LITEEVO: AUTOMATED, COST-EFFICIENT HARNESS EVOLUTION FOR GENERALIZATION TO UNSEEN TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

An LLM agent is defined by two things: the weights inside its model and the harness of components assembled around it. Harnesses are still handcrafted, and HarnessX, which evolves them automatically, starts each benchmark from a handcrafted harness, reports gains on the tasks it evolved on, and budgets 100 to 175 million meta-agent tokens per benchmark. We propose LiteEvo, a lightweight harness-evolution algorithm whose tool-free meta-agents mine agent trajectories for reusable components, curate them into a versioned library, and compose each round's harness from it, starting every benchmark from the same neutral harness and never naming the benchmark. Evolving on the graded tasks of five agentic benchmarks with a frozen Qwen3.5-9B, LiteEvo lifts pass@2 by 10.5 to 67.7 pp and reaches comparable or higher pass@2 than a reproduction of HarnessX (71.0 against 67.3 on average) at 13.0× lower mean API cost. Harnesses evolved on train tasks keep their gains on unseen test tasks of four benchmarks, and LiteEvo also lifts Claude Code with Sonnet 4.6 by 1.2 to 71.4 pp.

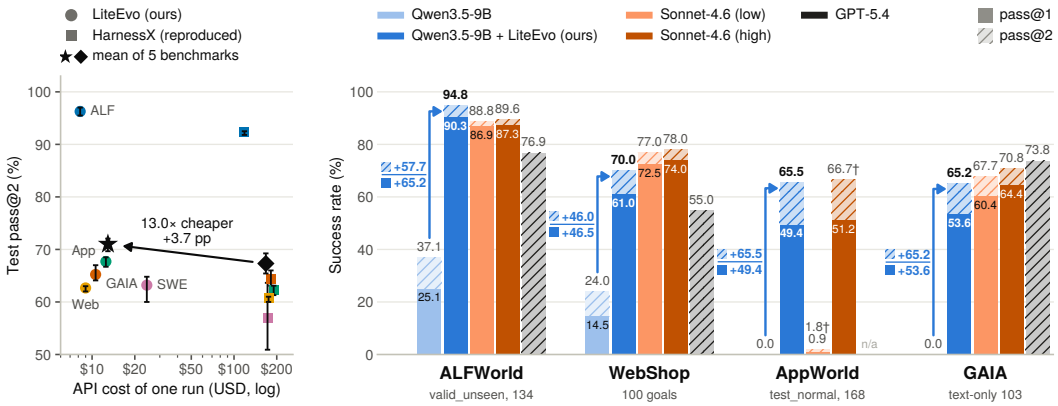
1 INTRODUCTION

Every language-model agent runs inside a harness, the layer that turns the model's replies into actions and the environment's responses into its next input. Its prompt, action and observation processing, and loop settings can decide whether a task is solved: given only the task text, a frozen Qwen3.5-9B solves none of AppWorld's test tasks, but with a harness that LiteEvo evolves on separate training tasks, the same model solves 65.5% of them within two attempts (Table 1). Changing the harness needs no gradient step and applies to closed models as well, whereas training the weights of a coding agent with reinforcement learning takes 512 GPUs for about 32 hours (Wei et al., 2025a).

Recent work automates harness design with a frontier model that acts as a meta-agent, reading the agent's traces and rewriting its harness (Chen et al., 2026; Lee et al., 2026; Lou et al., 2026). HarnessX (Chen et al., 2026) applies this idea to five agentic benchmarks, but at a high price, with a budget of 100 to 175 million meta-agent tokens per benchmark, and with human knowledge built in: it starts each benchmark from a handcrafted harness, so every new environment needs new human effort, and the gains it reports mix what evolution found with what the seed already knew. Its paper also reports gains at the best round on the tasks it evolved on and does not test whether an evolved harness helps on tasks it has not seen.

