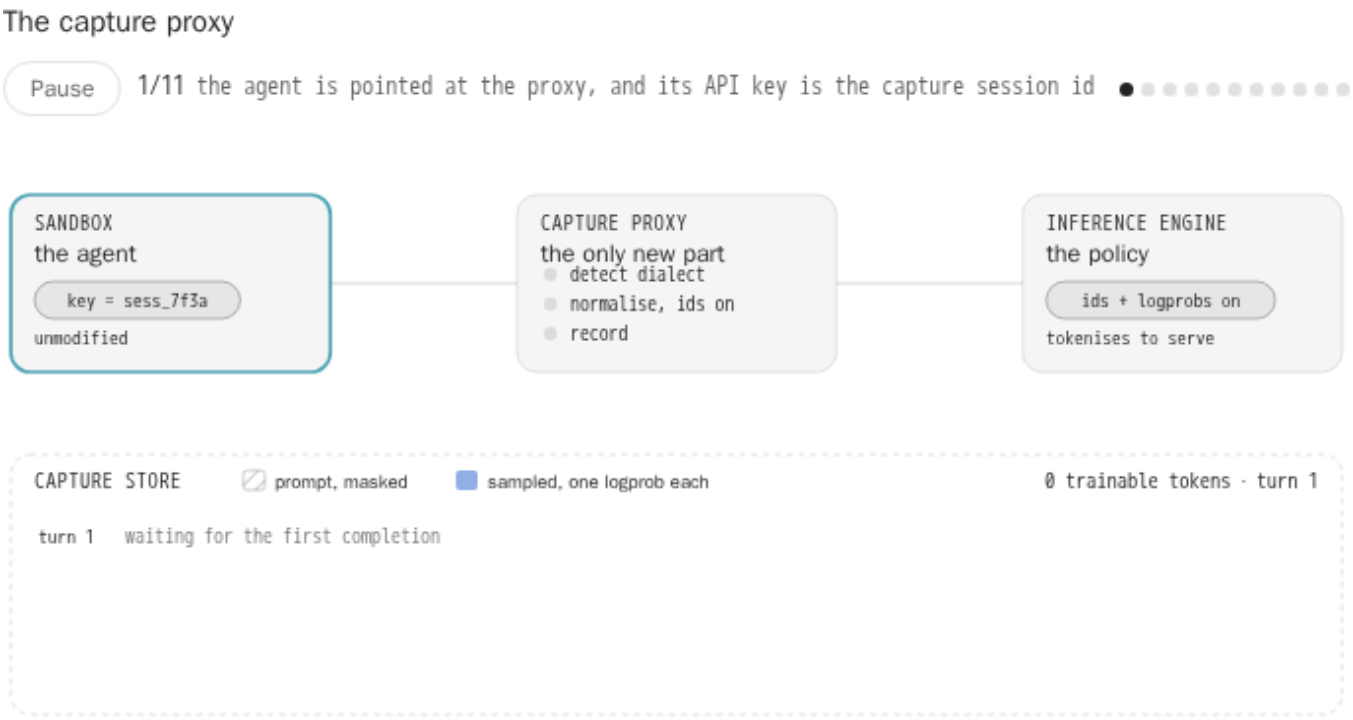


Figure 15 · One model call, step by step, through the capture proxy.

Harnesses speak four different model APIs, as the harness table in What is a harness? shows. The proxy works out which one a request uses from its path, then its headers, then the shape of its body. It converts the request to chat completions with converters vendored from NVIDIA's Polar gateway (Xu et al., 2026), and replays the answer in the caller's own format.

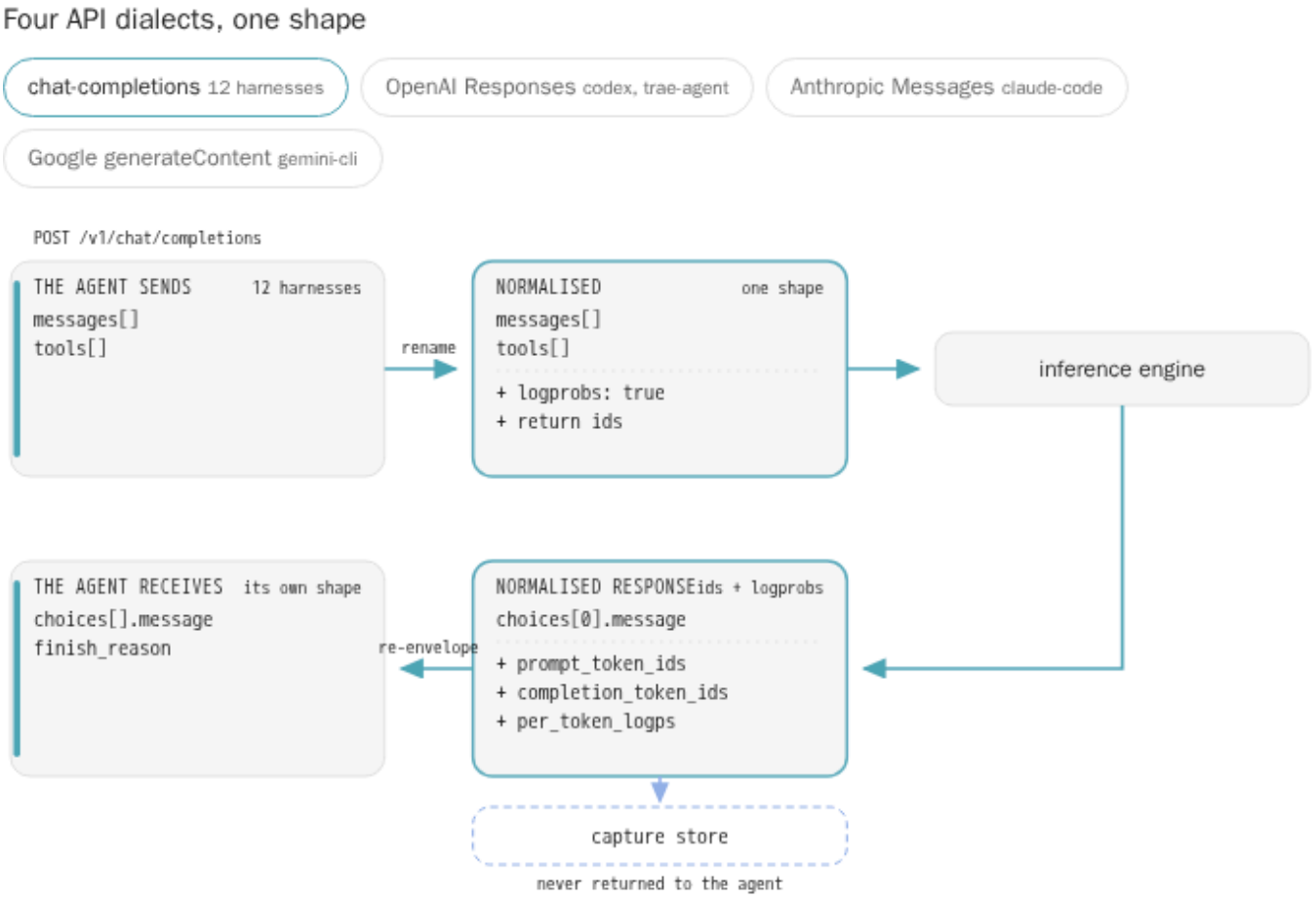


Figure 16 · The same request in the four formats coding agents speak, and the one shape the trainer sees.

The call to the engine never streams. The proxy waits for the whole completion, stores it, and replays it to the harness as a stream if the harness asked for one. Apart from a later first token, the harness cannot tell the difference, and the proxy never has to parse partial deltas.

WHAT GETS RECORDED

For each call the proxy asks the engine for the prompt's token ids, the sampled token ids and a logprob for every sampled token. It also switches off truncating sampling such as `top_p`, because vLLM's processed logprobs are computed after truncation. In our runs, moving `top_p` to 1.0 took the importance ratio from 0.985–0.993 to 0.9984–0.9999. On vLLM this needs two server flags:

```
vllm serve <model> --return-tokens-as-token-ids --logprobs-mode
processed_logprobs
```

SGLang works when built from its main branch. The v0.5.16 release cannot return the sampled token ids.

The proxy does not take any of this on trust. The first time it sees an engine it sends a probe and grades the result, and it also checks whether the logprobs are raw or processed. An engine below the `tokens` level still serves rollouts, but they are marked as evaluation only, and asking for training data from one raises an error. Hosted APIs such as OpenAI, Anthropic and Hugging Face Inference Providers land there, so they can evaluate a harness but not train through it.

Three levels of capture

`tokens`	prompt ids, sampled ids, aligned logprobs	trainable
`logprobs`	logprobs, but no ids to attach them to	eval only
`text`	neither	eval only

An endpoint missing the token-id flags still returns text, a 200 and plausible usage numbers, so the proxy does not take the configuration on trust. It probes each engine the first time it sees it and grades it. Anything below tokens still runs, but only for evaluation, and asking it for training data raises an error.

