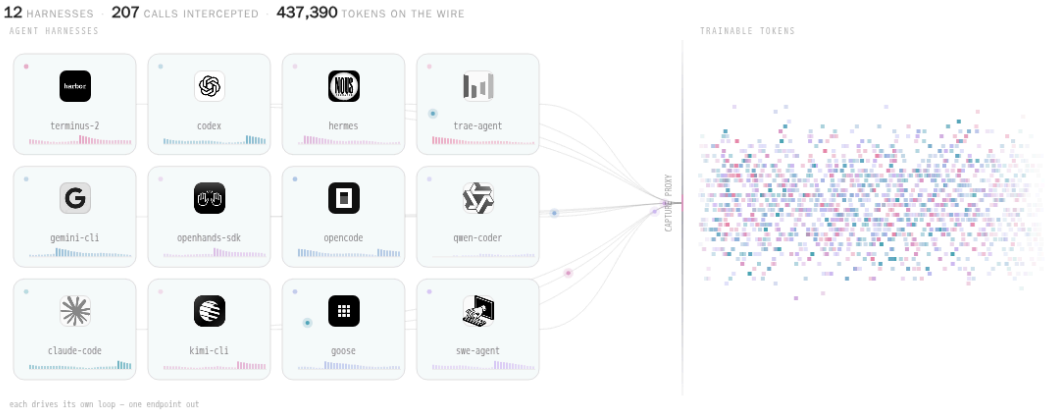
The ultimate guide to multi-harness RL



Harnesses | Rollouts

Training and evals together: how a capture layer turns any black-box agent harness into trainable tokens, and what it takes to train a small model across many of them.

AUTHOR

Adithya S Kolavi

AFFILIATION

Hugging Face

PUBLISHED

August 30, 2026

Table of Contents

Introduction

If you use AI to write code, you have probably run the same model through more than one tool. Claude Code in the terminal, Codex or Cursor in the editor, maybe Cline. And you may have noticed something that is easy to write off as a bad day: the model that felt sharp in one tool feels clumsy in the next. Same model, same task, different tool wrapped around it, and it behaves like a different assistant.

That is not your imagination. It is measurable, and it has become one of the central problems in training open models.

The tool wrapped around the model is called an agent harness. It runs the loop, decides which tools the model can use, writes the context the model reads, interprets what comes back, and decides when to stop. To the model, the harness is the entire world. And the open model you downloaded was almost certainly trained inside exactly one of them.

The labs already tell you this, in the fine print

Read the release notes of the newest open models and the same admission keeps surfacing in different handwriting.

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 twice, 25.0 under Terminus and 30.0 inside Moonshot's own framework, as two rows of one table (Kimi Team, 2025). MiniMax M2 names a different harness for almost every benchmark it lists (MiniMax, 2025). DeepSeek-V3.2 checks its scores across Claude Code and RooCode, then admits its thinking mode is incompatible with Terminus, so that number had to come from somewhere else (DeepSeek-AI, 2025).

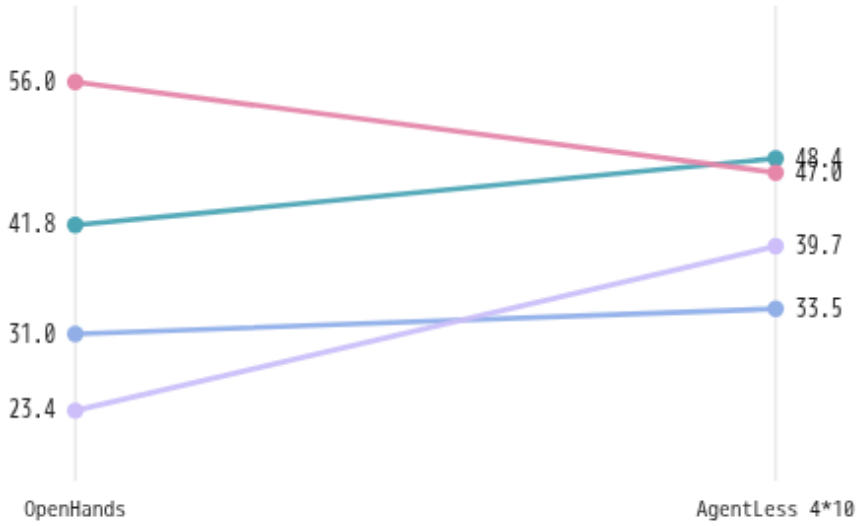
Sit with that last one. A major lab shipped a model that could not run in a major harness, noticed, and reported the score from a different one.

Which harness a model is good in has become a shipping claim, printed beside the benchmark numbers, because nobody runs raw weights. They run Claude Code with your model behind it.

It gets worse than uneven scores. Run the same benchmark under two harnesses and the ranking between models can flip. ByteDance's Seed-OSS card has two SWE-Bench Verified rows covering the same models, and the order reverses twice between them (ByteDance Seed, 2025). Some of that gap is extra sampling rather than the harness, and the reversals survive

that objection anyway: more sampling can raise a score, it cannot explain why one model passes another while a second model moves the opposite way.

One benchmark, two scaffolds, two rank reversals



Lines that cross are rank reversals: the two models swap order when only the scaffold changes. AgentLess 4*10 is best-of-n sampling, so absolute gaps mix scaffold with test-time compute. The rank reversals survive that; the raw point gaps do not. Source: Seed-OSS-36B-Instruct model card, ByteDance Seed.

Figure 1 · SWE-Bench Verified for four models, run under OpenHands and under Agentless.

So “model A is better than model B” is not a harness-independent statement. Anthropic said as much back in 2024 (Anthropic, 2024). What has changed since is how much now rides on it.

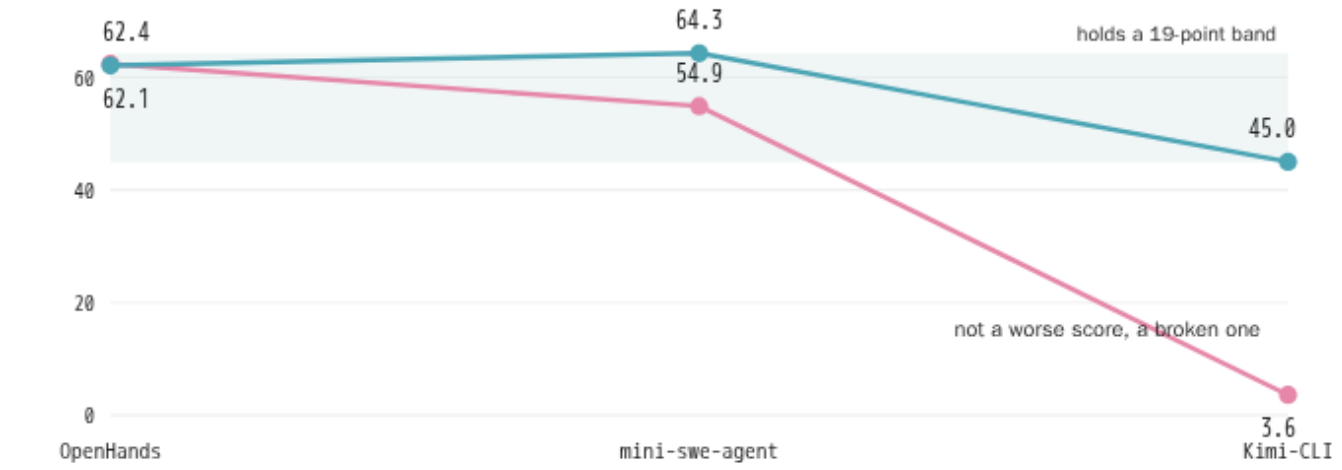
How badly it breaks

Not gently. The Orchard team trained a coding agent in one harness, then ran it in others (Peng et al., 2026):

System	OpenHands	mini-swe-agent	Kimi-CLI (unseen)
OpenSWE-32B, single-harness	62.4	54.9	3.6
Orchard-SWE, multi-harness	62.1	64.3	45.0

62.4 percent of SWE-bench Verified in the harness it trained in. 3.6 percent in one it had never seen. That is not a worse score, it is a broken one. A second system in the same table does not even produce valid tool calls outside its own harness, so it has no score at all.

What single-harness training costs



LEGEND

OpenSWE-32B, trained in one harness Orchard-SWE, trained across several

Same benchmark, same three harnesses, two training recipes. The model trained in one harness resolves 62.4 percent inside it and 3.6 percent in a harness it has never seen. Percentage points on the axis, so the drop is a cliff rather than a slope.

Figure 2 · Two systems, three harnesses, one of them collapsing.

If you have ever watched a local model call a tool its harness has never heard of, or spill raw markup into a field that wanted JSON, you have seen the small version of this. The model is not confused about the task. It is confused about the interface, and the interface is the only thing it can see.

The KwaiKAT team names the failure, harness scaling, and describes it better than anyone (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.'"*

It shows up three ways, worth keeping for later. Format overfitting anchors the model to one action format, so parsing breaks when the protocol changes. Context-structure overfitting ties it to one harness's way of stacking up history. Control-flow overfitting leaves it leaning on someone else's retry timing and stop conditions, unable to plan for itself once they are gone.

Three ways a model overfits to its harness

	ANCHORED TO	BREAKS WHEN
Format overfitting	the action format it was trained to emit	the tool protocol changes, and parse failures spike
Context structure overfitting	the order its training harness concatenated history in	the context is laid out differently, and behaviour degrades
Control flow overfitting	someone else's reflection timing and stopping conditions	that scaffolding is absent, and it cannot plan for itself

Named by the KAT-Coder team, and worth keeping to hand: the rest of this article is about removing all three at once by varying the harness during rollouts, rather than patching each of them separately.

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

Frontier labs stopped reporting this and started training for it

The fix is to vary the harness during training, which KwaiKAT frames as domain randomization applied to the environment. Labs now pay real money for it.

poolside's Laguna report has a section titled Multi-harness Training, and its opening line is this article's thesis in someone else's words (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 trajectories from three harnesses that are not theirs, each one's native behaviour deliberately left intact. Their RL phase then runs through the harness they ship to customers, so "any change to the deployed harness is also a change the policy is trained against."

The obvious worry is that spreading training across harnesses trades peak performance for robustness. The one controlled ablation published so far says otherwise. OpenForgeRL trained the same model on one harness and on three, and the three-harness run won even on the single harness's home turf, 48.5 against 46.0, while nearly tripling Codex (Yu et al., 2026). Harness variety behaved like a regularizer, not a tax.

None of this has to be taken on my word. Here are the passages themselves.

The claims, in the papers themselves

poolside | KwaiKAT | Orchard | OpenForgeRL | Agent Lightning

POOLSIDE 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.

Crops rendered from the official arXiv PDFs. Nothing is retouched or recomposed; the only edit is the crop rectangle. Follow any link to read the passage in context.

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

What is missing

Not the idea, and not the tooling. Agent Lightning takes Qwen3.5-9B from 41.8 to 56.4 percent on SWE-bench Verified using roughly 6,000 training examples and modest compute (He et al., 2026). That is not a frontier-scale result and is not meant to be. It shows the work now fits inside what a small team, or one person with a rented GPU, can actually run.

What is missing is an open one. Every published result that trains across many harnesses comes from a closed or semi-closed lab. Nobody has swept how many harnesses you need, or along which axes, before the returns flatten out. The pieces all exist, in OpenEnv, in verifiers, in SkyRL, in Harbor. The run does not.

That is what the rest of this article is: what a harness is, what an RL environment is, how to capture trainable tokens out of an agent you do not control, and then the thing itself, a small open model trained across many harnesses, evaluated in all of them, with the parts that did not work left in.

What is a harness?