Other systems share some of these limits. Methods in which a model designs agents automatically report search costs of hundreds to tens of thousands of dollars per run (Hu et al., 2025; Robeyns et al., 2025; Zhang et al., 2026c), and each of the closest harness-evolution methods tells its optimizer about the benchmark in some form (Table 6): Meta-Harness (Lee et al., 2026) guides its proposer with a hand-written skill for each domain, AutoHarness (Lou et al., 2026) gives its refiner the game's name and description, and HarnessCompass (Zhang et al., 2026d) puts software-repair hints into its meta prompt. Three recent methods start from a minimal harness or score on held-out tasks, but each covers at most two of our five benchmarks (Zhang et al., 2026d;b; Yang et al., 2026). It therefore remains open whether a cheap evolution loop that knows nothing about the benchmark, neither through its starting harness nor through its instructions, can produce harnesses that help on unseen tasks across agentic benchmarks as different as ours.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) (b)

Figure 1: LiteEvo reaches comparable or higher pass@2 than our reproduction of HarnessX at 13.0× lower mean API cost and lifts a frozen 9B agent to the level of frontier agents on ALFWorld and AppWorld. (a) Test pass@2 against the API cost of one evolution run, both methods transductive. Error bars span the three test runs, and the black markers average the five benchmarks. (b) Qwen3.5-9B given only the task text, and with the harness LiteEvo evolves (inductive, except on GAIA), against Claude Code with Sonnet 4.6 at low and high effort and GPT-5.4 with the static, unevolved harness that HarnessX reports (Chen et al., 2026), on HarnessX's own task sample for WebShop. Solid bars give pass@1 and hatched bars pass@2. †: Sonnet 4.6 is told how to finish a task; the agent under LiteEvo never is.

We address this question with LiteEvo, a harness-evolution loop designed for low API cost and a benchmark-blind start. Every benchmark starts from the same neutral harness, a 788-character description of the turn protocol with no benchmark-specific content, and the four meta-agents are never told which benchmark they work on: a filter masks names and identifiers in everything they read, and their instructions forbid using recalled knowledge of a benchmark. Apart from this harness and the benchmark interfaces, the only parts we design by hand are the loop and the meta-agents' instructions, so the benchmark-specific content of a returned harness is learned from the agent's own trajectories. In each round, a single analyst reads a failure-oriented sample of these trajectories: the tasks that some attempts solve and others fail, and the lowest-scoring tasks that no attempt solves. Three more meta-agents turn its diagnosis into edits of a versioned component library and a new harness, for about five tool-free API calls per round in all; a pass@2 gate then decides whether to keep that harness (Figure 2). We evaluate LiteEvo both transductively, as HarnessX does, and inductively, on test tasks the loop never sees. Our contributions are as follows:

- LiteEvo, a generic harness-evolution method that needs no benchmark-specific seed: it starts every benchmark from the same neutral harness, never tells its four tool-free meta-agents which benchmark they work on, and evolves the harness through failure-oriented group sampling, a versioned library of typed components and a pass@2 gate.
- Comparable or higher pass@2 for a small share of the API cost: with a frozen Qwen3.5-9B on ALFWorld, WebShop, AppWorld, GAIA and SWE-bench Verified, LiteEvo reaches a mean pass@2 of 71.0 against 67.3 for a same-stack reproduction of HarnessX when both evolve on the graded tasks, at a 13.0× lower mean API cost (Figure 1).
- Generalization beyond the evolution tasks: harnesses that LiteEvo evolves on train tasks and selects on val tasks stay within 2.2 pp of the transductive ones on unseen test tasks of ALFWorld and AppWorld, exceed them on WebShop and SWE-bench Verified (70.0 against 62.7 and 66.7 against 63.2), and reach 48.6 pass@2 on AppWorld tasks that need apps unseen in training, where the neutral harness solves none; transfer to an unseen benchmark remains open.