Figure 17 · Only one of them can be trained against.

FROM CALLS TO TRAINING SEQUENCES

A rollout is rarely one clean conversation. Harnesses retry, spawn subagents and compact their context. The proxy stores each model call as a node and links it to the earlier call whose prompt and completion form the longest exact token prefix of its own prompt. A retry shows up as a sibling that never continued. A subagent has its own system prompt, so its first call extends nothing and starts a new root, and a compacted context does the same.

Figure 18 · A trace as a graph of model calls, and its trainable branches.

Each path from a root to a leaf becomes one training sequence. Context tokens get a loss mask of 0 and sampled tokens get 1. A turn whose logprobs are missing or misaligned stays in the sequence as context and is never used as a target. These checks run as each call arrives rather than at export, because a misaligned turn is easiest to diagnose while the proxy still knows which turn it was.

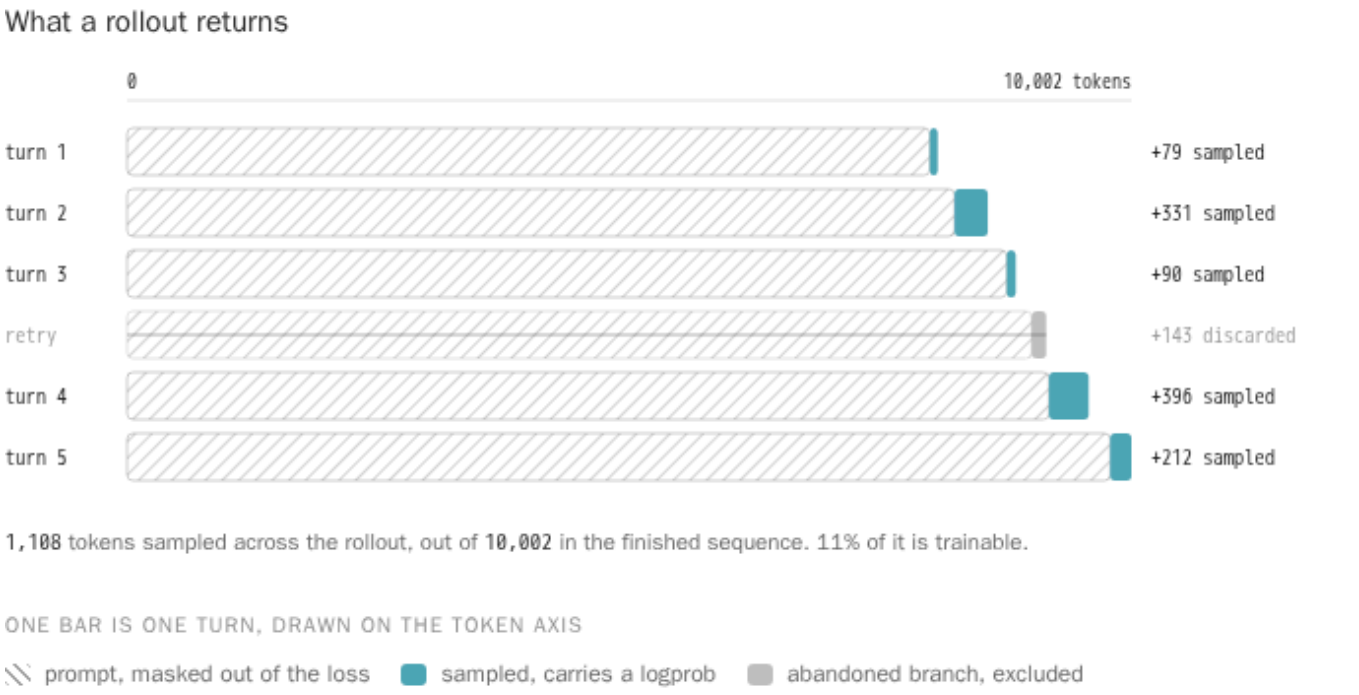


Figure 19 · What one rollout hands back, turn by turn.

CAPTURING YOUR OWN HARNESS

Nothing in the proxy is specific to Harbor, so any program that calls one of the four APIs can be captured. Start the proxy, mint a session with `POST /sessions`, give your agent the proxy's URL as its base URL and the session id as its API key, and read the rollout back from `GET /sessions/{id}/rollout`.

```
python -m openenv.core.harness.capture.server --llm-url
http://127.0.0.1:8000 --model <model>
```

The proxy binds to localhost by default. On any other interface it needs an admin key, which guards the routes that mint and read sessions.

The proxy is a single process. In our runs its health check started to starve at around 200 concurrent sessions, and the process crashed at 320. A rollout that hits its model-call budget is also stopped by the proxy itself, so that final reply is never part of the training data.

Harbor

Harbor, from the team behind Terminal-Bench (Merrill et al., 2026), runs agents against containerised tasks and keeps the task, the harness and the sandbox independent of each

other. Version 0.22.0 ships adapters for more than 40 agent harnesses and 26 execution backends, from local Docker to Daytona, Modal and E2B, and about 80 task datasets already use its format, including Terminal-Bench and SWE-bench (Jimenez et al., 2024). Any harness can attempt any task on any backend, and a run is one command that names the dataset, the agent and the backend.

A Harbor task is a directory with an instruction, an environment and a test script. The ones below come from SmolDataEnvs, the task suite used for every run in this article: data-analysis questions built from real Kaggle notebooks in the jupyter-agent dataset, each with an automatic grader, published in the SmolDataEnvs collection. Pick a task and a file to read it as the agent and the verifier see it.

Three SmolDataEnvs tasks, file by file

Easy · level 1 | Medium · level 2 | Hard · level 4

What percentage of all matches have a goal difference of zero (i.e., draws)? Gold answer: `25.4%` · data from `hugomathien/soccer`

tasks/0000_369_369503_qa_1/

- task.toml — the spec
- instruction.md — the prompt
- environment/
  - Dockerfile — the image
  - pull_bucket.py — the data
- tests/
  - test.sh — the verifier
  - grader.py — the reward

`instruction.md` The prompt. Exactly what the agent is shown: where its files are, what is installed, and how to write the answer. — this task only

```
1  You are a data-analysis agent working in a sandbox. Use your code-
     execution tool to inspect the files and compute the answer.
2
3  Files (in /home/user/input, no subfolders):
4  - database.sqlite
5
6  Installed: pandas, numpy, matplotlib, seaborn, scipy, scikit-learn,
     statsmodels, tabulate, sqlite3, plotly (pip install more if
     needed).
7
8  Question:
9  What percentage of all matches have a goal difference of zero (i.e.,
     draws)?
10
11 Work it out step by step - inspect the data first (head, shape,
     dtypes), then compute.
12
13 Express the value as a percentage (e.g. 95.5), not a fraction.
14
15 Answer with a single clean value: a bare number (no commas or units.
```

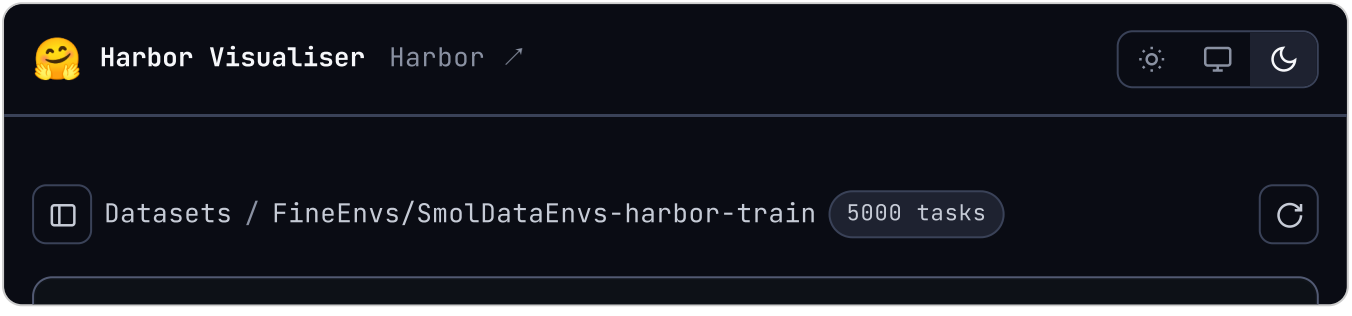
HOW THE REWARD GETS OUT

agent writes `/workdir/answer.txt` → `test.sh` reads it → `grader.py` prints one float → lands in `/logs/verifier/reward.txt` → Harbor returns it with the trial

Real files from FineEnvs/SmolDataEnvs-harbor-train, shown unedited. The gold answer is highlighted where `task.toml` hands it to the verifier.

Figure 20 · The six files of a Harbor task, taken unedited from the training set, and how the reward gets out.

The task names no harness, sandbox backend or trainer, because those are chosen when it runs. The Harbor visualiser opens the whole training set the same way, all 5,000 tasks.