The introduction leaned on the word without defining it, which is fair for a first pass and not good enough for the rest of the article. So: an agent harness is the program that runs the agent. It owns the loop, decides which tools exist, builds the context the model sees, parses what comes back, and decides when to stop. From the model's side the harness is the entire world. From the trainer's side it is a binary you do not control.

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). LangChain's version is shorter and sticks in the head better: agent equals model plus harness, and if you are not the model, you are the harness.

It helps to say what a harness is not, because four neighbouring things get called by the same name in casual writing.

	Owns
Model	The next-token distribution. Stateless, no loop, no memory between calls.
Harness	The loop, the tool surface, the prompt, context management, retries, the stop cond
RL environment	State, transition and reward. It receives an action and returns an observation.
Sandbox	The isolated place actions actually run: a container, a VM, a tmux session.
Benchmark	A fixed task set plus an evaluation protocol, administered on top of environments.

The distinction that matters most here is the second against the third, and it is a distinction of direction rather than content. The harness executes the loop. The environment receives actions and returns observations. A harness pulls, an environment is pulled.

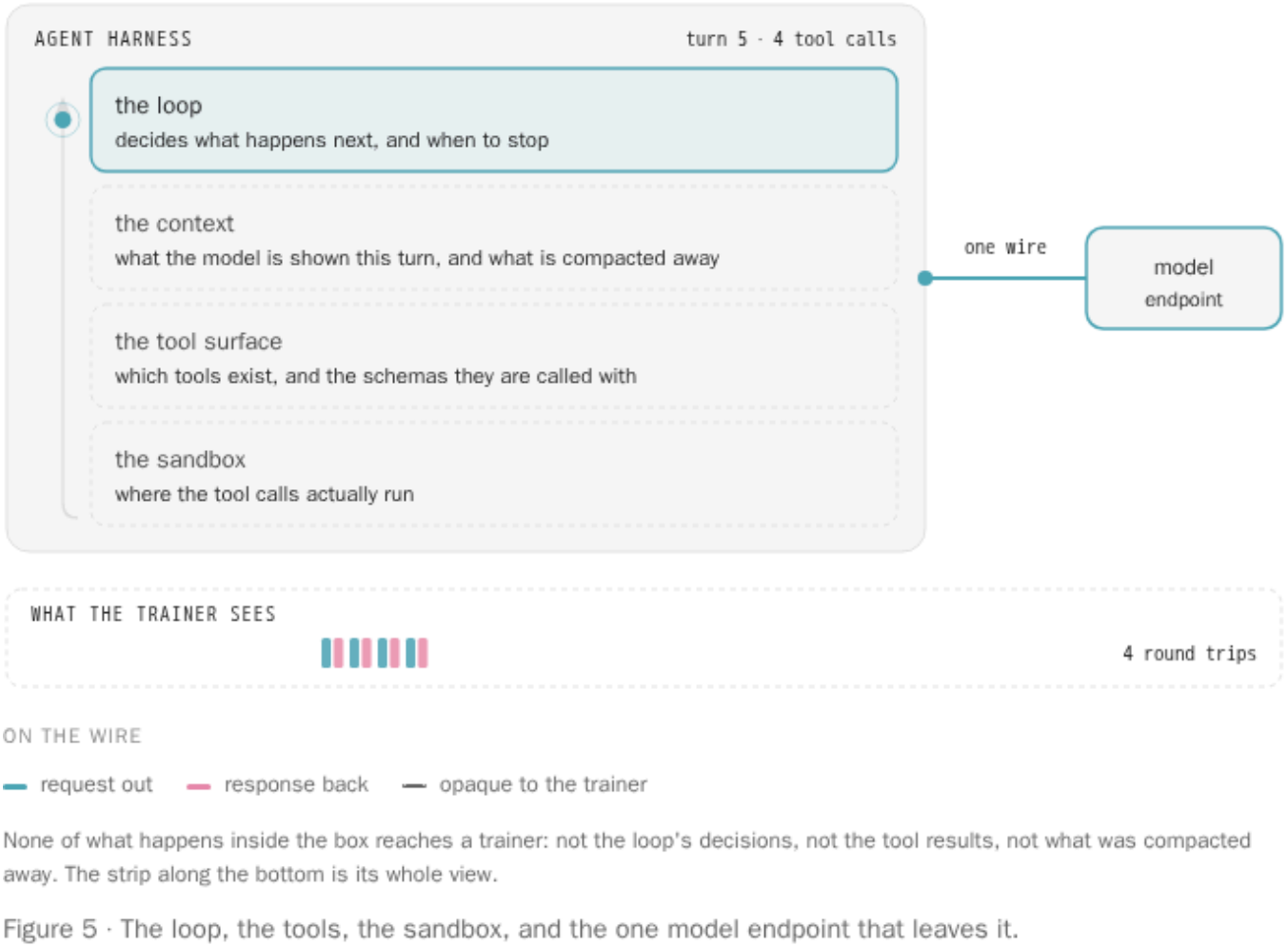
> *Note: the previous guide uses "harness" in a narrower sense, meaning the trainer-side interaction layer, the tool surface a trainer drives. This article uses it in the sense the 2026 literature settled on, meaning the whole deployed agent program that owns the loop. Same word, nearly opposite locus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

What a harness owns, and the one thing it does not

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 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*. Every system in the next chapter works by taking the endpoint away from the harness and putting a proxy in its place, which is possible precisely because the endpoint was never the harness's to own.

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.

The harness ecosystem today

Why does variety here cause so much trouble?

Because the variety is not cosmetic. Harnesses differ in the API dialect they speak, 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.

The harness landscape

HARNESS	MAKER	WIRE DIALECT	TOOL CALLING	WHAT MAKES IT DI
`claude-code`	Anthropic	Anthropic Messages	yes	The de-facto referenc
`codex`	OpenAI	OpenAI Responses	yes	The only harness to
`opencode`	Anomaly	all four	yes	Headless daemon wi
`goose`	Agentic AI Foundation	Anthropic, chat, Google	yes	Most MCP-native; ca
`swe-agent`	Princeton, Stanford	chat-completions	yes	Tools installed into t
`mini-swe-agent`	SWE-agent team	chat, Responses, text	one tool	Deliberately minimal,
`openhands-sdk`	OpenHands	chat-completions	yes	Event-stream archite
`aider`	Aider-AI	chat-completions	no	Parses edit formats
`qwen-coder`	Alibaba	multi, Google-shaped	yes	Forked from Gemini
`kimi-cli`	Moonshot	chat-completions	yes	A coding agent that i
`terminus-2`	Laude Institute	chat-completions	no	Uses no tool-calling
`hermes`	Nous Research	all three, switchable	yes	Persistent multi-surf

Dialect is the axis that matters most for capture: it decides whether one proxy can serve a harness at all.

Figure 6 · Agent harnesses, the API dialect each speaks, and what it targets.

Two entries in that table are worth pulling out. The first is that mini-swe-agent's own README gives scaffold overfitting as a reason to use it, which tells you the problem was understood by harness authors before it was named in a tech report. The second is aider, which does not do tool calling at all and instead asks the model to emit edit blocks in prose. A model trained to produce structured tool calls and then dropped into aider is not slightly out of distribution, it is being asked to speak a different language.

That is also the practical shape of harness overfitting in the wild. The failure does not usually look like a lower score. It looks like an open model emitting capitalised tool names that its new harness has never heard of, or leaking raw markup into a field that was supposed to hold JSON, and the harness returning an error rather than a result. The model is not confused about the task. It is confused about the interface, and the interface is the only thing it can see.

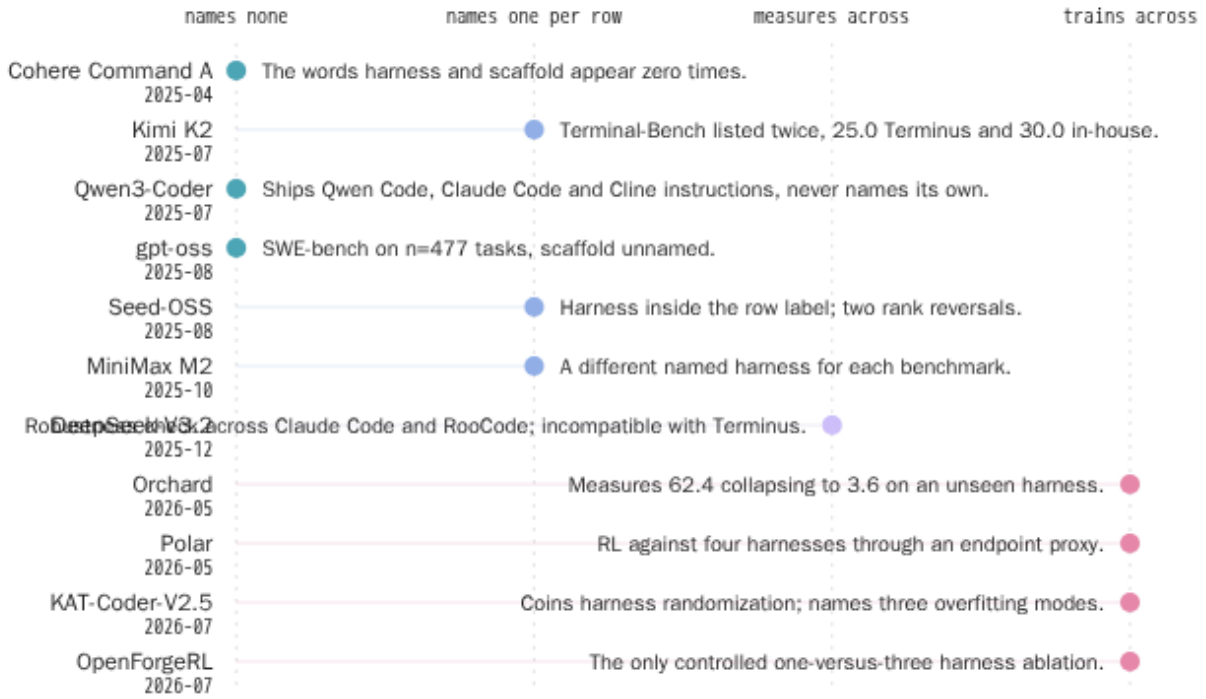
What harness overfitting actually looks like

This is what harness overfitting looks like from the outside. Not a lower score: a tool name memorised in one harness's casing, and a foreign harness that has never heard of it. Reported repeatedly against open models dropped into agents they were never trained in.

Figure 7 · One model output, accepted by its native harness and rejected by a foreign one.

The fact that this table needs to exist at all is recent. Eighteen months ago a model report could omit the word harness entirely and lose nothing; today the harness is named per benchmark row, and in some cases trained against.

When the harness entered the tech report



Disclosure level rises from naming no harness at all, to naming one per benchmark row, to measuring across several, to training across several. The lower half of this chart is eighteen months old.

Figure 8 · When the harness started appearing in tech reports.