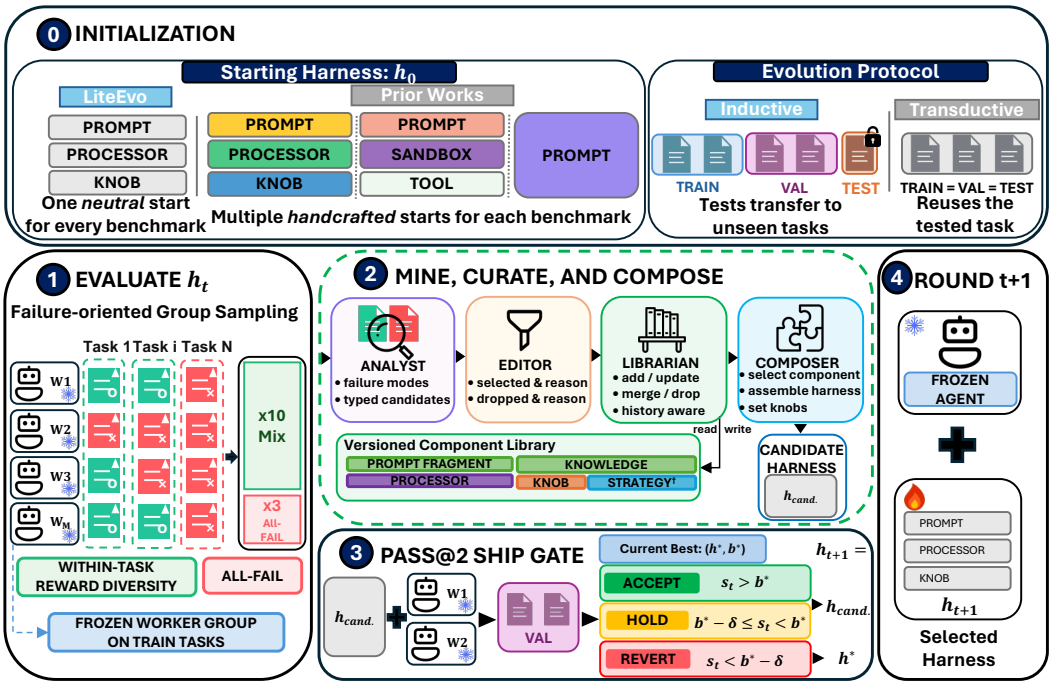


Figure 2: Overview of LiteEvo. (0) Every benchmark starts from the same neutral harness h_0. (1) Frozen workers run the current harness h_t on the train tasks, and failure-oriented group sampling keeps ten mixed-outcome and three all-fail task groups as evidence. (2) Four meta-agents mine failure modes, curate a versioned component library († marks the strategies, which advise the composer and never reach the agent) and compose a candidate. (3) A pass@2 gate on the val tasks accepts, holds or reverts the candidate. (4) The frozen agent runs the selected harness h_{t+1} in the next round.

2 PRELIMINARIES

Tasks and the Frozen Agent. A benchmark provides a set of tasks $\mathcal{T}$, each an interactive environment whose first observation states the task and which answers every action with a new observation. When an episode ends, the benchmark's own evaluator returns a success flag $y \in \{0, 1\}$ and a partial credit $f \in [0, 1]$. The agent, a frozen language model with fixed decoding, acts through a fixed runner: a multi-turn chat loop that passes the action in each reply to the environment until the episode ends or a per-benchmark turn cap is reached.

Harness. A harness $h = (p, \Phi, \kappa)$ is everything that may change between the frozen model and the environment. Here p is the system prompt, and Φ is a set of *processors*, short programs that the runner calls at fixed points of a turn. A processor can add a one-turn message before the model call, rewrite the action after it is extracted, or rewrite the observation after the environment step. The last part, κ, sets five runner *knobs*, which control the reply protocol, the listing of available actions, an observation length cap, a last-line action fallback and a display of the remaining turns. The environment, the turn cap, the decoding parameters and the model lie outside the harness and never change.

Metric and Regimes. We measure a harness by $\mathrm{pass@}k(h; \mathcal{T})$, the probability that at least one of k independent attempts at a task succeeds, averaged over a task set $\mathcal{T}$ and estimated with the unbiased estimator of Chen et al. (2021). The tasks fall into three sets: (i) *train* tasks $\mathcal{T}_{\text{train}}$, whose trajectories a method may read, (ii) *val* tasks $\mathcal{T}_{\text{val}}$, used to choose between harnesses, and (iii) *test* tasks $\mathcal{T}_{\text{test}}$, used to grade the harness a method returns. The *inductive* regime keeps the three sets disjoint, so that the test tasks stay unseen during evolution, whereas the *transductive* regime uses one task set for all three, as HarnessX does for all its results (Chen et al., 2026). As a floor, the *bare* agent receives only the benchmark's own task text, with no system prompt.

Algorithm 1 One run of LiteEvo with R rounds. Round t runs h_t: round 0 runs the neutral harness, and the loop picks $h_1, \ldots, h_{R-1}$ for the $R-1$ evolution rounds. In the transductive regime, the train and val tasks are the same, and s_t is computed from the train rollouts.