> *Note: Harbor calls its sandbox backends environments. In this article, sandbox means the box the agent runs in, and environment means the OpenEnv server.*

Serving Harbor through OpenEnv

Harbor already runs an agent against a task and returns a score. A trainer also needs the tokens the policy generated and their probabilities. Harbor's RL documentation lists two ways to get them: intercept the tokens at the inference server, or have the harness return them with its result. The second needs a harness that puts its tokens in the result metadata, and the docs say that support is still being added to Terminus 2, Harbor's own harness. The integration takes the first, through the capture proxy, so it works with any harness that calls a model API.

The trainer talks to the environment over HTTP instead of running rollouts in its own process. An earlier in-process version let an exception from inside someone else's agent reach the training loop, where one crashed rank left the others waiting at the NCCL barrier indefinitely. Now nothing raises across that boundary. A failed rollout comes back with `ok=False` and `reward=None`, and the trainer handles a value instead of catching an exception.

THE FOUR COMMANDS

Command	
`openenv harbor info`	Reports what this machine can run: whether the engine returns token
`openenv harbor rollout`	Runs rollouts without an environment server. If a rollout works here an
`openenv harbor serve`	The environment server: a Task API, a `run_rollout` tool over MCP, a
`openenv harbor push`	Deploys the same server to a Hugging Face Space, with configuration

```
openenv harbor info    --llm-url $LLM --dataset org/train,org/eval
openenv harbor rollout --llm-url $LLM --dataset org/train --task-index 0 -n 5 --harness codex --sandbox e2b
openenv harbor serve   --dataset org/train,org/eval
openenv harbor push    --llm-url $LLM --dataset org/train,org/eval --repo-id you/harbor-env
```

`rollout` requires `--llm-url` and has no default, because a stale endpoint produces rollouts that look fine and carry no token ids. `serve` can start without one, because each rollout

names its own engine. That let one deployment serve both training, pointed at the trainer's vLLM, and evaluation. A rollout prints one line when it finishes:

```
[opencode / e2b] task 0: 0000_369_369503_qa_1 ...
   ok    reward=1.00  turns=9  roots=2  multi-turn  tokens=1043
atif=match  48s

```

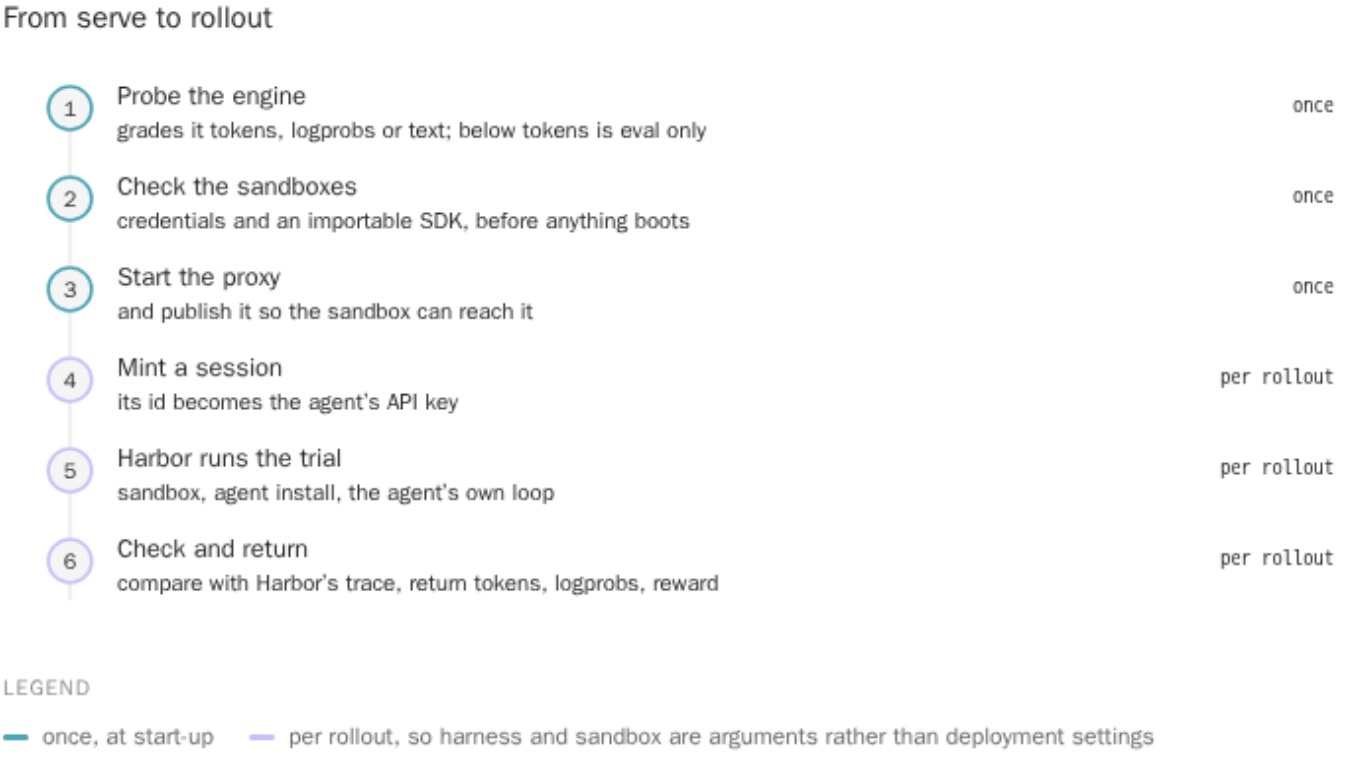
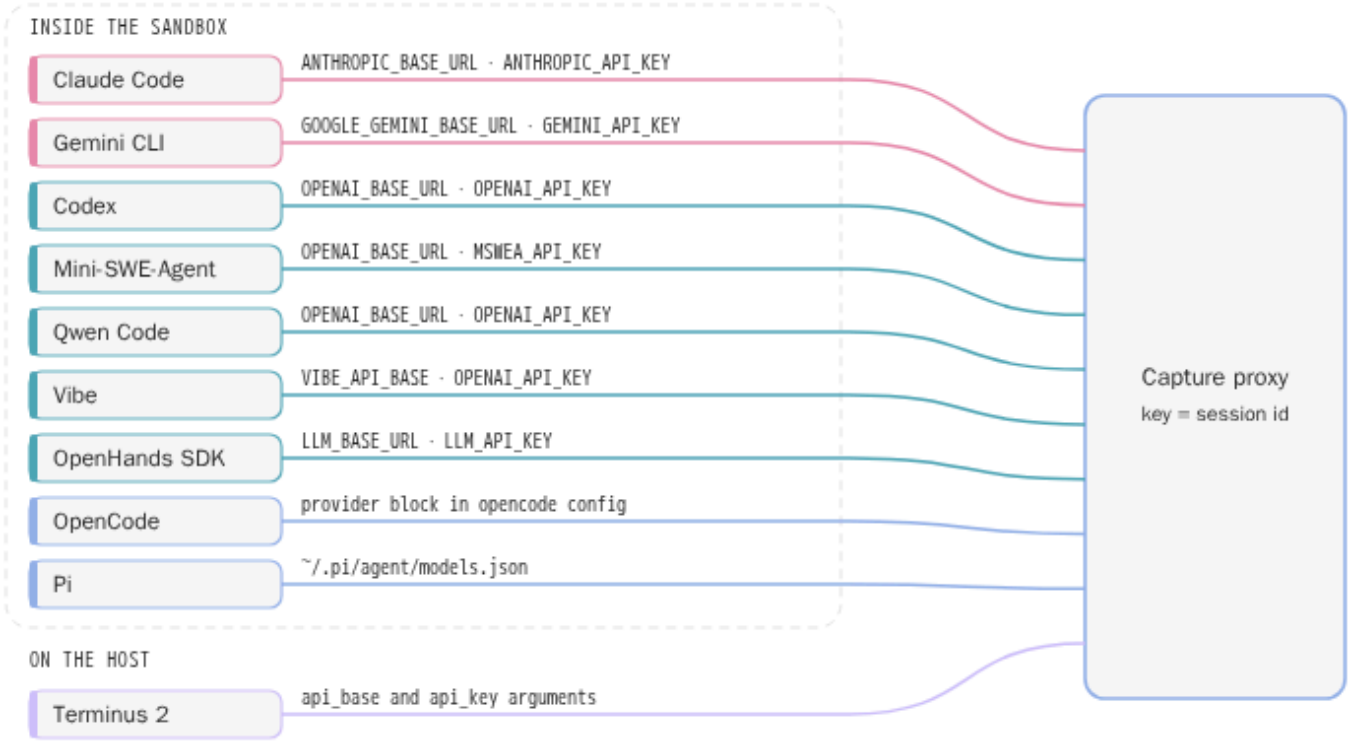


Figure 21 · What happens once at start-up, and what happens on every rollout.