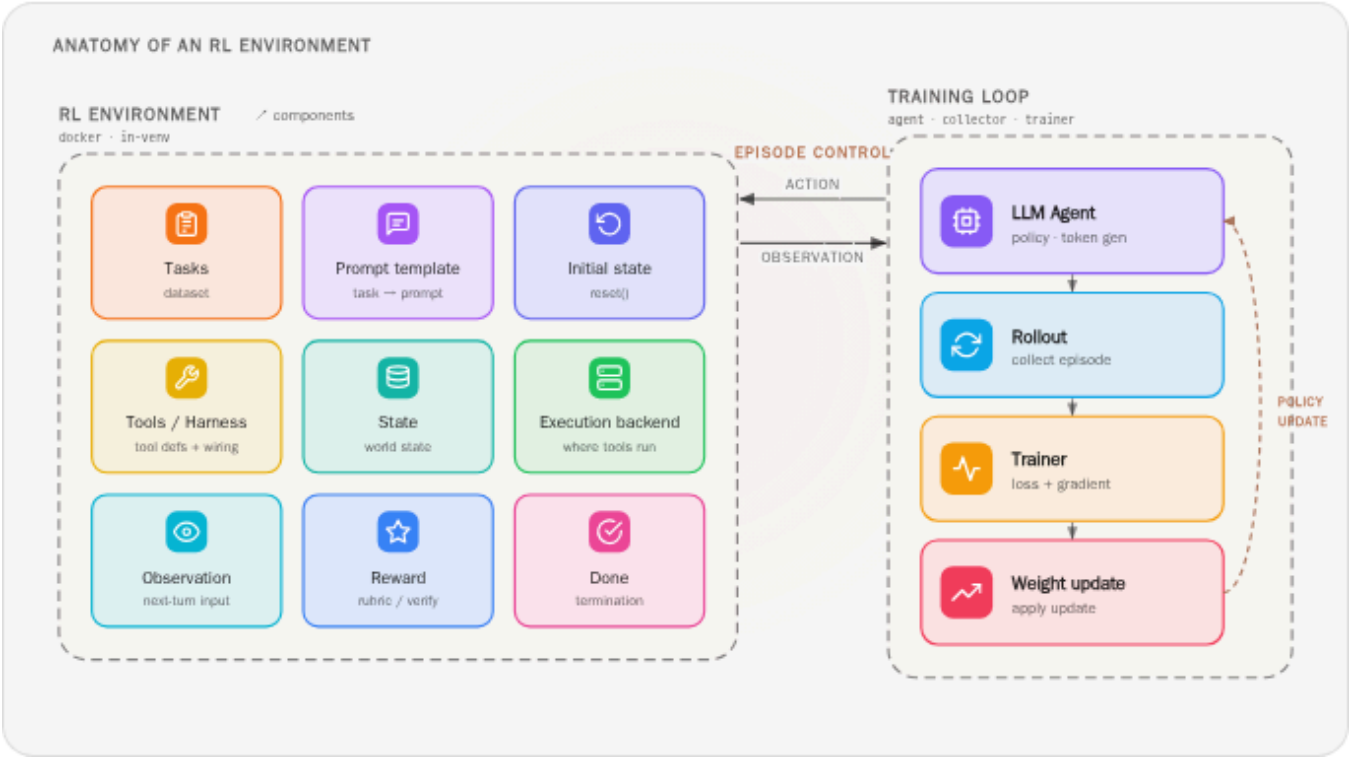
The count keeps moving, which is itself part of the argument. Harbor ships adapters for roughly forty harnesses. The OpenEnv integration this article builds on validates sixteen end to end. Any number printed here will be wrong within a quarter, and a model trained against one of them will still be running inside the others.

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.

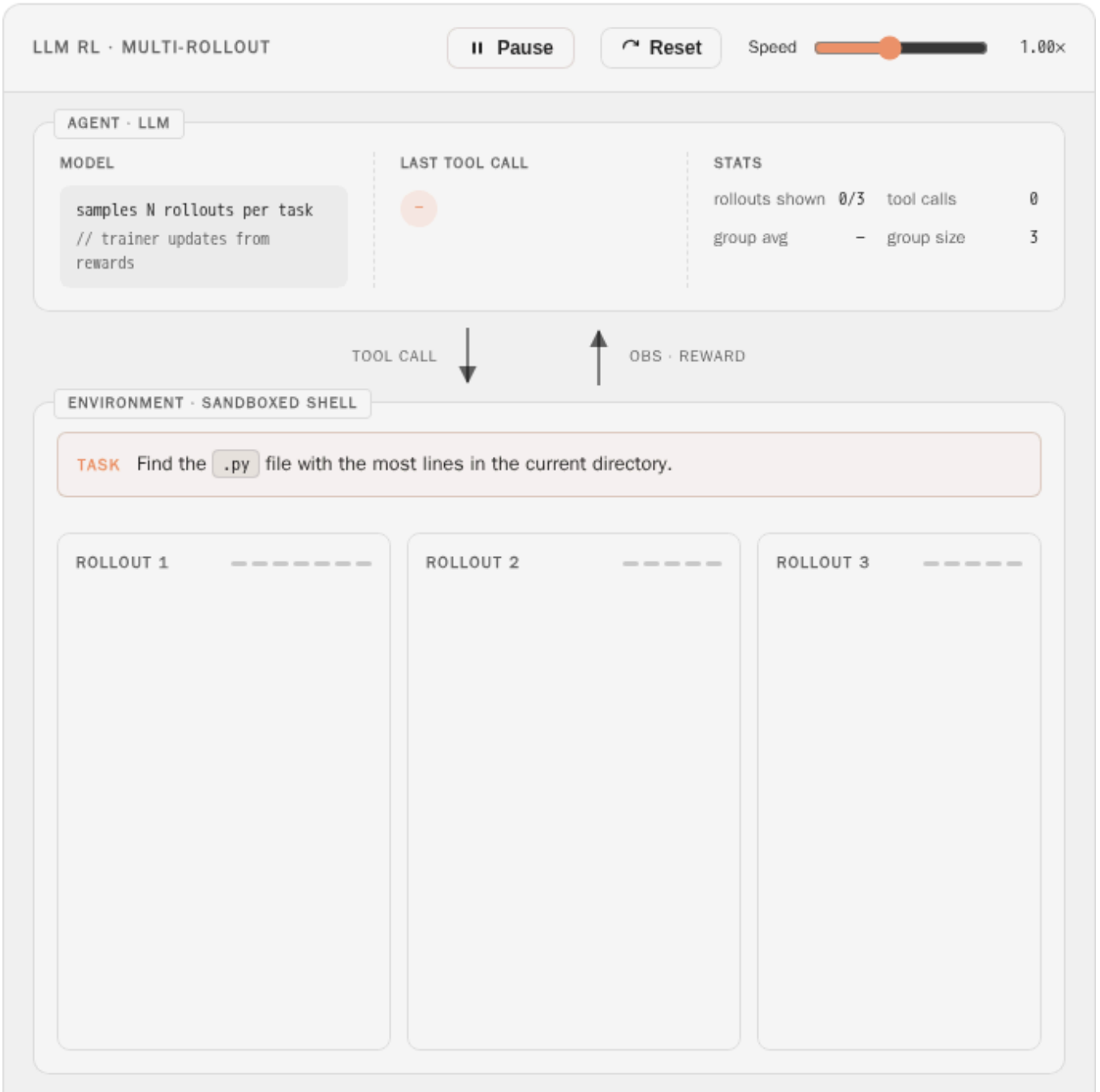
Drawn out, the two halves look like this. On one side a training loop that samples, scores and updates. On the other an environment made of far more parts than the word suggests.



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 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.

Why OpenEnv?

OpenEnv is a standard interface for RL environments: 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 around eleven organisations, under BSD-3-Clause.

The reason it exists is worth quoting, because it could have been written as the abstract for this article:

> *"Agent harnesses like Claude Code, Codex, OpenClaw, and Hermes just keep improving. One reason for their improvement is that models like GPT-5.5 and Opus 4.8 are trained to use their respective harnesses. We want those gains with open source models too."*

Frontier labs train the model and the harness together, so the two fit. In the open, nobody controls both ends: developers pick any harness, any model, any inference engine. OpenEnv's role in this article is narrow and specific. It is not the trainer and it does not define rewards. It is the socket that the harness, the environment and the trainer all plug into, which is what makes it possible to serve the same task set into a dozen different agent programs without reimplementing anything.

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):

> *"Harnessed agentic RL differs fundamentally from traditional agentic RL: the harness, rather than the training engine, owns the environment interaction loop, while the trainer observes only sequences 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 uses white-box and black-box to mean something else, namely whether a harness's internals are inspectable, with mini-swe-agent as white-box and Claude Code as black-box. 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.

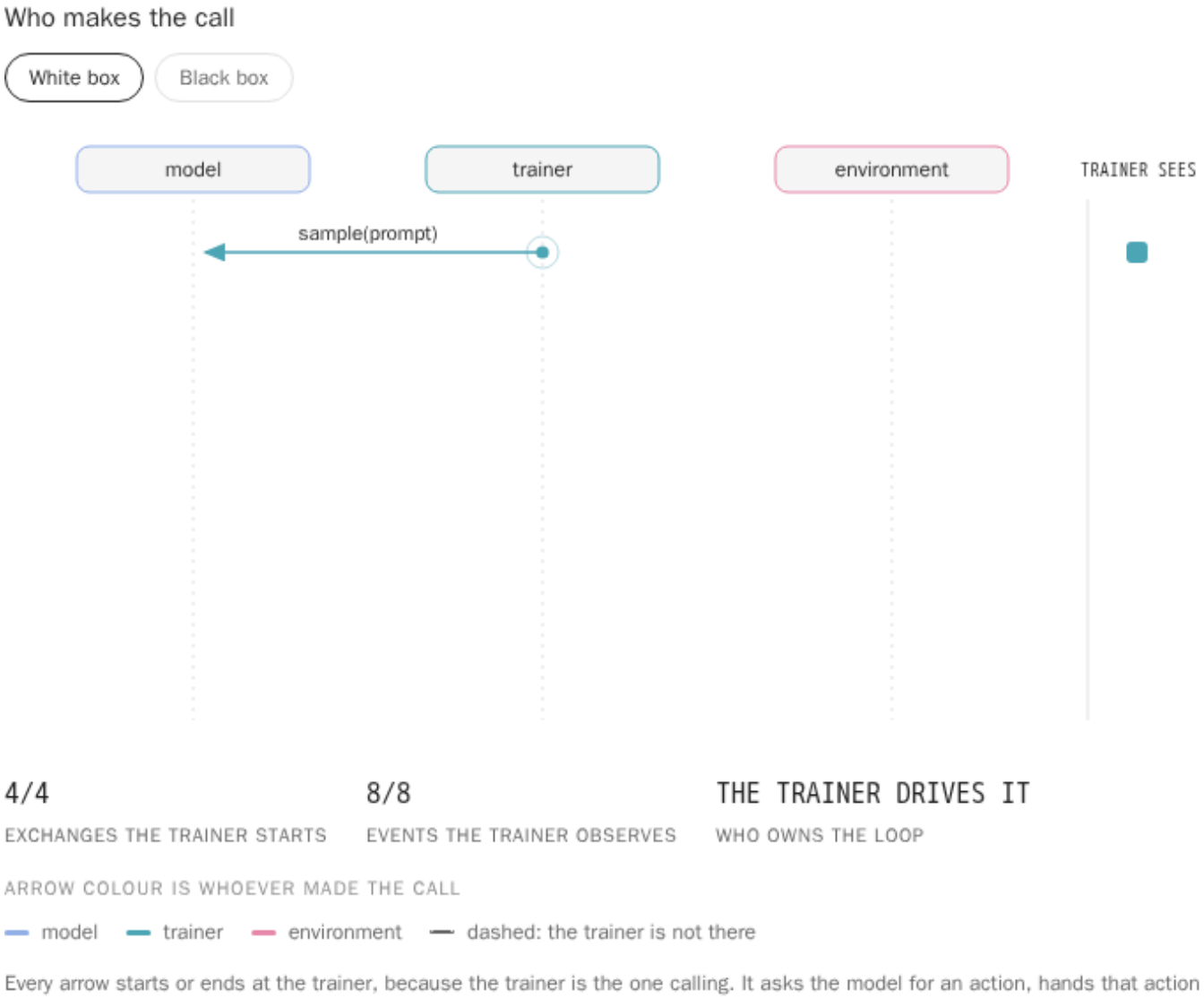


Figure 11 · Who initiates each call, under both architectures.

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.