$\mathcal{L} \leftarrow$ three hand-written strategies; $\quad \mathcal{J} \leftarrow \emptyset$ ▷ library, journal
$(h^\star, b^\star) \leftarrow \big(h_0,\ \text{pass@2}(h_0; \mathcal{T}_{\text{val}})\big)$ ▷ best harness and its score
for $t = 0, \ldots, R-2$ **do** ▷ picks h_{t+1}
  $\mathcal{G} \leftarrow$ one group of M rollouts of h_t per task in $\mathcal{T}_{\text{train}}$ ▷ Section 3.2
  $A \leftarrow \textsc{Analyst}\big(\textsc{GroupSample}(\mathcal{G}),\ h_t\big)$ ▷ Algorithm 2
  $E \leftarrow \textsc{Editor}(A, \mathcal{J}); \quad \mathcal{L} \leftarrow \textsc{Librarian}(\mathcal{L}, E)$ ▷ Section 3.3
  $h_{\text{cand}} \leftarrow \textsc{Composer}(\mathcal{L}, E, \mathcal{J}, h^\star)$
  $s_t \leftarrow \text{pass@2}(h_{\text{cand}}; \mathcal{T}_{\text{val}})$, from M rollouts per val task
  if $s_t > b^\star$ **then** ▷ accept
    $h_{t+1} \leftarrow h_{\text{cand}}; \quad (h^\star, b^\star) \leftarrow (h_{\text{cand}}, s_t)$
  else if $s_t < b^\star - \delta$ **then** ▷ revert
    $h_{t+1} \leftarrow h^\star$
  else ▷ hold
    $h_{t+1} \leftarrow h_{\text{cand}}$
  end if
  append round t (scores, decision, failure modes) to $\mathcal{J}$
end for
return $h^\star$

3 LITEEVO

LiteEvo is a fully automated harness-evolution algorithm: given a frozen agent and a benchmark, it improves the agent's harness over R rounds, starting from a neutral harness h_0 that is the same for every benchmark and using no human input written for the benchmark (Figure 2, Algorithm 1). Round t runs the current harness h_t with M frozen workers on the train tasks. Four meta-agents then turn the resulting trajectories into a candidate h_{cand}: the *analyst* diagnoses failure modes and proposes components, the *editor* selects and checks them, the *librarian* records them in a versioned library $\mathcal{L}$ of components, and the *composer* assembles the candidate from $\mathcal{L}$, while a journal $\mathcal{J}$ keeps each round's scores, decisions and failure modes for later rounds. A pass@2 gate on the val tasks then compares the candidate's score s_t with the score $b^\star$ of the best harness so far, $h^\star$, and picks the harness h_{t+1} of the next round, either h_{cand} or $h^\star$. Round 0 thus runs the neutral harness, and each of the $R-1$ evolution rounds that follow runs the harness that the gate picked in the round before. In the inductive regime the train and val rollouts come from disjoint task sets, whereas in the transductive regime both come from the test tasks and the gate reuses the train rollouts.

The meta-agents read what the agent did and how its episodes ended but never act in an environment: each of their calls is a single request without tools that returns structured output. Apart from h_0 and the runner that connects the agent to each benchmark, the only parts we design by hand are the loop and the meta-agents' instructions, neither of which names or describes any specific benchmark; Appendix B lists all human inputs.

3.1 A NEUTRAL START AND BENCHMARK-BLIND META-AGENTS

LiteEvo starts all five benchmarks from one *neutral harness* h_0, whose 788-character prompt p states only the protocol of an episode: the agent reads an observation, replies with one action inside action tags, and has a limited number of turns. Where actions are code, the reply format allows a multi-line action. Since h_0 has no processors and uses default knob values, what a returned harness knows about its benchmark comes from evolution, apart from generic advice in the meta-agents' instructions. HarnessX, by contrast, starts each benchmark from a handcrafted harness (Chen et al., 2026); its GAIA prompt alone holds about 11,000 characters of hand-written search recipes, several of which name specific websites.

The meta-agents are blind to the benchmark as well. Before a meta-agent reads a trajectory, a filter removes task identifiers, split names and URLs, masks benchmark and dataset names, and shows the turn limit only as short or long, a label that at least two benchmarks share. The meta-agents'

instructions forbid relying on recalled knowledge of a benchmark, even if they recognize it, and Appendix B details how both rules are enforced.