WIRING EACH HARNESS

Harbor already knows how to install and launch each harness. OpenEnv adds only the place each one reads its model URL and API key from, kept as one table entry per harness, so supporting a new harness means adding an entry rather than writing a new environment. Ten harnesses have passed the capture contract and the trace check described below, across 16,000 rollouts on the SmolDataEnvs test set.

How each validated harness reaches the proxy



These are the ten harnesses marked validated in OpenEnv 0.6. Another 19 are listed but not validated: 2 unstable, 3 unsupported, 3 blocked and 11 untested.

Figure 22 · Where each harness gets the proxy's URL and its session id from.

Most harnesses read environment variables, and OpenEnv sets those for the agent's run only. The session id Codex receives as `OPENAI_API_KEY` therefore never reaches the verifier, which may need a real key of its own. Claude Code and Gemini CLI also read the variables in the server process, so they get them there too. OpenCode and Pi take a config file written into the sandbox, and Terminus 2 runs on the host and is handed the URL and key directly.

PICKING THE REWARD, AND CHECKING THE CAPTURE

Harbor's verifier can return several named scores, and GRPO (Shao et al., 2024) needs one number per rollout. The integration uses the only score if there is one, or the one named `reward`, and otherwise asks the caller to choose. It never combines scores itself, because whatever weighting it picked would become the training objective. An earlier run had a `+0.2 for submitting anything` term. The policy learned to submit immediately, and held-out accuracy fell from 0.740 to 0.178 while training reward still looked healthy. When the verifier never runs, the reward stays `None` all the way to the trainer, so a sandbox that died is not scored as a wrong answer.

The proxy's own checks can only look at data it captured. As an independent check, each rollout is also compared with the trajectory file Harbor writes itself, in its ATIF format, call by call: the number of turns, the completion tokens per call, and which calls count as agent steps. Any mismatch fails the rollout.

```
ATIF completion_tokens : [37, 36, 104, 264, 255, 119, 32, 27]   total 874
intercept turn_lengths : [37, 36, 104, 264, 255, 119, 32, 27]   total 874
ATIF step-1 prompt_tokens 7990  ==  intercept prompt_len 7990

```

This caught a harness that sent an empty tools array, got a 400 from the inference server and was cut short, leaving a rollout graph that looked well formed. The proxy now drops empty tools arrays before forwarding.

DEPLOYING IT AND TRAINING AGAINST IT

On one machine, `serve` listens on two ports, one for trainers and the web UI and one for the capture proxy. Sandboxes usually run somewhere else, so the proxy reaches them through a tunnel, set with `--expose` and Gradio by default. A Hugging Face Space has only one port, so there the proxy is mounted at `/capture`. The Space has to be public, because the agent in the sandbox cannot send the auth header a private Space requires. It stays safe because the proxy only answers registered session ids, and minting one needs an admin key. Before training, raise the concurrency limit. `serve` allows four environments at once by default and rejects the rest with `CAPACITY_REACHED`, fewer than the eight rollouts in each of our GRPO groups.

The trainer side lives in OpenEnv's `harbor_env` package. Its `HarborSessionFactory` produces sessions in the shape TRL's `HarnessRolloutWorker` expects when the agent owns the loop, a worker added in TRL pull request #6947, which is still open. Each session requests one rollout, waits for it, and returns the captured turns as TRL trace entries. In our runs every rollout in a GRPO group used the same harness, so the advantage compared actions rather than harnesses. The sampling temperature sent with each rollout also matched the one the trainer used to recompute logprobs. The training chapter covers the training loop.

Training small models across harnesses

We trained two small open models with the setup from the previous chapter: Qwen3.5-2B first, then LFM2.5-2.6B. Most of this chapter is about LFM, where both the scores and the tool use

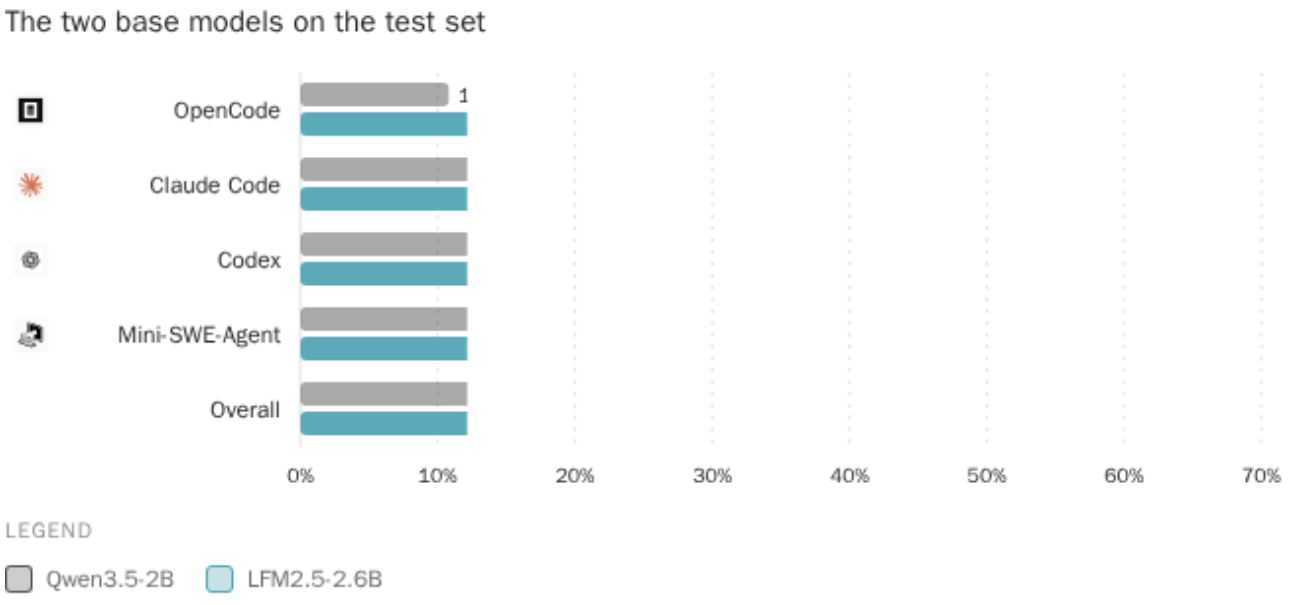
moved the most. The Qwen runs, and what they taught us, are in a collapsible section at the end.

Setup

- Tasks. 1,000 SmolDataEnvs training tasks, 400 medium and 600 hard, in one fixed order shared by both runs. The test set is 250 held-out tasks with no overlap in notebook, question or instruction.
- Evaluation. Every 100 steps, each test task runs once under OpenCode, Claude Code, Codex and Mini-SWE-Agent, 1,000 cells in all. A cell counts as solved when its first graded attempt is correct.
- Two runs. One trains in OpenCode only. The other is multi-harness, where each GRPO (Shao et al., 2024) group of eight rollouts uses one of the four harnesses.
- Recipe. Async GRPO in TRL for 1,000 steps. One H100 trains, a second serves the policy with vLLM (Kwon et al., 2023), and the tasks run in E2B sandboxes.

Each cell runs once, so a point or two between neighbouring checkpoints is noise. Trust the trend across checkpoints over any single one.

Where the base models start



Pass@1 on the 250 test tasks under each harness, before any training. LFM's baseline has 998 of 1,000 cells graded.

Figure 23 · Pass@1 by harness before any training.

LFM starts almost three times higher than Qwen. It also shows the problem this article is about before any training. The same weights solve 62% of tasks under Mini-SWE-Agent and 33% under Claude Code.

What we rewarded

For Qwen, the reward was the verifier's verdict alone, 1 for a correct answer and 0 for a wrong one. It ignores how the model got there. A correct answer after 30 tool calls scores the same as one after 3, so nothing pushes the model to stop exploring once it has the answer, and in an early Qwen run the tool calls per rollout crept from 13 to 41.

For LFM we kept correctness as the main signal and added a small bonus for correct answers that take fewer calls. A wrong answer still scores 0, however few calls it made, so the model cannot earn the bonus by giving up early.