The capture layer

If the agent owns the loop, the only place a trainer can stand is the wire between the agent and the model. Put an OpenAI-spec proxy there and every call the agent makes passes through it. The agent is pointed at the proxy by a per-harness seam, usually a single environment variable, and its API key is really a capture session id, which is how one proxy serves many concurrent rollouts without needing a port each.

Two properties make this general rather than a per-agent hack. Nothing is tokenised locally: the inference engine tokenises each prompt in order to serve it and hands back the token ids, so turn *k+1*'s prompt is by construction the canonical tokenisation of everything before it, tool results included. And four wire dialects are supported, because coding agents never converged on one. That is not completeness for its own sake: a capture layer that speaks only chat-completions cannot see codex, claude-code or gemini-cli at all, so the other three dialects are exactly what buys those agents.

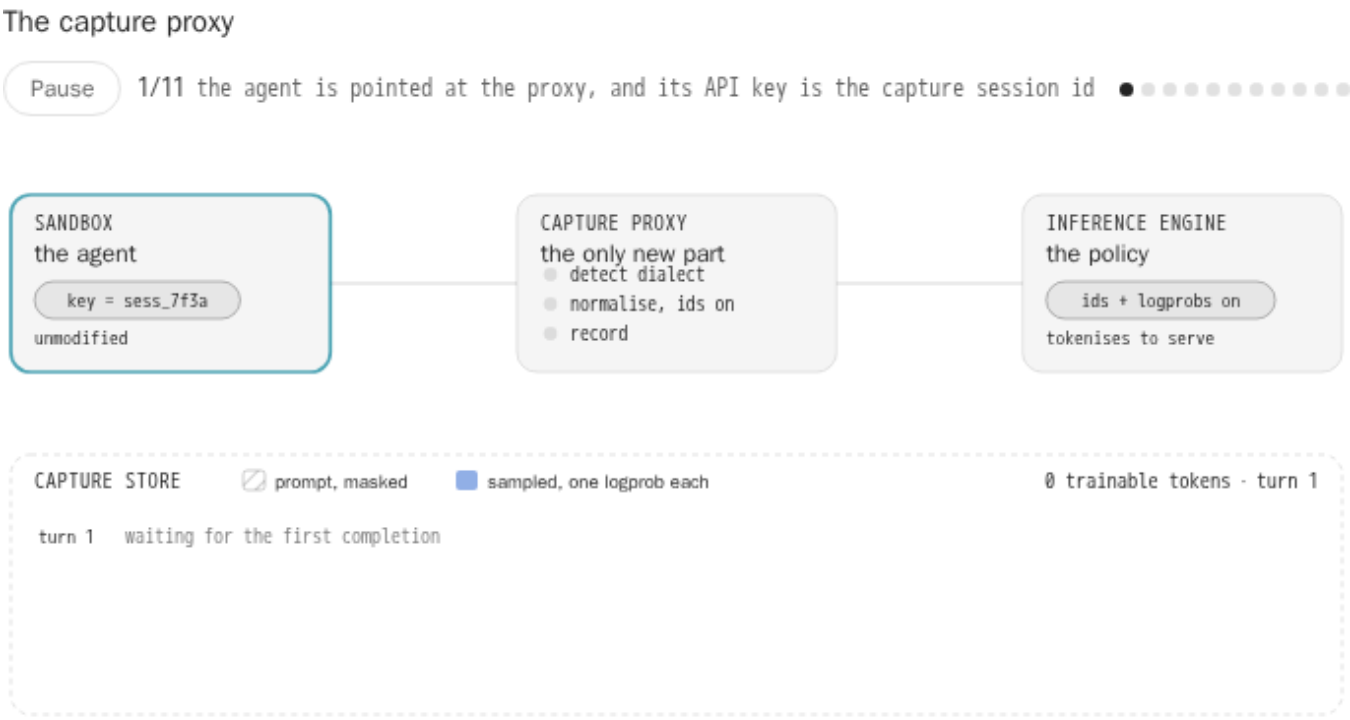


Figure 13 · One rollout, step by step, through the capture layer.

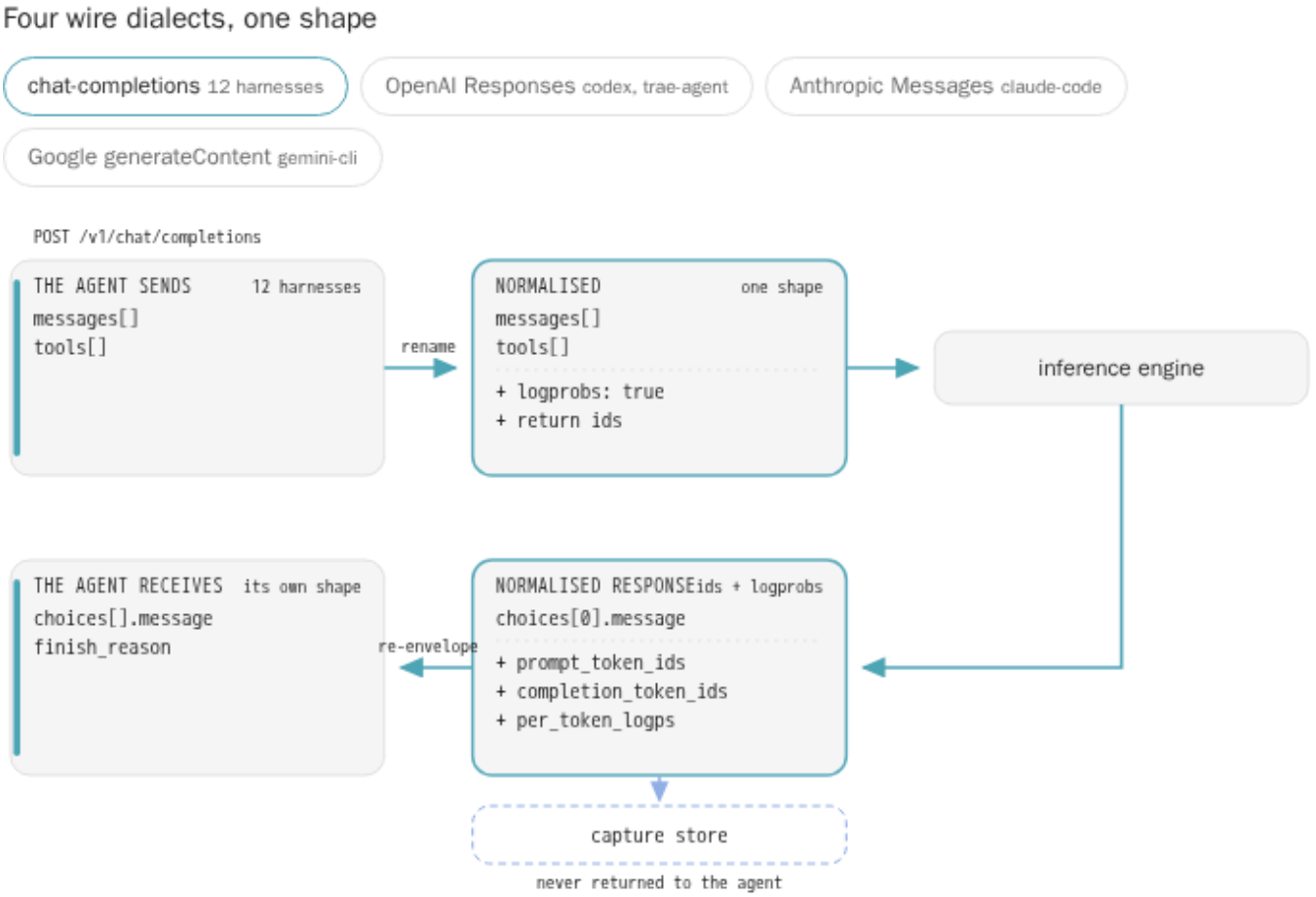


Figure 14 · The same request in four dialects, and one shape in the middle.

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 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.

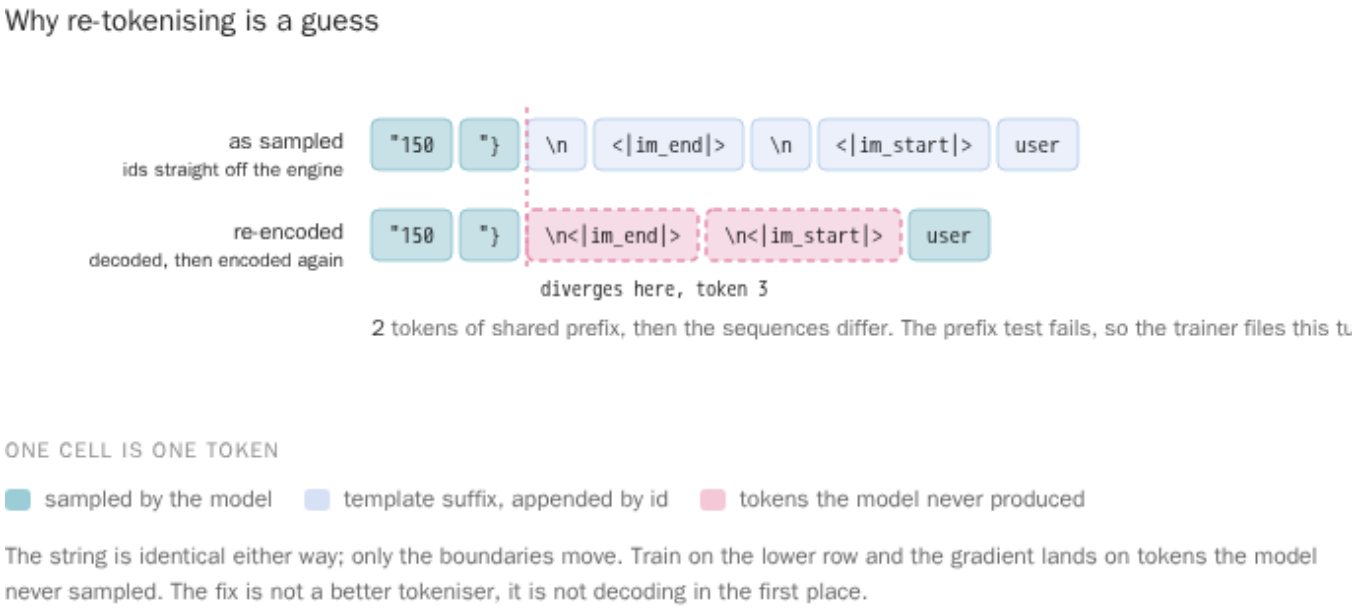
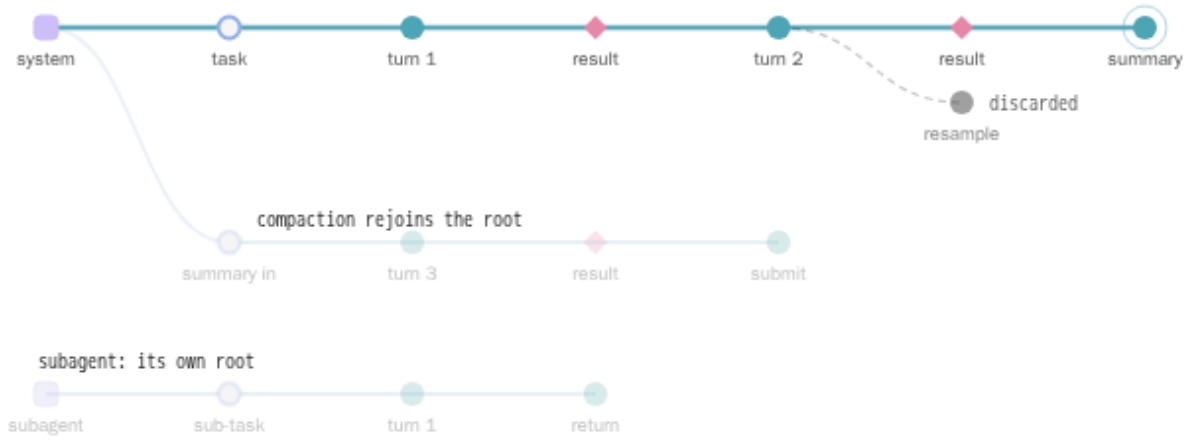


Figure 15 · The same characters, tokenised two different ways.