3.2 FAILURE-ORIENTED GROUP SAMPLING

Each round evaluates the current harness with M frozen *workers*, each of which runs h_t once on every train task, so that every task yields M trajectories that form its *group*. The groups are the input of the analyst, the first meta-agent, but it does not read all of them. A group is *mixed* when some but not all workers solve the task, and *all-fail* when none does; with *failure-oriented group sampling*, the analyst reads the ten mixed groups with the highest mean partial credit and the three all-fail groups with the lowest. Each comes with all M trajectories and their outcomes, together with outcome statistics over the whole round and the prompt p that the agent ran (Algorithm 2). The two kinds serve different ends. A success and a failure in one mixed group share the task and the harness and differ only by sampling, so the step that separates them is one that the harness leaves to chance, and fixing it makes the harness more reliable (exploitation). An all-fail group instead shows a task that the harness does not yet make solvable, and asks for the insight that would (exploration). Without both kinds, evolution stalls where successes are rare and hard to reach, as on AppWorld: reading only all-success or only all-fail groups ends there at 16.9 and 12.3 pass@2, against 67.7 with both (Appendix D.6). The design echoes group-relative reinforcement learning, which learns only from groups whose rewards differ (Shao et al., 2024; Yu et al., 2025), and Training-Free GRPO, which turns such within-group contrasts into prompt-level experience (Cai et al., 2025); unlike them, LiteEvo also keeps all-fail groups, as targets for exploration.

3.3 THE META-AGENT STAGE

The library $\mathcal{L}$ stores typed components that fill the three parts of a harness $h = (p, \Phi, \kappa)$. Prompt fragments, which are instructions or worked examples, and knowledge components, which record facts about the environment seen in trajectories, go into the prompt p; processors go into Φ, and knob components advise the settings of κ. A fifth kind, strategies, advises the composer and never reaches the agent. Initially the library is empty except for three hand-written strategies, generic composing rules such as leaving the reply format to the runner (Appendix B). The edits of a round produce a new library version, and components record when they were selected and how often they fired. On AppWorld, for instance, the returned harness holds a worked login example as a fragment and the login API's credential fields as knowledge, and one of its processors restates the login steps after two authorization errors.

Analyst. The analyst first compares the trajectories within each sampled group and only then generalizes across groups. It returns a list of failure modes, giving for each one its share of the failures, the evidence and the mechanism behind it. For each mode, it also proposes candidate components of a given kind, written out in full. Its instructions ask for fixes that the harness can carry out, such as a worked example or a processor, rather than general advice to the agent.

Editor. The editor merges, rewrites or drops the analyst's candidates with a reason for each decision, reading a journal $\mathcal{J}$ of earlier rounds (their gate scores, decisions, failure modes and rejected components) so that it does not repeat a change that already failed. Every processor it keeps is compiled and replayed on 24 episodes of the round, and one that does not compile, raises an error or never fires is dropped; a separate review call then reads the outputs of each remaining processor and drops those that contradict its stated purpose.

Librarian. The librarian applies the editor's selection to the library: it sees all components with their usage statistics and returns operations that add a component, update or merge existing ones, or drop one. Each operation is validated again before it is applied: a processor must compile, a knob setting must parse, and no body may name a benchmark. The library is never rolled back, and a component therefore survives even when the harness that first used it is reverted.

Composer. The composer writes the candidate harness from the library: it selects components, writes a free-text frame that opens the prompt, and sets the five knobs. It sees the library's index and bodies, the journal, the selection and knobs of the best harness $h^\star$, the editor's picks, and a short

excerpt of failed trajectories. A fixed assembly then builds $h_{\text{cand}} = (p, \Phi, \kappa)$: p joins the frame, the selected prompt fragments and knowledge, and the runner's reply-format block; Φ holds the selected processors; and κ takes the composer's knob values. The composer builds each candidate afresh from the library instead of editing the previous harness.

3.4 THE PASS@2 GATE

The gate keeps the best harness so far, $h^\star$, and its val score $b^\star$, initially h_0 and its own score. It scores a candidate by $s_t = \text{pass@2}(h_{\text{cand}}; \mathcal{T}_{\text{val}})$, the metric we report, estimated from M rollouts per val task, and picks the harness of the next round:

$$h_{t+1} = \begin{cases} h_{\text{cand}}, \text{ and } (h^\star, b^\star) \leftarrow (h_{\text{cand}}, s_t) & \text{if } s_t > b^\star \quad \text{(accept)}, \\ h^\star & \text{if } s_t < b^\star - \delta \quad \text{(revert)}, \\ h_{\text{cand}} & \text{otherwise} \quad \text{(hold)}. \end{cases} \tag{1}$$