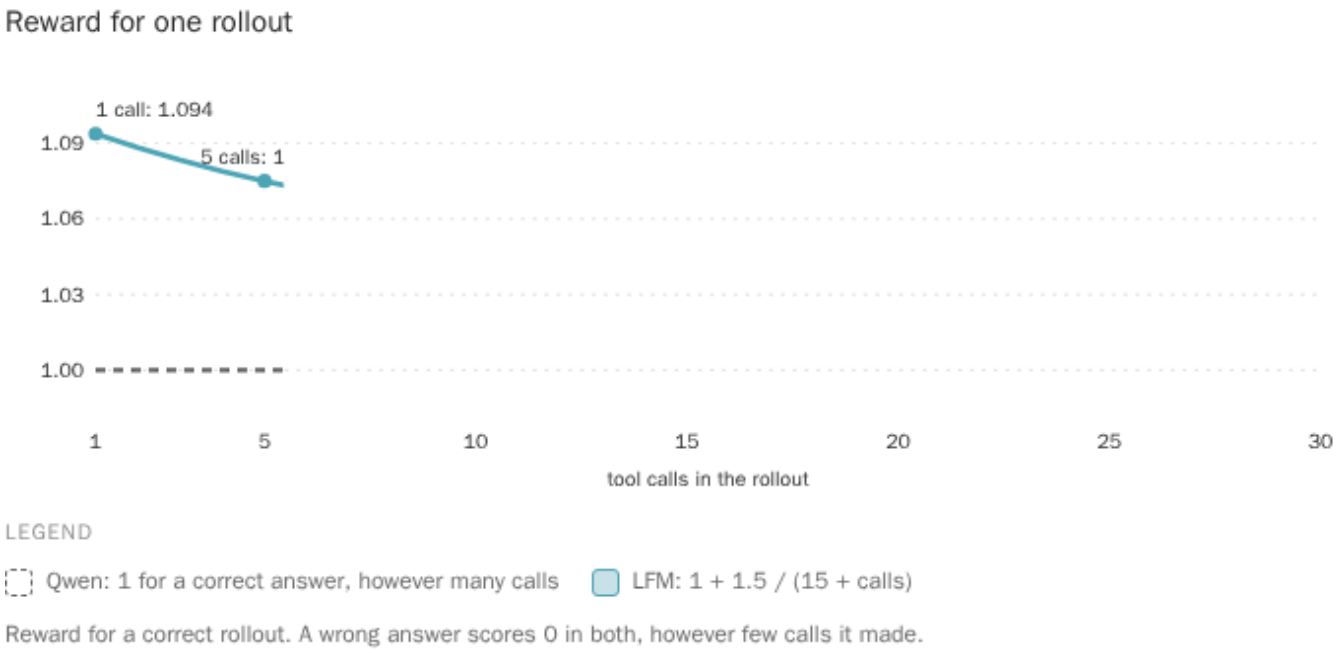
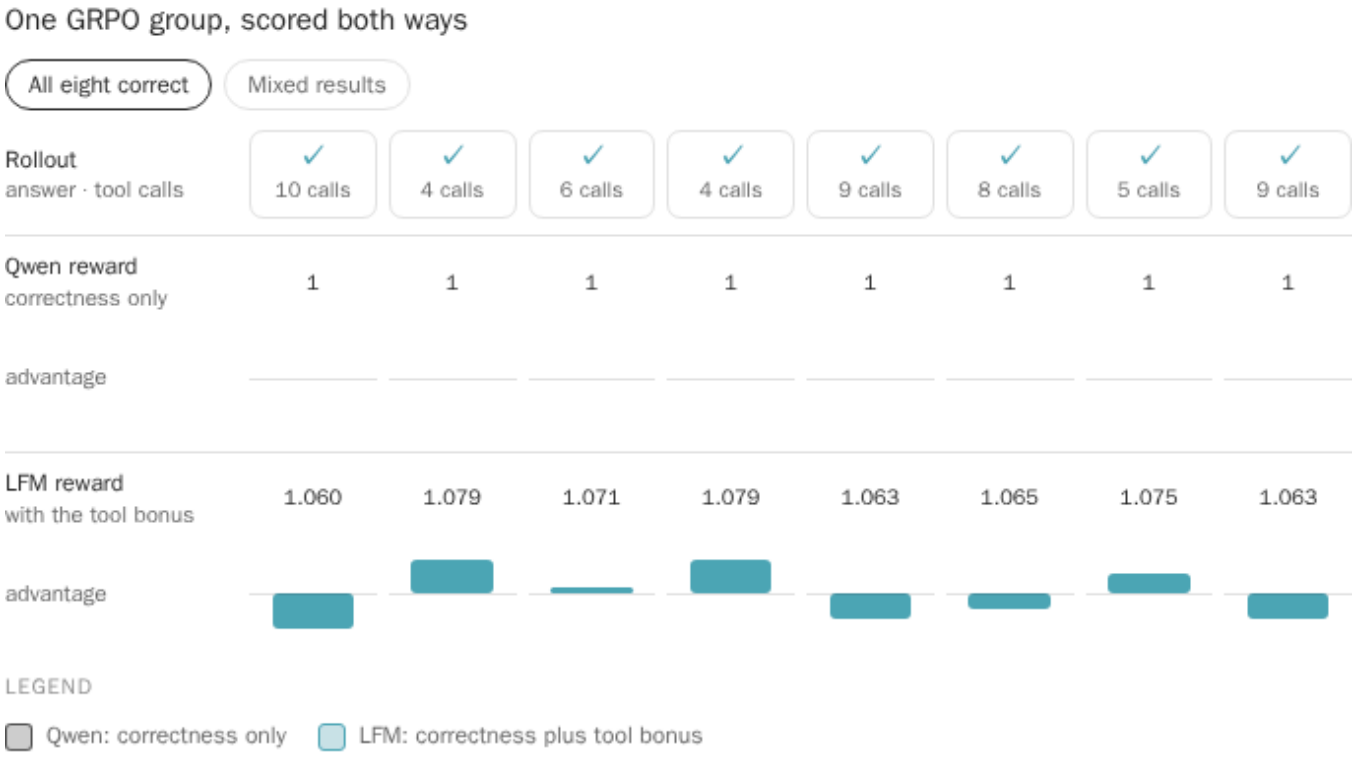


Figure 24 · Correctness alone for Qwen, and correctness with a tool-call bonus for LFM.

The bonus is at most 0.1, but GRPO learns from differences inside a group, so even a small bonus counts. GRPO runs eight rollouts of the same task and trains on how each one compares with the group's average. When all eight are correct, correctness alone makes them identical and the group teaches nothing. The bonus is then the only thing that tells them apart, and it favours the shortest solutions. Here are two real groups from the multi-harness run, scored both ways.



LEGEND

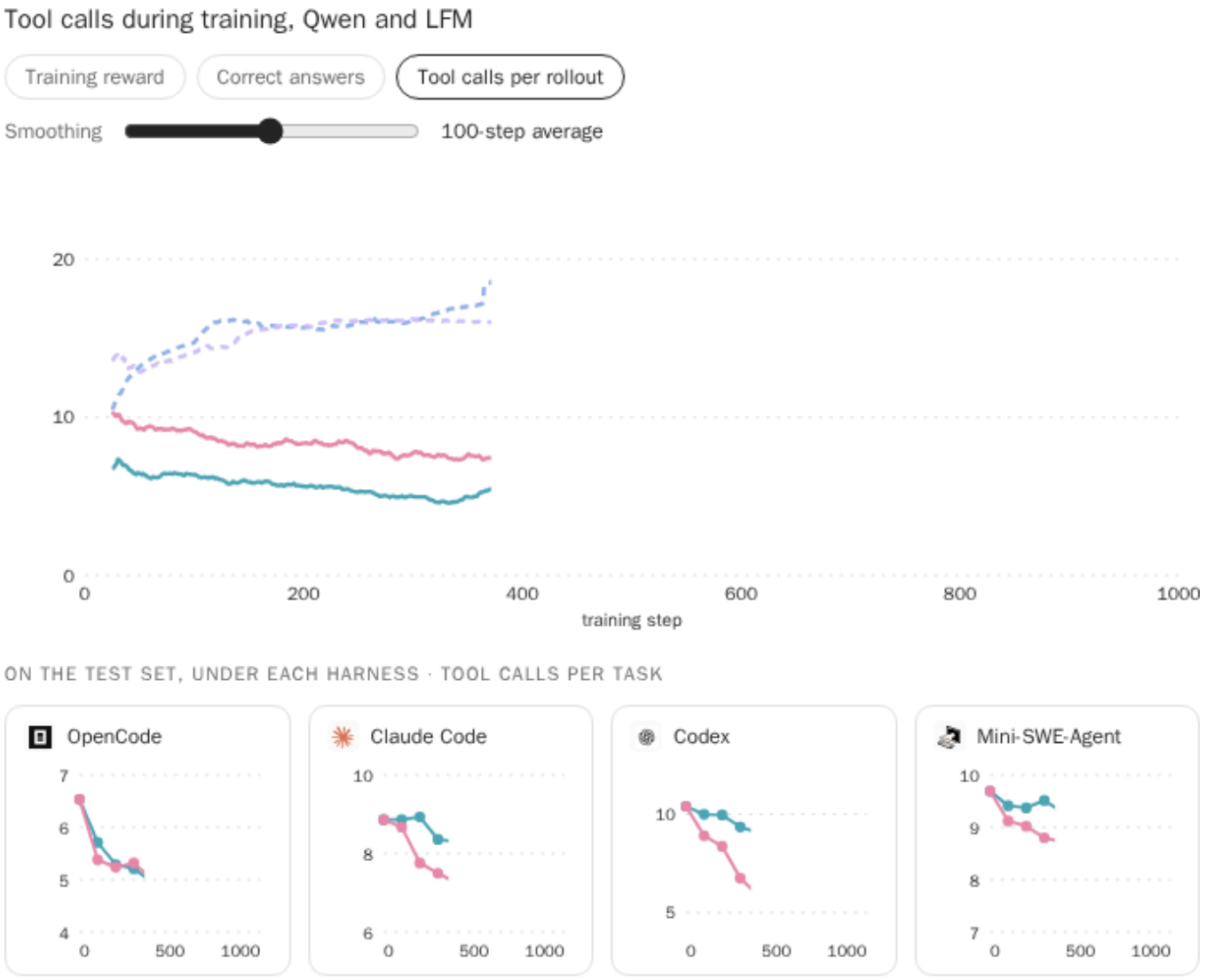
Qwen: correctness only LFM: correctness plus tool bonus

All eight solved this hard Claude Code task (training step 80). With correctness alone every rollout scores 1, so every advantage is zero and the group teaches nothing. With the bonus, the rollouts that needed 4 calls are pushed up and the one that needed 10 is pushed down. Advantage is how far each reward sits from the group's average, in standard deviations. It is what GRPO trains on.