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.

The rollout graph

Branch 1 | Branch 2 | Branch 3 | All



16	3	36	7
UNIQUE MESSAGES	BRANCHES, SO 3 SAMPLES	COPIES IF STORED PER TURN	MESSAGES IN THIS BRANCH

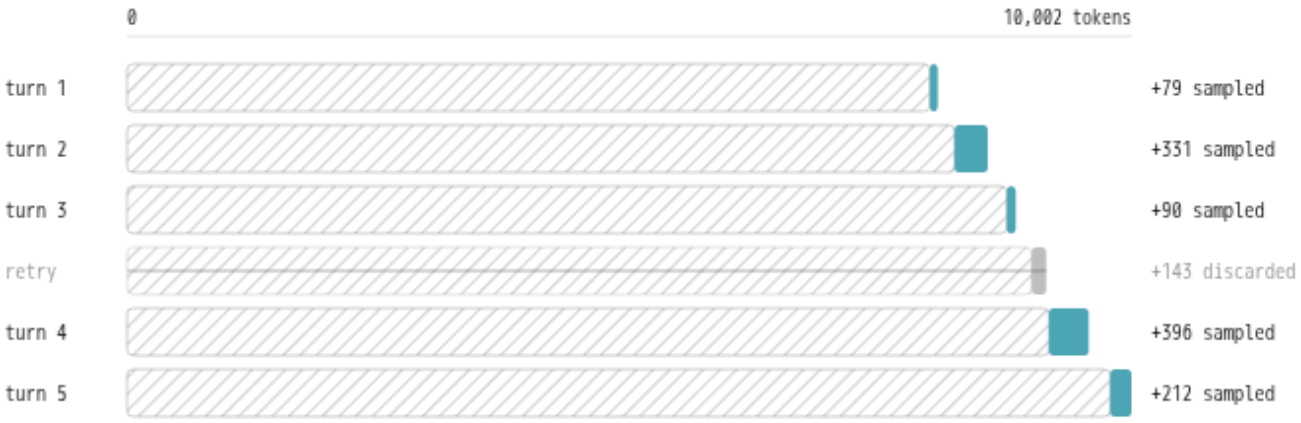
ONE NODE IS ONE MESSAGE

system | user | assistant, the sampled part | tool result

Turns link by exact token prefix, not by request id: ids are per-harness, the prefix is not. Storing each message once instead of re-storing the whole prompt keeps a long rollout linear rather than quadratic.

Figure 16 · A trace as a graph of messages, and its trainable branches.

What a rollout returns



1,108 tokens sampled across the rollout, out of **10,002** in the finished sequence. 11% of it is trainable.

ONE BAR IS ONE TURN, DRAWN ON THE TOKEN AXIS

prompt, masked out of the loss | sampled, carries a logprob | abandoned branch, excluded

Each bar starts at zero because each prompt contains every turn before it. The step between one bar's tip and the next bar's start is the tool result appended in between.

Figure 17 · What one rollout hands back, turn by turn.

DOWNSIDES AND LIMITATIONS

The capture layer is not free, and the failure modes are specific enough to be worth listing.

Capture has to happen on the wire. Re-rendering a prompt offline with `apply_chat_template` drifts from what the model actually saw, and a prompt that is off by a single token silently fragments one long conversation into several short ones. There is no fixing this after the fact.

The endpoint has to cooperate. vLLM needs `--return-tokens-as-token-ids` and `--logprobs-mode processed_logprobs`. Without them any OpenAI-spec endpoint still works, you just get evaluations rather than trainable rollouts. SGLang cannot return token ids at all today, which rules it out entirely.

A hosted proxy has to be publicly reachable. A private Space needs an auth header that the agent inside the sandbox does not send, so the capture endpoint has to be public. That is safe here only because the proxy rejects any caller without a registered session id.

`reward=None` is not zero. It means the verifier never ran. Collapsing the two makes a dead sandbox look like a wrong answer, which is the kind of bug that quietly poisons a training run rather than crashing it.

Validation is only as good as its independence. Reconciling captured calls against the harness's own trace format is the one check that is not self-referential, and it can come back as a mismatch. A proxy can be internally consistent and still wrong.

The word sandbox is overloaded. An OpenEnv provider hosts the environment server. A Harbor backend is the sandbox the agent executes in. They are different things, and `--sandbox` refers to the second.

Why a middle layer

The line count is the argument. Wiring each harness to each sandbox means maintaining the product; putting one capture layer between them means maintaining the sum, and adding the next harness becomes a table entry rather than a package.

Figure 18 · Every harness and sandbox pair, or one capture layer they all share.

There is one honest caveat to close on. Not everyone agrees the token-level machinery is necessary. OpenForgeRL builds the same proxy architecture and then reconstructs training samples from plain text prompt and response pairs, with no token ids and no log probabilities, and still reports the strongest multi-harness results published so far (Yu et al., 2026). Same year, same architecture, opposite answer on what the trainer needs. This article takes the token-faithful side, and the question is not settled.

Data Agent

Everything so far has been other people's evidence. This is where the article stops citing and starts running things.

The goal is narrow and deliberately unglamorous: take a small open model, something under five billion parameters that a person with one rented GPU could actually train, and make it competent at real data-analysis work. Then use that as an end-to-end recipe other people can rerun.

Why not Wordle

If you go looking for a worked example of RL on a language model today, you will almost certainly find Wordle. It is the house plant of RL environments: easy to keep alive, present in

every tutorial, and not doing very much. It is a fine way to check that a training loop is wired up correctly. It is a poor way to learn whether any of this works, because nobody wants a model that is good at Wordle, and the task shares almost nothing with the work people actually hand to an agent.

The gap that leaves is the reason this project exists. There is no shortage of RL environment frameworks and no shortage of toy tasks. What is missing is a task that is realistic enough to be worth winning, cheap enough to iterate on, and verifiable enough that the reward can be trusted, all published together with the code that produced it.

Data analysis fits, and it is worth saying why, because the domain does most of the work in making an RL experiment either honest or useless.

It is multi-turn but bounded. The model loads a file, looks at the shape of it, tries something, reads the result, corrects itself. That is genuinely agentic behaviour, several tool calls deep, and it still terminates in a minute or two rather than an hour. Long-horizon software engineering is more impressive and far more expensive to iterate on.

It is verifiable without a judge. Most questions have exactly one right answer, and it is a number or a short string, so a grader can check it without asking a language model for an opinion. That matters more than it sounds, and the next section is largely about how much work went into keeping it true.

It is not saturated. Small models are genuinely bad at this to begin with, which leaves room for training to show up as something other than noise.

And the raw material already exists. The jupyter-agent dataset pairs real Kaggle notebooks with the questions they answer, roughly forty thousand of them, each with an answer that a human actually computed. That is a large supply of verifiable tasks sitting in a form nobody had turned into environments yet.

Turning notebooks into environments

Everything below is published as the Data Agent collection.

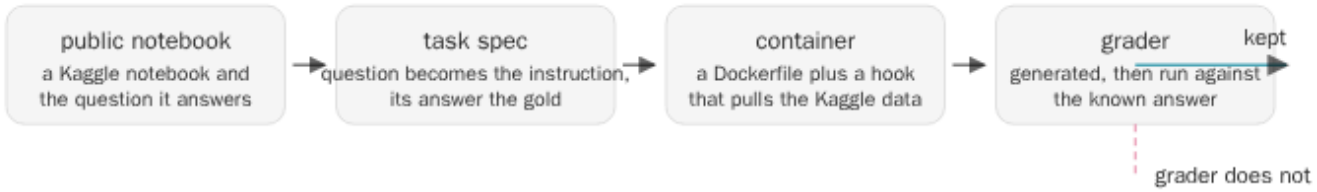
Each candidate question goes through the same pipeline. The question becomes the instruction and the notebook's own answer becomes the gold. A build-time model canonicalises that gold into a single clean value and picks how it should be compared, so an answer recorded in a notebook as "Y=3 with 95,293 instances" becomes the string `95293` with a numeric reward mode. The Kaggle dataset the notebook depended on becomes a container that pulls the data

into `/home/user/input/` before the agent starts. Then the task is generated as a Harbor task directory.

The step that decides whether the task ships is the last one. An anchor agent runs the finished task for real, in a sandbox, with a single bash tool, and writes its answer to `/workdir/answer.txt`. The deterministic grader scores that answer against the gold. If the anchor reproduces the gold, the task is admitted. If it does not, the task is dropped. DeepSeek-V4-Flash runs first and Claude Sonnet 4.6 picks up its failures.

That gate is expensive and it is the reason to trust anything downstream, so the attrition is worth stating rather than hiding. Fresh candidates are admitted at roughly 46 percent under the DeepSeek anchor and 53 percent under Sonnet. A separate deterministic re-audit of the previous 2,238-task generation dropped 604 of them, 27 percent, because those tasks had only ever passed under an LLM judge. A final recomposition pass dropped another 83 as structurally un-gradable: non-comma separators, parenthetical asides, free text. Most raw candidates never become tasks.

From a public notebook to a verified task



The drop step is where the trust comes from. A question is only useful for training if its grader demonstrably fires on the answer the notebook already gave, so anything that fails that check never reaches the suite.

Figure 19 · How each Harbor task is built, and where unverifiable ones get dropped.

NO JUDGE ANYWHERE

The single most important property of this suite is that nothing in the reward path asks a language model for an opinion. The grader tries exact match, then numeric comparison within `atol` and `rtol`, then list and percent normalisation, then a symbolic comparison, and prints one float. The earlier generation had a `gpt-4o-mini` judge tier as a fallback, and every task that depended on it was dropped rather than carried forward.

This is why the answers are typed. Across the 5,000 training tasks the reward modes fall out as numeric 2,906, exact short string 1,409, list 367, flexible 152, boolean 127 and comma-

separated list 39. Nearly three in five tasks are a number compared within a tolerance, which is about as unambiguous as a reward gets, and it is the reason the reward-hacking chapter of the previous guide does not need repeating here.

WHAT SHIPS

Suite	Tasks	Purpose
`data-agent-harbor-train`	5,000	training
`data-agent-harbor-test`	250	held out, deliberately skewed harder
`data-agent-harbor-eval`	144	validation during training
`data-agent`	5,394 rows	the same tasks flat, one row each, `load_dataset`-rea
`data-agent-sft`	4,677	verified trajectories for supervised fine-tuning

The first three are Harbor environments, meaning a runnable container per task. The fourth is the same content as plain rows, which is what you want for inspection or for a framework that is not Harbor. The fifth is different in kind: 4,677 complete agent trajectories that reached the correct answer under the deterministic grader, one per task, in TRL's conversational format with the bash tool attached. 2,608 of them were produced natively in that format and 2,069 were converted from a second agent's format, and the set is contamination-checked against the held-out splits, with no leakage.

Two trajectory sources for one dataset is a small version of this article's whole argument, and it happened for ordinary engineering reasons rather than principled ones.

DIFFICULTY, HONESTLY

Each task carries a level from 1 to 5 and a coarser tier. The distribution is the part of a synthetic suite most worth reading sceptically, so here it is in full:

Split	Easy (L1)	Medium (L2 to L3)	Hard (L4 to L5)	Mean level
train, 5,000	1,433	2,845	722	2.12
test, 250	33	118	99	3.00
eval, 144	16	74	54	2.90

Training skews easy, which is what a notebook corpus gives you: 29 percent of training tasks are a single line of pandas. The held-out set was rebalanced in the other direction on purpose, so it is measurably harder than what the model trains on, and every one of the twenty L5 tasks in the whole collection sits in that 250-task test split rather than in training. A model that improves here has learned to do short data-analysis tasks reliably. It has not learned to do research, and the test split is arranged so that the difference shows up rather than hides.