A hold keeps searching from the candidate without making it the best harness. The margin δ is the largest of three quantities (Appendix B): (i) 7 pp, or half of $b^\star$ if that is smaller, (ii) a noise band estimated from how far the workers' scores spread within the rounds so far, and (iii) the spread of the best harness's repeated measurements. A candidate whose replies fail to parse more than 5% of the time is never accepted. After R rounds, LiteEvo returns $h^\star$, and a result's best round t means that the returned harness is h_t, first run in round t. In the loop itself, a candidate's val rollouts run at the start of the next round together with its train rollouts, so the meta-agents have already composed from them when the gate decides; a revert discards that composition, while the library keeps its edits.

3.5 WHY LITEEVO IS CHEAP

Three design choices keep the API cost of LiteEvo low. First, every meta-agent call is a single tool-free request on an input that the loop assembles, so that a round makes about five calls, one per meta-agent plus the editor's processor review, which Figure 2 and Algorithm 1 omit for brevity. An agentic meta-agent such as HarnessX's instead works through dozens of tool-using turns per round and re-sends its growing context at each of them (Chen et al., 2026). Second, only the analyst reads full trajectories, and only a fixed-size sample of them, on the cheaper of the two models, while the other meta-agents read far shorter inputs, such as its structured output, the journal and the library. Third, the library persists across rounds, so that a round adds or revises a few components instead of re-deriving the harness from the trajectories. Section 4 measures the resulting cost.

4 EVALUATION

4.1 EXPERIMENTAL SETUP

Models and Benchmarks. We evaluate on ALFWorld (Shridhar et al., 2021), WebShop (Yao et al., 2022), AppWorld (Trivedi et al., 2024), GAIA (Mialon et al., 2024) and SWE-bench Verified (Jimenez et al., 2024), which ask for household tasks in a text world, product searches, everyday tasks through the APIs of nine apps, web research with code, and repository-level bug fixes. The frozen agent is Qwen3.5-9B in thinking mode, served locally, and the analyst runs on Claude Sonnet 5 and the other three meta-agents on Claude Opus 4.6.

Implementation Details. Each run lasts $R = 8$ rounds, round 0 with the neutral harness and seven evolution rounds, with $M = 4$ workers ($M = 2$ on SWE-bench Verified). Unless a caption says otherwise, we grade a returned harness in three test runs of two attempts per task and report the mean pass@1 and pass@2. The API cost of a run counts the meta-agents' tokens in the calls that produce h_1 to h_7, at Opus 4.6's list price and Sonnet 5's introductory price. We leave out the cost of running the agent, which runs on local GPUs, for all methods: every harness-evolution method pays it, and comparing it fairly would need one common setup for running the agent, which was not yet fully reproducible at the time of submission.

Table 1: Test pass@2 (%) of Qwen3.5-9B, averaged over three test runs. *Bare* gives the agent only the task text. *Initial* is the starting harness, a benchmark-specific seed for HarnessX (+Seed) and the neutral harness for LiteEvo (+Neutral), and *Evolved* is the returned harness h_t, first run in round t (*Best Round*). Δ is Evolved minus Initial in pp, computed before rounding.

Benchmark	Mode	Method	Initialization	pass@2 (%)		Δ	Best Round	API cost ($)
				Initial	Evolved			
ALFWorld	Bare	–	Task only	37.1		–	–	–
	Transductive	HarnessX	+Seed	64.2	92.3	+28.1	6	118.47
		LiteEvo	+Neutral	60.9	96.3	+35.3	5	8.24
	Inductive	LiteEvo	+Neutral	60.9	94.8	+33.8	4	9.17
WebShop	Bare	–	Task only	24.0		–	–	–
	Transductive	HarnessX	+Seed	45.3	60.7	+15.3	6	175.11
		LiteEvo	+Neutral	10.0	62.7	+52.7	5	9.00
	Inductive	LiteEvo	+Neutral	10.0	70.0	+60.0	3	9.89
AppWorld	Bare	–	Task only	0.0		–	–	–
	Transductive	HarnessX	+Seed	48.6	62.3	+13.7	6	190.32
		LiteEvo	+Neutral	0.0	67.7	+67.7	6	12.51
	Inductive	LiteEvo	+Neutral	0.0	65.5	+65.5	7	14.56
GAIA	Bare	–	Task only	0.0		–	–	–
	Transductive	HarnessX	+Seed	56.0	64.4	+8.4	2	181.89
		LiteEvo	+Neutral	48.5	65.2	+16.7	7	10.59
SWE-bench Verified	Bare	–	Task only	16.4		–	–	–
	Transductive	HarnessX	+Seed	53.9	57.0	+3.0	1	172.22
		LiteEvo	+Neutral	52.7	63.2	+10.5	6	24.30
	Inductive	LiteEvo	+Neutral	52.7	66.7	+13.9	4	28.50