Figure 25 · Eight real rollouts of one task, with the reward and advantage each would get under the two rewards.

All-correct groups like the first one made up 97 of the OpenCode-only run's groups and 155 of the multi-harness run's. The call count comes from the harness's own trace, and an audit of every training rollout found the bonus applied exactly as written and never to a wrong answer.

During training, tool calls per rollout in the Qwen OpenCode-only run doubled from about 10 to 20, and the Qwen multi-harness run stayed between 11 and 16. LFM's fell in both runs and under every harness.



Mean tool calls per rollout at every optimizer step, smoothed with a trailing average. The small charts show the same runs on the held-out test set, evaluated under all four harnesses every 100 steps.

Figure 26 · Every optimizer step for the two Qwen runs, trained on correctness alone, and the two LFM runs, trained with the bonus.

The models differ in other ways too, and there was no LFM run without the bonus, so we cannot say the bonus caused it on its own.

Training curves

Mean training reward at every optimizer step, smoothed with a trailing average. LFM's reward includes the tool bonus, so it sits slightly above its correctness. With heavier smoothing, a faint 10-step line shows the noise behind it. The small charts show the same runs on the held-out test set, evaluated under all four harnesses every 100 steps.

Figure 27 · Reward, correct answers and tool calls at every optimizer step. Drag the slider to change the smoothing.

The share of correct rollouts climbs over the first few hundred steps in both runs, then rises and falls with the difficulty of the tasks in each stretch, since both runs follow the same task order. Tool calls fall throughout. On the test set, the OpenCode-only model ends at 58% under OpenCode against 50% for multi-harness, and the multi-harness model ends at 49% under Claude Code against 42%.

Accuracy

Top: all 1,000 test cells. Below: the 250 cells of each harness. Hollow points are checkpoints where a few cells were never graded.

Figure 28 · Pass@1 at every evaluated checkpoint, overall and under each harness.

Both runs climb, by 10 points for OpenCode-only and 12 for multi-harness, most of it in the first 200 steps, and hold it to step 1,000. They end at 52.3% for OpenCode-only and 54.2% for multi-harness. Both gains are well outside the noise. The 1.9-point gap between the runs is within the noise, so overall accuracy does not separate them.

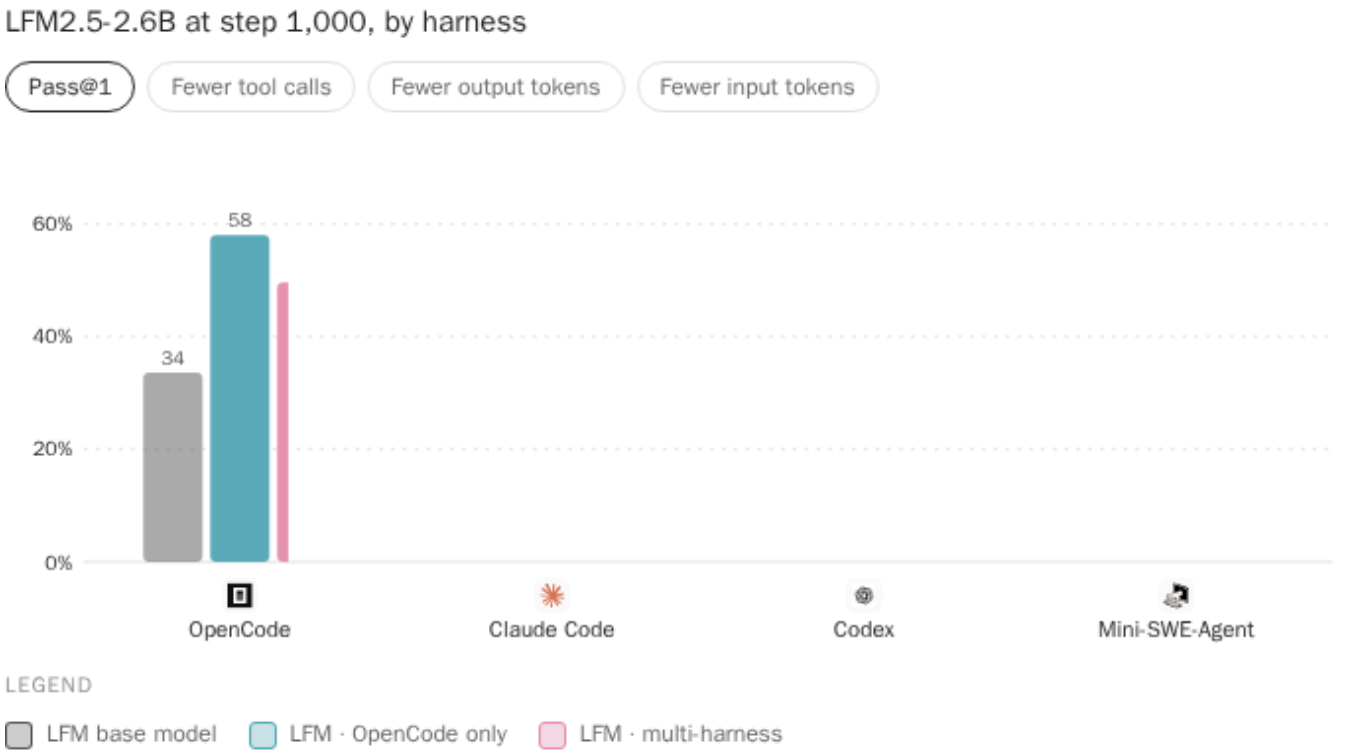


Figure 29 · Pass@1 by harness next to the base model, or the change in tool calls and tokens on tasks both solved.

The per-harness split is the clearest result. The OpenCode-only model is best under OpenCode, the harness it trained in. The multi-harness model is better under Claude Code and Codex, and the two tie under Mini-SWE-Agent. Most of the OpenCode-only model's gain is in OpenCode (34 → 58%), while the multi-harness model gained under all four.