One caveat to carry forward. The training split draws on 471 distinct Kaggle datasets, so the same underlying data appears behind many different questions. Task-level separation between splits is not the same as dataset-level separation, and any generalisation claim should say which one it means.

What ran, and on which suite

Here the article has to be careful, because the published suite and the measured results are not the same generation of the work.

Everything measured below ran on v1: the earlier 2,238-task training set and 366-task evaluation set, built before the LLM judge was removed. The rebuilt judge-free suite described above, the 5,000/250/144 one under HuggingEnvs, has not been rerun yet. So the numbers are real, they were expensive to get, and they are measured against a grader that has since been replaced.

How much does that matter? It is quantifiable, so here is the number rather than a hedge. Re-grading all 366 v1 evaluation tasks under the deterministic grader, 262 pass cleanly, 84 would fail because they only ever passed through the judge or through the compound-number parsing bug, and 20 were pure judge tasks. That is 104 of 366, 28 percent, graded more softly than the current grader would allow. Every v1 pass rate below should therefore be read as optimistic, with the softness concentrated on list and free-text answers rather than spread evenly.

What the evaluations said

Before any training, the suite was used to evaluate a range of models across four harnesses: `bash`, `jupyter`, `seta` and `opencode`. All four are code-execution harnesses, differing in whether the model runs shell commands, executes notebook cells, or works through a heavier multi-turn agent.

Three findings, and the first one cuts against this article’s own thesis, which is exactly why it is here.

Harness choice barely matters once the model is big enough. From 4B upward, the spread across all four harnesses is only two to three points. At 2B it widens to six. That is the opposite of the frontier-scale story in the introduction, where harness spread was widest for the open models, and it is a useful correction: the sensitivity is not uniform, it concentrates where capability is thinnest. Small models are the ones that need harness diversity most, which is convenient, because small models are what this project trains.

The cost difference is enormous even when the score is not. `opencode` burns 258 to 306 percent more tokens per passing task than the `jupyter` harness, which is consistently the cheapest. Two harnesses can agree to within a point on accuracy and differ by three to four times on what the run costs. Anyone comparing harnesses on score alone is reading one of two columns.

The task set is not hard enough on its own. Running the same models on DABstep costs them 40 to 48 points relative to Data-Agent v1, and the easy-to-hard cliff inside DABstep is about 50 points even for a 9B model. Scaling flattens too: 9B beats 4B by 14 points on the harder benchmark while costing about half of what 27B or 35B-A3B do, and returns are thin above that. The 250-task test split in the rebuilt suite was skewed harder specifically because of this result.

The full sweep, including per-harness token costs, is browsable in the evaluation Space.

Training a small model

Three GRPO runs, all full-epoch at 1,119 steps over the 2,238 v1 training tasks, using TRL's synchronous GRPO with Harbor for the task spec and E2B for the sandboxes. The models are from the Qwen 3.5 family.

Run	Wall clock	pass@4	pass@1
2B, tasks in random order	~22h	28% to 60%	10% to 41%
2B, difficulty-ranked curriculum	~22h	28% to 60%	10% to 40%
4B, tasks in random order	~27h	74% to 79%	60% to 64%

The 2B result is the encouraging one: pass@4 roughly doubles and pass@1 goes up about fourfold off a weak base. That is what you want from a first run, and it is the answer to whether a model this size has anywhere to go on this task.

Two details are more interesting than the headline. The curriculum run and the random-order run land on exactly the same final score, 0.603 in both cases. Feeding tasks easy-to-hard changed the shape of the curve, slower at first and steeper later, and changed nothing about where it ended up. Curriculum ordering is the kind of intervention that looks obviously beneficial until it is measured against a control.

And the 4B run plateaued early, peaking around step 200 at 0.802 and then holding a flat 0.75 to 0.80 band for the remaining nine hundred steps. It was already solving most of the suite before training started, so there was little headroom, and most of the compute bought nothing. That is an argument about the task set rather than about the model, and it is the second reason the rebuilt test split is weighted harder.

Trackio has the raw logs for these runs, at `AdithyaSK/DataAgent_harbor` and `AdithyaSK/harbor-data-agent`.

The part that was actually hard

None of the above was the difficult bit. The difficult bit was keeping a run alive for twenty-two hours. Leaked and ghost sandboxes, session timeouts, and account-wide rate limits between them required a lot of custom crash-recovery, retry and cleanup logic that was not in anybody's plan, and none of it is specific to this task or this model. It is the tax on every trainer that hand-rolls its own sandbox lifecycle, and it is a large part of the argument for putting something like OpenEnv in the middle rather than teaching each trainer to manage containers.

What is not done

Stated plainly, so nothing above is read as more finished than it is.

Nothing has been rerun on the rebuilt judge-free suite: no supervised fine-tuning run on the 4,677 trajectories, no GRPO run on the 5,000 training tasks, and no cross-model evaluation on the 250-task test split. That last one is designed and the models are compatibility-checked, seventeen of eighteen returning clean tool calls through the inference router, but sandbox throttling stalled the run.

The three training curves above come from the run logs rather than from a committed report, so treat them as reported results awaiting a reproducible artifact. The evaluation numbers are from the checked-in evaluation reports.

And the multi-harness claim this article is built on has not yet been tested here. The four harnesses were used to *evaluate*, not to train across. Training the same task set through several harnesses at once, which is the whole point, is the next run rather than a finished one.

OpenEnv × Harbor

What is Harbor?

Harbor comes from the people who built Terminal-Bench, and its origin story is the useful part. They shipped a benchmark, watched what people actually did with it, and found the container format being used for things it was never designed for. Their own words:

> *"When we released Terminal-Bench in May, we were surprised to see it used in unexpected ways like building custom evals, optimizing prompts, running RL, generating SFT traces, and CI/CD agent testing. We also learned that defining and managing containerized tasks at scale is hard. We built Harbor to make it easy."*

So Harbor is a framework for evaluating and optimizing agents in container environments. What it contributes here is one thing done well: it decouples the task from the agent and from the machine the agent runs on, thoroughly enough that all three vary independently.

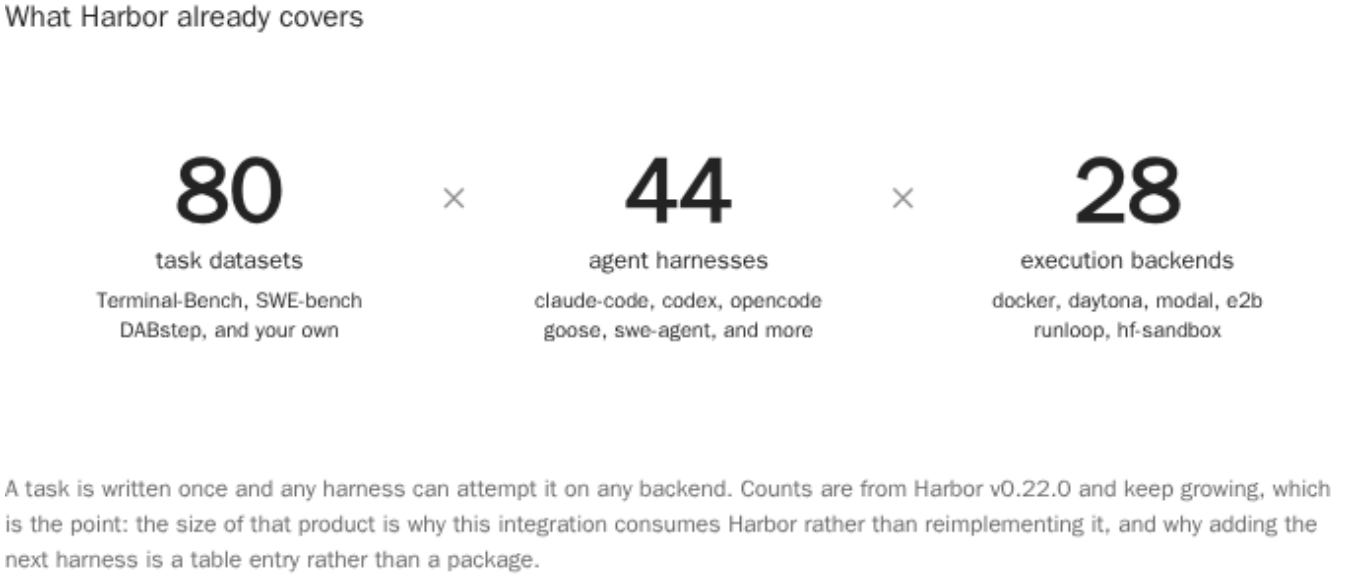


Figure 20 · Three axes that vary independently.

The interface an agent implements is deliberately small. A Harbor agent declares its name and version, sets itself up inside a container, and runs against a task, with both setup and run

handed the environment they execute in. That is nearly the whole contract, which is why adding a harness is a file rather than a project.

The one property that makes the rest of this section possible is stated flatly in Harbor's own documentation:

> *"Harbor tasks are independent, isolated, reproducible pieces of code. Harbor tasks have no dependency on the Harbor framework. They can easily be plugged into any framework that supports the Harbor format."*

A task is an instruction, an environment and a test script. Nothing more. That is why it can be served to something Harbor has never heard of.

> *Note: Harbor calls its execution backends environments, which collides with how this article and the previous guide use the word, and with OpenEnv's provider, which is the thing that hosts an environment server. Harbor knows: its own docs say "It's a bad name, we know. But it's too late to change now." Read Harbor's environment as the box the agent runs in.*

What a task looks like

Abstractions are easier to trust once you have seen one instance, so here is a single task from the Data Agent training suite, unedited.

One Harbor task, in full

tasks/0000_324_324947_qa_1/

task.toml	the spec task name and description, metadata including the gold answer and difficulty, the environment shape, the verifier, and the agent timeout
instruction.md	the prompt exactly what the agent is shown, including where its files live and how to format an answer
environment/Dockerfile	the image what the sandbox is built from
environment/pull_bucket.py	a pre-agent hook run by the healthcheck after the container starts and before the agent does, to pull this task's data in
tests/test.sh	the verifier entrypoint reads the agent's answer, and with no answer writes 0.0 and exits cleanly rather than failing
tests/grader.py	the reward exact match, then numeric tolerance, then list and percent normalisation, then symbolic comparison, printing a single float. No LLM judge anywhere in it

HOW THE REWARD GETS OUT

1. the agent writes /workdir/answer.txt
2. test.sh reads it, or writes 0.0 if it is missing
3. grader.py prints one float
4. the float lands in /logs/verifier/reward.txt
5. Harbor reads that file and returns it with the trial

5,000 TASKS BY DIFFICULTY

L1 1433 · one-line filter or aggregation L2 2256 · filter, groupby, aggregate, a few turns

L3 589 · multi-step pandas, joins, light feature work L4 722 · ML training, complex stats, non-trivial pipelines

L5 0 · held back entirely for the test split

One task from HuggingEnvs/data-agent-harbor-train, unedited. Every task in the suite has this shape, which is what makes it servable: nothing about it names a harness, a sandbox or a trainer.

Figure 21 · What a Harbor task is on disk, and how its reward gets out.

Two details in that layout matter later.

The first is `environment/pull_bucket.py`, wired in through the container's healthcheck. Harbor runs it after the container starts and before the agent begins, which is how each task pulls its own Kaggle dataset instead of baking gigabytes into an image. Small plumbing, and it is the difference between a suite of five thousand tasks being buildable and not.

The second is the failure branch in the verifier. If the agent never wrote an answer, `test.sh` writes `0.0` and exits cleanly rather than erroring. That looks pedantic until you are holding a training batch and cannot tell which rows are wrong answers and which are dead sandboxes.

Notice what the task does not contain: no harness, no sandbox backend, no trainer. It says what to do and how to score it, then stops. That silence is the whole opening.

Serving Harbor through OpenEnv

If Harbor already runs agents against tasks, what is left to build?

Harbor runs an agent and hands back a verdict. A trainer needs something else: the exact tokens the policy emitted and the probability it emitted them with. Harbor's own RL documentation names the two ways to get them:

> *"There are two strategies for collecting tokens: 1. Intercepting tokens from a vLLM server 2. Returning tokens as part of the agent result metadata."*

Strategy two is cleaner and only works for harnesses that have been modified to cooperate, which in practice means Terminus 2. Strategy one works for anything that speaks an OpenAI-shaped API, which is every harness in the matrix. This integration takes strategy one, and the rest of the section is the consequences of that choice.

WHY AN HTTP BOUNDARY, NOT AN IMPORT

The in-process version was tried first, and the shape of its failure is the architectural argument. Running agent rollouts inside the trainer means an exception thrown five layers down inside somebody else's agent binary propagates into the training loop. On one GPU that is an annoyance. Across ranks it is a hang: one rank dies, the others wait at the NCCL barrier for a peer that is never coming, and the job wedges until a human notices. TRL's in-process Harbor integration runs to roughly five hundred lines, with nearly every environment call individually wrapped in `try` and `except`, and that is the maintenance cost of defending against this rather than removing it.

Behind an HTTP boundary the failure class cannot occur. A rollout that dies returns a result that says so, carrying `ok=False` and an error, and the trainer handles a value rather than catching an exception. The same boundary collapses evaluation and training onto one code path, so hardening either hardens both.

THE FOUR COMMANDS

Command	
`openenv harbor info`	Reports what this machine can actually run: whether the endpoint retu
`openenv harbor rollout`	Rollouts with no environment server involved. Also the debugging path
`openenv harbor serve`	The environment server. Task API for discovery, one long-running rollou
`openenv harbor push`	Deploys that same server to a Space, with configuration as Space vari

```
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 modal
openenv harbor serve   --llm-url $LLM --dataset org/train,org/eval
openenv harbor push    --llm-url $LLM --dataset org/train,org/eval --repo-id you/harbor-env
```

`--llm-url` is required, with no default and no environment-variable fallback. That unfriendliness is deliberate, and it is the same instinct as refusing to start without token ids. An endpoint that cannot return them still answers every request perfectly well: text, a 200, plausible usage numbers. What it does not return is anything trainable, so every row rebuilt downstream is empty, training silently does nothing, and the first symptom is a loss curve that never moved, noticed days later. That failure has no loud edge, so the check has to be at the front.

Three levels of capture

tokens	prompt ids, sampled ids, aligned logprobs	trainable
logprobs	logprobs, but no ids to attach them to	not trainable
text	neither	not trainable

An endpoint missing the token-id flags still returns text, a 200 and plausible usage numbers. It just returns nothing trainable, so the first symptom is a loss curve that never moved, noticed days later. That is why the server refuses to start instead of degrading quietly.

Figure 22 · Only one of them can be trained against.

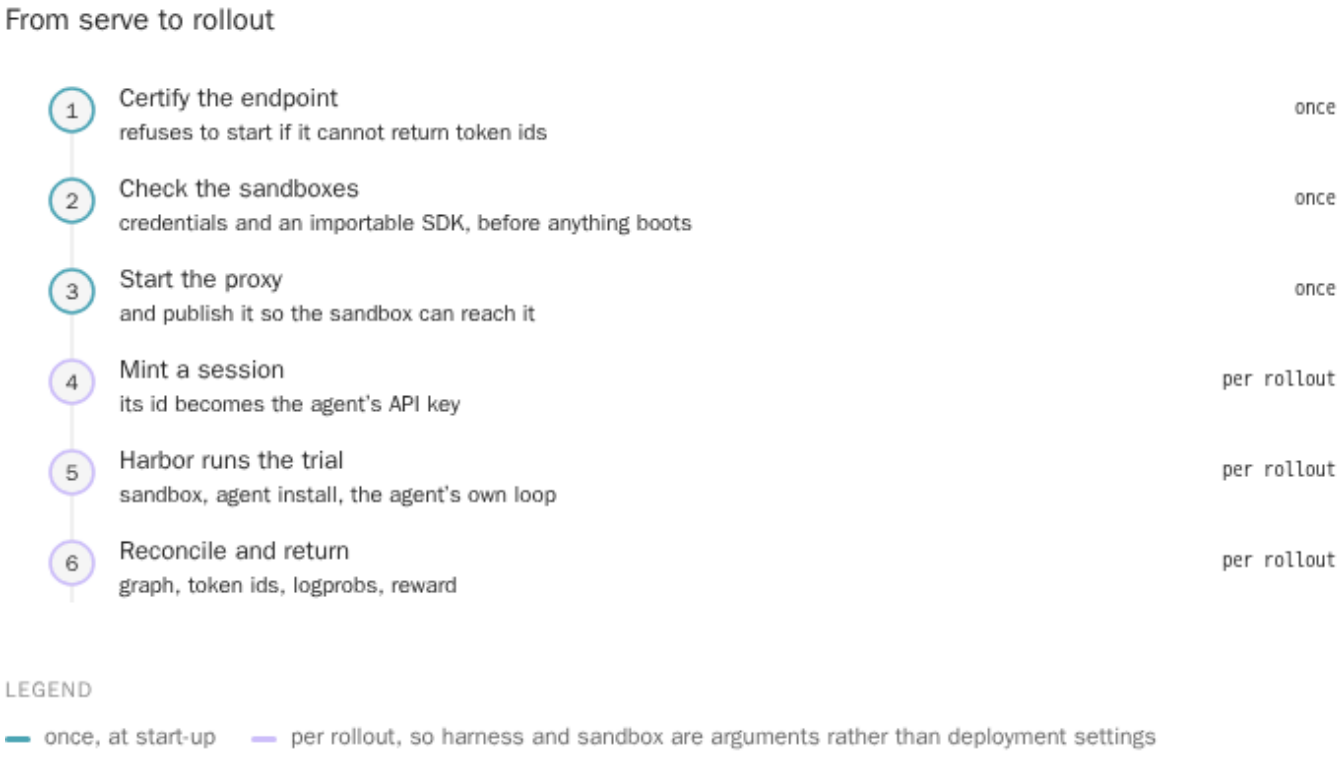


Figure 23 · From one command to a validated trajectory.

WHAT IS ACTUALLY NEW

The division of labour is the point. Harbor already had the tasks, the verifiers, the harnesses, the backends and the trial concurrency. OpenEnv already had the environment server, the Task API, the typed client, the sandbox transport and the CLI. What is new is small: a capture proxy, a rollout graph, and a token contract.

The one piece of per-harness knowledge in the whole stack is a table of seams, and the module that holds it says why it is a table rather than code:

> *"For every Harbor agent the only thing that differs is which env var or config key carries the base URL and the API key. Sandbox, capture, stitching, masking and validation are identical downstream."*

Claude Code takes `ANTHROPIC_BASE_URL`. Codex takes `OPENAI_BASE_URL` and an `OPENAI_API_KEY` that is really the capture session id. opencode takes no environment variables at all and needs a provider block instead. Each of those is one row. The alternative, which the ecosystem was drifting toward, is one environment package per coding agent, each carrying its own interception proxy that has to be correct about token ids. Get that wrong once and you are training on tokens the model never emitted.

A seam is only marked validated after an end-to-end run passes both the capture contract and the independent trace cross-check. Everything else is untested however plausible it looks, and

one harness has already been removed outright rather than left in as aspirational: it failed five out of five tasks before the agent even started, because its install script exited 127.

The parts that are easy to get wrong

Three details decide whether what comes out is trainable or merely plausible.

Harness and sandbox are per-rollout arguments, not deployment settings. One server covers the whole matrix. That reads as a convenience feature and is really the entire multi-harness story: if the harness were fixed at deploy time, varying it during training would mean standing up a separate server for each one, and nobody does that at scale.

The reward is a scalar chosen by an explicit rule, and the rule refuses rather than guesses. Harbor's verifier produces a dictionary of named floats and OpenEnv wants one number, so the dictionary travels through verbatim and the scalar is picked by rule: one key, or one named `reward`, otherwise fail and demand that the caller says which. Combining keys automatically would be inventing reward semantics, and there is a scar behind that decision. An earlier run carried a `+0.2 for submitting anything` term, the policy learned to submit immediately, training reward looked healthy throughout, and held-out evaluation fell from 0.740 to 0.178.

`reward=None` is not zero. It means the verifier never ran. Conflating the two makes a dead sandbox look like a wrong answer, and a run that quietly learns from broken infrastructure is very hard to diagnose afterwards. This is the same care the task's own `test.sh` took, carried all the way up to the result object, where `solved` is defined as graded *and* positive rather than simply non-zero.

There is a fourth failure worth including because it is so unobvious. Setting `OPENAI_API_KEY` on one harness broke grading rather than execution: Harbor forwards the variable into the sandbox, where the Data Agent grader's own LLM judge checks for it, finds the capture session id instead of a real key, gets a 401, and silently skips the judge. Every answer that was right but not an exact string match scored zero. Not a crash. Reward corruption.

Validating against something that is not us

Every check described so far is one we wrote, run on data we captured, using assumptions we chose. A proxy can be perfectly self-consistent and completely wrong.

ATIF is the way out of that circle. It is Harbor's own trajectory format, written by the harness independently of anything the capture layer does, so reconciling the two is the only test

available that is not self-referential. Compare call by call: turn counts, per-call completion token counts, and which calls the harness itself considers real agent steps rather than internal bookkeeping. A rollout comes back marked as a match, a mismatch, or as having no trace to compare against.

When they agree, they agree precisely:

```
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

```

That agreement is worth more than it looks. The masking, the turn segmentation and the prefix stitching are all being checked at once, by a path that shares none of our assumptions.

It has already earned its place. One harness sent an empty tools array, received a 400 from the inference server, and had its trajectory truncated, leaving behind a rollout graph that looked perfectly well formed. Nothing in the captured data was visibly broken. The mismatch is what surfaced it. The check runs on ingest rather than export, for the obvious reason: a turn whose logprobs are misaligned has to be caught while we still know which turn it was.

The endpoint of this is neater than two formats being reconciled forever. ATIF already has optional fields for logprobs and completion token ids that harnesses leave empty. Filling them in gives one artifact that is trace, SFT dataset and RL data at the same time.

Running it somewhere other than a laptop

Locally this uses two ports. The environment server faces trainers and browsers; the capture proxy faces the sandbox and is the only one published. Sharing one port would expose the environment server the moment the proxy became reachable.

Hosted, that inverts. A Space has one port and one URL, so the proxy is mounted on the environment server's own application at `/capture`, with nothing forwarded. The Space has to be public, because a private one requires an authorization header that the agent inside the sandbox does not send. That is safe here only because the proxy rejects any caller without a registered session id, so a public mount is not an open relay. The `--private` flag exists and its help text is honest about what it is for: the sandbox then cannot reach the proxy at all, so rollouts are impossible and you are only parking a deployment.

Where this sits

Serving Harbor tasks to a trainer is not a new idea, and it is worth being precise about what is different here rather than claiming novelty that is not there.

	How it consumes
TRL	In-process, external agents only. Harbor's own installed agents are unsupported, becaus
SkyRL	In-process, and the integration Harbor itself endorses. Reads token ids and logprobs ou
NeMo Gym	In-process, with a compatibility layer, plus its own sandbox for HPC.
verifiers	Consumes Harbor's task format, then runs rollouts on its own runtime. Harbor is a task
Polar	Does wire-level capture across dialects, and has no Harbor.