Tool calls and tokens

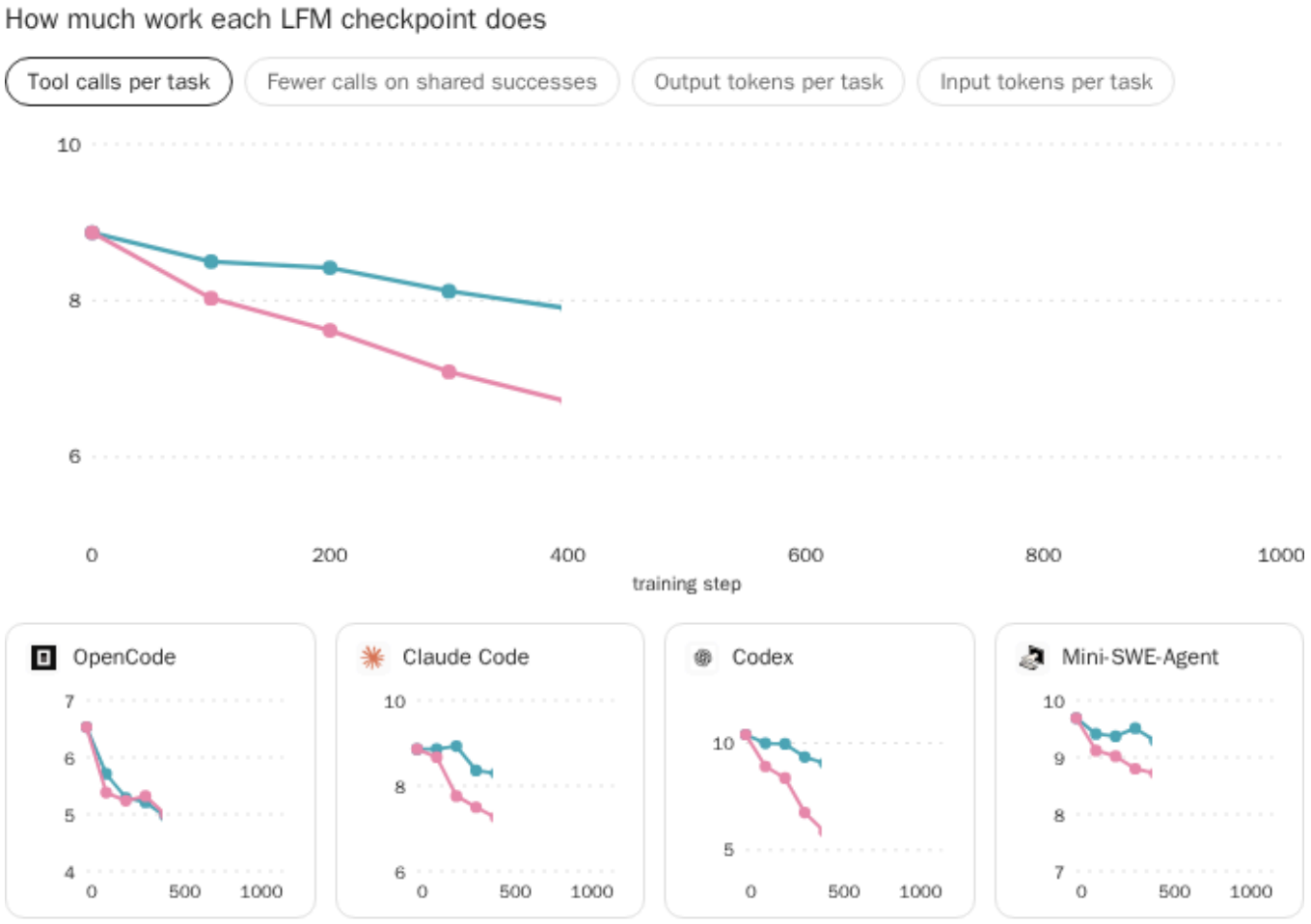
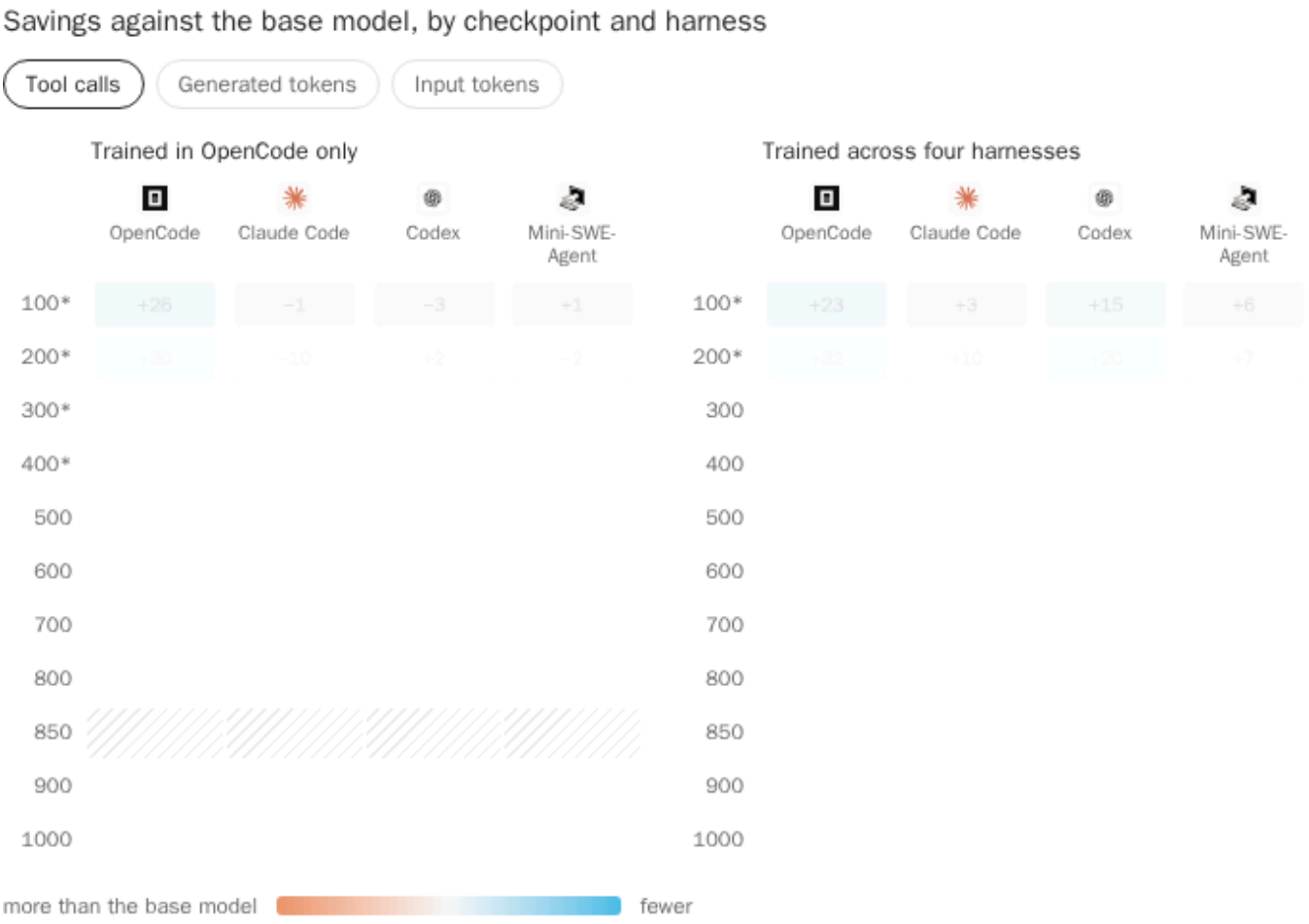


Figure 30 · Tool calls and tokens per test task at every checkpoint, overall and under each harness.

Both models solve more tasks with fewer tool calls, and the multi-harness model cuts more, about a third fewer calls than the base model on the tasks both solved, against about a tenth for OpenCode-only. Fewer calls also means less history resent on every call, so input tokens fall with them. The biggest saving is under Codex, where the multi-harness model uses about half the calls. The clear exception is OpenCode-only under Claude Code, a harness it never trained in, where it used more calls and more tokens than the base model.



Each cell compares a checkpoint with the base model on the test tasks both solved under that harness. Blue means fewer tool calls, orange means more. * marks an evaluation that was still missing a few cells.

Figure 31 · Percent fewer tool calls, generated tokens or input tokens than the base model, on the test tasks both solved. Hover a cell for its task count.

The multi-harness model's Codex savings grow at almost every checkpoint, and by step 1,000 every harness is blue on all three measures. The OpenCode-only model saves steadily in OpenCode and more and more in Codex. Under Claude Code, it generated more tokens than the base model at nearly every checkpoint, up to 62% more at step 500.

How much each run saw

Both runs took 1,000 optimizer steps, but they did not see the same amount of data.

	OpenCode only	Multi-harness
Distinct training tasks	555	626
Supervised tokens	12.5M	18.7M
Tokens processed in training	162M	451M

Claude Code rewrites its own history as it goes, so each of its rollouts became about eight training rows, each carrying its own copy of the context, while the other harnesses averaged about one. With one seed per run and unequal exposure, the comparison between the two runs is observational.

The Qwen3.5-2B runs, and what they taught us

Reproducing it

- Training comparison dashboard: every training curve and checkpoint evaluation.
- Harbor environment and standalone OpenCode environment: the environment servers used in training.
- OpenEnv #1036 for the capture proxy and Harbor integration, and TRL #6947 for the trainer side.

> **Bigger runs are coming**
>
> Every run in this article uses a small model, 2 to 2.6 billion parameters, trained for 1,000 steps on a single seed. We are now setting up larger runs with bigger models on the same stack, and we will add the results here when they are in. Stay tuned.

Conclusions

Results as of the run audit on 24 September 2026.

The question behind this work was whether you can take a small open model and train it inside the harnesses people actually use, Claude Code, Codex, OpenCode, without changing them. With the setup in this article, you can. The capture proxy sits between the harness and the model, records the exact tokens, and hands the trainer something it can learn from. Harbor supplies the tasks and the sandboxes. Pick a model, pick a harness, pick a sandbox, and train.

The harness matters from the very first step. Before any training, LFM2.5-2.6B solved 62% of our test tasks under Mini-SWE-Agent and 33% under Claude Code, with the same weights. If

you ship an open model and evaluate it in one harness, you are describing one point on that spread.

Gains follow the training harness. The LFM runs gained ten and twelve points. The model trained only in OpenCode ended strongest in OpenCode. The model trained across four harnesses ended stronger under Claude Code and Codex. It is slightly ahead overall, by less than the noise, and its gains are spread across more of the harnesses people use.

The reward shapes tool use. Rewarding only the final answer left Qwen with no reason to stop exploring, and a lot of our Qwen compute went into steps where every rollout scored the same. A small bonus for solving tasks in fewer tool calls changed LFM's behaviour. The multi-harness model solved more tasks while making about a third fewer calls on the ones it already could, and both runs kept learning from groups where every rollout was already correct.

Exact tokens are only the starting point. They make the update match what the model generated. The Qwen runs had exact tokens and still peaked and fell back, one while its answers grew past what evaluation allowed and the other while it kept working without submitting. Output budgets and reward design shaped how those runs ended.

Steps are a poor unit of comparison. Two runs with the same 1,000 steps saw different numbers of tasks and processed nearly three times as many tokens, in part because of how one harness rewrites its history. Any comparison between harness mixes has to account for that.

The next runs we want are an LFM run without the efficiency bonus, several seeds per setup, harness mixes compared at equal token exposure, and larger models. All of it runs on the same open stack, so if you have a harness, a task set or a trainer of your own, you can plug it in and try.

Citation

For attribution in academic contexts, please cite this work as

```
Adithya S Kolavi (2026). "The ultimate guide to multi-harness RL".
```

BibTeX citation

```
@misc{kolavi2026_the_ultimate_guide_to_multi_harness_rl,
  title={The ultimate guide to multi-harness RL},
  author={Adithya S Kolavi},
  year={2026},

}
```

1. DeepSeek-AI. (2025). DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. *arXiv Preprint arXiv:2512.02556*. https://arxiv.org/abs/2512.02556↑

2. Deng, X., Da, J., Pan, E., He, Y. Y., & others. (2025). SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? *arXiv Preprint arXiv:2509.16941*. https://arxiv.org/abs/2509.16941↑

3. He, Z., Zhang, S., Zhou, Z., Yang, Y., Kang, Y., Zhang, Y., Qiu, L. K., Tsui, T. Y., Xu, J., & Luo, C. (2026). Agent Lightning v1.0: Towards Harnessed Agentic RL. *arXiv Preprint arXiv:2608.17528*. https://arxiv.org/abs/2608.17528↑

4. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *The Twelfth International Conference on Learning Representations*. https://arxiv.org/abs/2310.06770↑

5. Kimi Team. (2025). Kimi K2: Open Agentic Intelligence. *arXiv Preprint arXiv:2507.20534*. https://arxiv.org/abs/2507.20534↑

6. Kimi Team, Bai, T., Bai, Y., Bao, Y., & others. (2026). Kimi K3: Open Frontier Intelligence. *arXiv Preprint arXiv:2607.24653*. https://arxiv.org/abs/2607.24653↑

7. Kolavi, A. S. (2026). *Harbor integration: serve Harbor task datasets through OpenEnv as trainable environments*. https://github.com/huggingface/OpenEnv/pull/1036↑

8. KwaiKAT Team. (2026). KAT-Coder-V2.5 Technical Report. *arXiv Preprint arXiv:2607.05471*. https://arxiv.org/abs/2607.05471↑

9. LLM-Core Xiaomi. (2026). *MiMo-V2.6: Scaling Reinforcement Learning Towards Self-Improvement*. https://huggingface.co/XiaomiMiMo/MiMo-V2.6-Pro-RL/blob/main/MiMo_V2_6_technical_report.pdf↑

10. Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., & others. (2026). Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv Preprint arXiv:2601.11868*. https://arxiv.org/abs/2601.11868↑

11. Meta PyTorch, & Hugging Face. (2025). *OpenEnv: Agentic Execution Environments*. https://github.com/huggingface/OpenEnv↑

12. MiniMax. (2025). *MiniMax-M2 Model Card*. https://huggingface.co/MiniMaxAI/MiniMax-M2↑

13. Peng, B., Yao, W., Wu, Q., Cheng, H., Yu, X., Yang, R., Ge, T., Sordoni, A., Yuan, X., Shen, Y., He, P., Zhang, T., Yu, Z., & Gao, J. (2026). Orchard: An Open-Source Agentic Modeling Framework. *arXiv Preprint arXiv:2605.15040*. https://arxiv.org/abs/2605.15040↑

14. poolside. (2026). Laguna M.1/XS.2 Technical Report. *arXiv Preprint arXiv:2605.27605*. https://arxiv.org/abs/2605.27605↑

15. Qwen Team. (2026). Qwen3-Coder-Next Technical Report. *arXiv Preprint arXiv:2603.00729*. https://arxiv.org/abs/2603.00729↑

16. von Werra, L., Belkada, Y., Tunstall, L., Beeching, E., Thrush, T., Lambert, N., Huang, S., Rasul, K., & Gallouédec, Q. (2020). *TRL: Transformers Reinforcement Learning*. https://github.com/huggingface/trl↑

17. Yu, X., Peng, B., Xu, R., Zou, H., Wu, Q., Cheng, H., Yao, W., Singh, N., Yu, Z., & Gao, J. (2026). OpenForgeRL: Train Harness-native Agents in Any Environment. *arXiv Preprint arXiv:2607.21557*. https://arxiv.org/abs/2607.21557↑

18. Z.ai. (2025). *GLM-4.7*. https://huggingface.co/zai-org/GLM-4.7↑

19. Zhang, Y., Wang, J., Ge, Y., Xu, W., Hamm, J., & Reddy, C. K. (2026). Stop Comparing LLM Agents Without Disclosing the Harness. *arXiv Preprint arXiv:2605.23950*. https://arxiv.org/abs/2605.23950↑ back: 1, 2

1. Kolavi, A. S., Tunstall, L., von Werra, L., Gallouédec, Q., Dirhoussi, A., Burtenshaw, B., & Paniego, S. (2026). *The ultimate guide to RL environments: building and scaling them in the LLM era*. https://huggingface.co/spaces/AdithyaSK/rl-environments-guide↑

2. KwaiKAT Team. (2026). KAT-Coder-V2.5 Technical Report. *arXiv Preprint arXiv:2607.05471*. https://arxiv.org/abs/2607.05471↑

3. Zhang, Y., Wang, J., Ge, Y., Xu, W., Hamm, J., & Reddy, C. K. (2026). Stop Comparing LLM Agents Without Disclosing the Harness. *arXiv Preprint arXiv:2605.23950*. https://arxiv.org/abs/2605.23950↑

1. Gallouédec, Q., & Rasul, K. (2026). *Agentic RL: Token-In, Token-Out Done Right*. https://huggingface.co/blog/huggingface/tito↑ back: 1, 2

2. He, Z., Zhang, S., Zhou, Z., Yang, Y., Kang, Y., Zhang, Y., Qiu, L. K., Tsui, T. Y., Xu, J., & Luo, C. (2026). Agent Lightning v1.0: Towards Harnessed Agentic RL. *arXiv Preprint arXiv:2608.17528*. https://arxiv.org/abs/2608.17528↑

3. Kolavi, A. S., Tunstall, L., von Werra, L., Gallouédec, Q., Dirhoussi, A., Burtenshaw, B., & Paniego, S. (2026). *The ultimate guide to RL environments: building and scaling them in the LLM era*. https://huggingface.co/spaces/AdithyaSK/rl-environments-guide↑

4. KwaiKAT Team. (2026). KAT-Coder-V2.5 Technical Report. *arXiv Preprint arXiv:2607.05471*. https://arxiv.org/abs/2607.05471↑

5. Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3), 229–256. 10.1007/BF00992696↑

6. Yu, X., Peng, B., Xu, R., Zou, H., Wu, Q., Cheng, H., Yao, W., Singh, N., Yu, Z., & Gao, J. (2026). OpenForgeRL: Train Harness-native Agents in Any Environment. *arXiv Preprint arXiv:2607.21557*. https://arxiv.org/abs/2607.21557↑

1. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *The Twelfth International Conference on Learning Representations*. https://arxiv.org/abs/2310.06770↑

2. Kolavi, A. S. (2026). *Harbor integration: serve Harbor task datasets through OpenEnv as trainable environments*. https://github.com/huggingface/OpenEnv/pull/1036↑

3. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of the 29th Symposium on Operating Systems Principles*. https://arxiv.org/abs/2309.06180↑

4. Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., & others. (2026). Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv Preprint arXiv:2601.11868*. https://arxiv.org/abs/2601.11868↑

5. Meta PyTorch, & Hugging Face. (2025). *OpenEnv: Agentic Execution Environments*. https://github.com/huggingface/OpenEnv↑

6. Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., & Guo, D. (2024). DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv Preprint arXiv:2402.03300*. https://arxiv.org/abs/2402.03300↑

7. Xu, B., Zhang, H., Zhang, S., Han, S., Liu, M., Hu, J., Diao, S., Jin, Z., Zou, Y., Demoret, M., Kautz, J., & Dong, Y. (2026). Polar: Agentic RL on Any Harness at Scale. *arXiv Preprint arXiv:2605.24220*. https://arxiv.org/abs/2605.24220↑

1. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of the 29th Symposium on Operating Systems Principles*. https://arxiv.org/abs/2309.06180↑

2. Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., & Guo, D. (2024). DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv Preprint arXiv:2402.03300*. https://arxiv.org/abs/2402.03300↑

Made with ❤️ with research article template