Read down that column and the gap is specific. Everyone running Harbor in-process depends on the harness handing tokens back, which limits them to harnesses modified to do so. The one project doing wire-level capture across dialects is not connected to Harbor at all. Nothing yet serves Harbor tasks to a trainer over an HTTP boundary with wire-level capture across four dialects, which is exactly the combination that makes the harness a per-rollout argument.

> *Status. The integration described here is OpenEnv pull request #1036, open and unmerged at the time of writing. Harbor's own RL submodule, which would give tasks a* `step` *and* `grade` *interface directly, is also an open pull request. Both numbers and both designs may move.*

Evals

We evaluate Qwen3.5-2B on 250 fixed test tasks: 33 easy, 118 medium and 99 hard. Each checkpoint runs once under OpenCode, Claude Code, Codex and Mini-SWE-Agent: 1,000 task–harness cells, pass@1. Aggregate scores weight the four harnesses equally. Only complete, accepted cohorts enter the comparison. Missing cells are not counted as failures, and a graded zero is not replaced by a later retry. Historical retry and provenance limitations are recorded in the analysis.

The base model already changes score with the harness. Under the Harbor/E2B protocol, OpenCode scores 10.8%, Claude Code 16.8%, Codex 16.4%, and Mini-SWE-Agent 14.4%, averaging 14.6%. The native OpenCode training run has a separately measured four-harness

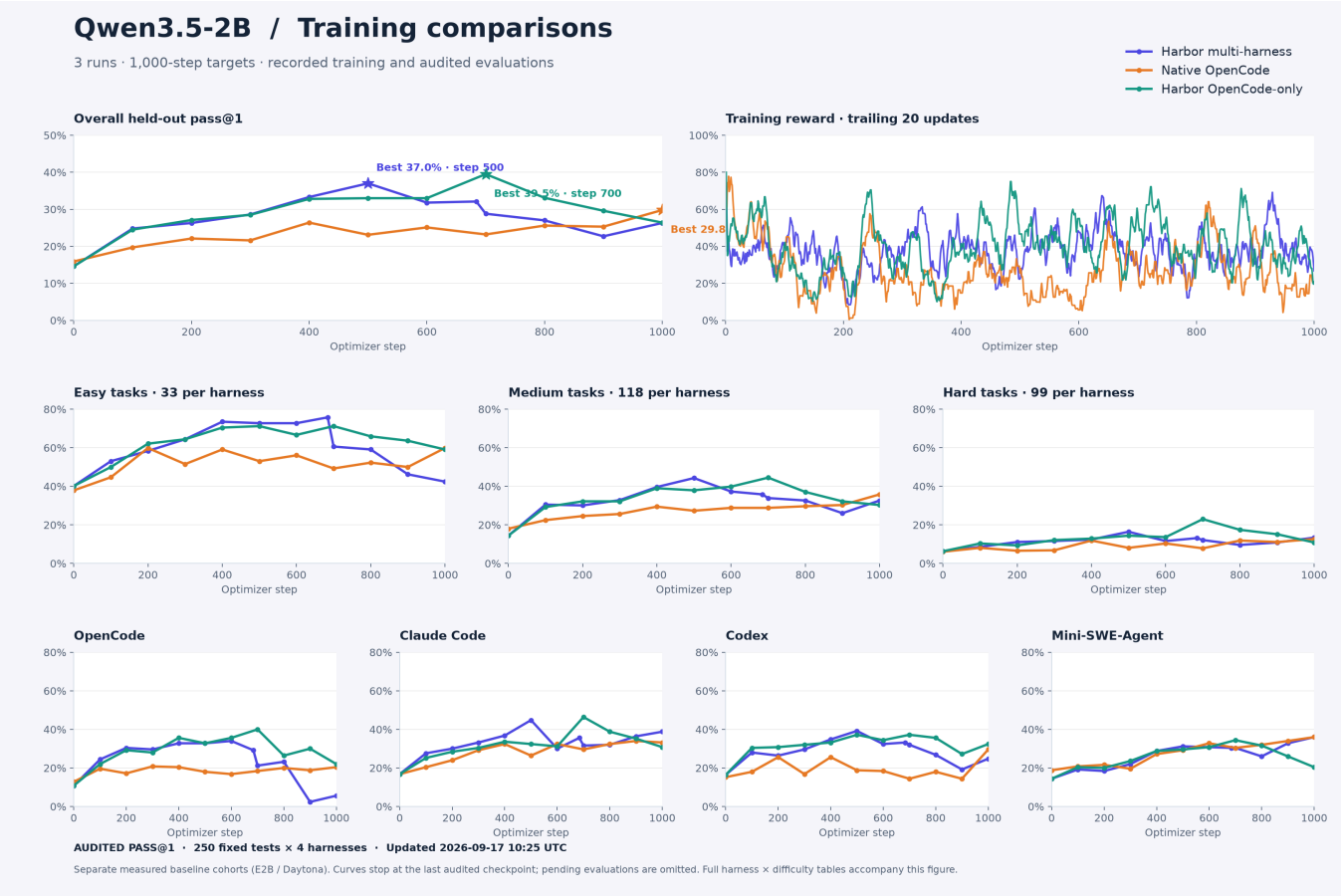
Harbor/Daytona baseline of 15.9%. These are different measured cohorts; we retain both rather than assigning every run the same starting score.

The checkpoint report includes correct counts and scores by harness and difficulty at every checkpoint. SETA's native bash evaluator is a separate 250-cell protocol: 18.8% → 38.0% at step 150. It is not a fourth line in the four-harness comparison below.

Training runs and checkpoint results

Three asynchronous Qwen3.5-2B runs reached 1,000 optimizer steps: Harbor with four training harnesses, native OpenCode, and Harbor with OpenCode alone. All three are evaluated under the same four harnesses. These are observational comparisons; backend, task exposure, rollout filtering and resume history differ. They do not isolate a causal benefit of multi-harness training.

Run	Baseline pass@1	Best measured checkpoint	Final step 1,000
Harbor multi-harness	14.6%	37.0% at 500	26.3%
Native OpenCode	15.9%	29.8% at 1,000	29.8%
Harbor OpenCode-only	14.6%	39.5% at 700	26.4%



The full table and counts include steps 900 and 1,000, whose recovery evaluations are now complete. “Best” is selected from the measured checkpoints on this test set; it is not an independently selected validation checkpoint. The peak difference of 39.5% versus 37.0% is not clearly separated by paired task-level uncertainty.

Equal steps do not mean equal training exposure

Run	Distinct training tasks	Supervised tokens	Zero-fresh-gradient steps
Harbor multi-harness	482	22.15M	349/1,000
Native OpenCode	566	5.33M	583/1,000
Harbor OpenCode-only	523	14.63M	380/1,000

The 1,000-step limit did not cover the full 1,000-task pool. Uniform-reward groups supply no fresh GRPO contrast, although optimizer momentum can still change weights. Captured training rows also differ from independent tasks: prompt forks preserve supervision but duplicate context. See the token and tool accounting for definitions, coverage and per-harness breakdowns.

OpenEnv × Harbor × TRL

TRL requests a task rollout from OpenEnv. Harbor starts the harness in a sandbox; the harness sends model requests through the capture proxy to vLLM. The proxy records engine prompt IDs, generated token IDs and aligned log probabilities. The environment returns the verified reward and captured trajectory, which the trainer converts into masked training samples. A rollout may produce several samples when its prompt history forks; those rows must remain associated with their originating rollout and reward group.

Training and evaluation use the same task-serving interface. Checkpoints are saved every 50 steps and scheduled for evaluation every 100 steps on separate inference resources. Evaluation retains task identities and the first accepted grade per cell. Isolation removes competition for the training GPU; sandbox capacity and proxy throughput still require their own limits and measurements.

Reproducing it and inspecting artifacts

- Reproduction guide: local/Slurm and Hugging Face Jobs/Spaces entry points.
- Results and qualifications: completed scores, accounting definitions and limitations.
- Public artifact index: published checkpoints, report downloads and qualification receipts.
- Trackio comparison: training metrics and checkpoint evaluations.
- Harbor environment, native OpenCode environment, and SETA environment: the three environment implementations.

The Data Agent PR combines the reproduction code and this article. The integration work is tracked in OpenEnv #1036 and TRL #6947. Use the pinned source and qualification records for a run rather than assuming a moving PR head is the same build.

Conclusions and observations

Results snapshot: September 18, 2026; completed comparisons were audited September 17.

Training in one harness can transfer to others. Harbor OpenCode-only reaches 39.5% aggregate pass@1 at step 700, including 46.4% under Claude Code and 40.0% under OpenCode. Native

OpenCode improves all four evaluation harnesses and ends at 29.8%. The evidence does not support a simple claim that one-harness training only improves its own harness, or that multi-harness training necessarily wins.

The late declines have different observable signatures. Multi-harness falls from 37.0% at step 500 to 26.3% at step 1,000. Across those evaluation cohorts, output-truncated rollouts rise from 9/1,000 to 556/1,000. Its training responses get longer while emitted tool calls fall. Harbor OpenCode-only instead uses more tools but submits answers less reliably; output truncation remains rare. These patterns suggest different follow-up experiments, not a single proven cause.

Resume accounting and reward contrast matter. After the multi-harness restart at step 684, 1,575 of 1,579 admitted rollouts revisit previously seen tasks. That bug changes late exposure, but cannot explain the initial decline after step 500, which precedes the restart. Across the three runs, 35–58% of optimizer steps have no fresh gradient from reward contrast. Those steps still consume rollout and training compute. Compare token exposure, task coverage and useful gradient steps, not just nominal optimizer steps.

Successful token capture is necessary, but not sufficient, for a reliable run. TiTO preserves which tokens were generated and their behavior log probabilities. It does not by itself solve prompt-fork weighting, sample budgets, uniform-reward groups, serving interruptions or evaluation coverage. Each needs separate telemetry and qualification.

A separate continuation from checkpoint 500 uses 500 hard tasks for two passes. Its evaluation transport is being qualified through an authenticated HF Jobs endpoint, with the environment and capture proxy inside the job and task sandboxes external. It is not included in the completed results above: partial qualification cells are not a benchmark score, and no curriculum improvement is claimed yet.

The next comparison should preserve task coverage across resume, match train/eval output budgets, and compare harness mixtures at matched token exposure. The analysis, training accounting, and accepted checkpoint counts make those follow-ups traceable to the recorded runs.

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},

}
```

References

1. Anthropic. (2024). *Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet*. https://www.anthropic.com/engineering/swe-bench-sonnet ↑
2. ByteDance Seed. (2025). *Seed-OSS-36B-Instruct Model Card*. https://huggingface.co/ByteDance-Seed/Seed-OSS-36B-Instruct ↑
3. DeepSeek-AI. (2025). DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. *arXiv Preprint arXiv:2512.02556*. ↑
4. 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*. ↑
5. Kimi Team. (2025). Kimi K2: Open Agentic Intelligence. *arXiv Preprint arXiv:2507.20534*. ↑
6. KwaiKAT Team. (2026). KAT-Coder-V2.5 Technical Report. *arXiv Preprint arXiv:2607.05471*. ↑
7. MiniMax. (2025). *MiniMax-M2 Model Card*. https://huggingface.co/MiniMaxAI/MiniMax-M2 ↑
8. 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*. ↑
9. poolside. (2026). LAGUNA M.1/XS.2 Technical Report. *arXiv Preprint arXiv:2605.27605*. ↑
10. 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*. ↑
11. Z.ai. (2025). *GLM-4.7*. https://huggingface.co/zai-org/GLM-4.7 ↑

1. 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*. ↑

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*. ↑
3. 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*. ↑