Baselines. We compare with the bare agent and with HarnessX (Chen et al., 2026), which we reproduce from its released code with the same agent and tasks, using Opus 4.6 as its meta-agent for eight rounds. As in its paper, HarnessX evolves on the tasks it is graded on, and we report its best round; we wrote its seeds for ALFWorld, WebShop and AppWorld, which the release lacks. On GAIA, whose agent searches the live web, LiteEvo's web tools block known copies of the benchmark's answers. For details of the setup not mentioned here, see Appendix C.

4.2 Main results

Main Comparison. Table 1 compares LiteEvo with HarnessX, and Figure 1a plots each run's pass@2 against its API cost. In the transductive regime, where both methods evolve on the tasks they are graded on, LiteEvo reaches a mean pass@2 of 71.0 over the five benchmarks, compared with 67.3 for HarnessX. It spends \$8.24 to \$24.30 per run where HarnessX spends \$118.47 to \$190.32, a 13.0× lower mean API cost. It does so from a weaker start on all five benchmarks: the neutral harness solves no AppWorld task, whereas the handcrafted seed reaches 48.6, and it scores 10.0 on WebShop, where the seed reaches 45.3. Evolution closes both gaps within eight rounds, adding 67.7 and 52.7 pp of pass@2. The neutral harness is not always the strongest trivial start: the bare agent scores 24.0 on WebShop, and a one-line format hint reaches 61.2 on SWE-bench Verified (Table 2). Measured from these stronger starts, LiteEvo's transductive gains are 38.7 and 2.0 pp.

Cost Breakdown. As Section 3.5 anticipates, HarnessX's meta-agent, an agentic session with file tools, makes 65 to 130 Opus calls and consumes 3.3 to 5.6 million input tokens per round in our

Table 2: The returned harness (*Full*) split into its system prompt and processors (Qwen3.5-9B, transductive): *+Prompt* and *+Processor* add one part to *Bare + format*, the bare agent with a minimal action-format hint. P@1 and P@2 are pass@1 and pass@2 in %, averaged over three test runs (one for Bare on WebShop, GAIA and SWE-bench Verified).

	ALFWorld		WebShop		AppWorld		GAIA		SWE-bench Verified	
Harness	P@1	P@2	P@1	P@2	P@1	P@2	P@1	P@2	P@1	P@2
Bare	25.1	37.1	14.5	24.0	0.0	0.0	0.0	0.0	9.1	16.4
Bare + format	46.8	56.7	9.3	13.8	0.0	0.0	29.1	44.3	50.9	61.2
+Processor	49.3	57.7	41.3	53.0	0.0	0.0	44.6	59.2	51.0	63.1
+Prompt	**92.8**	95.5	51.0	59.0	46.1	63.1	41.9	49.5	49.4	58.2
Full	92.6	**96.3**	**54.0**	**62.7**	**51.9**	**67.7**	**53.6**	**65.2**	**54.2**	**63.2**

Table 3: pass@2 (%) of Claude Code with Sonnet 4.6 at low effort and its own tools turned off, given only the task (*Initial*) and with the harness that LiteEvo evolves for it; each evolved cell is a single measurement. The GAIA run blocks known copies of the benchmark's answers. In Figure 1b, Sonnet 4.6 on AppWorld is also told how to finish a task.

Benchmark	Mode	Method	pass@2 (%)		Δ	Best Round	API cost ($)
			Initial	Evolved			
ALFWorld	Transductive	LiteEvo	88.8	96.3	+7.5	2	8.21
	Inductive	LiteEvo	88.8	95.5	+6.7	7	12.92
WebShop	Transductive	LiteEvo	77.0	81.0	+4.0	6	9.84
	Inductive	LiteEvo	77.0	81.0	+4.0	5	12.57
AppWorld	Transductive	LiteEvo	0.0	66.1	+66.1	7	11.09
	Inductive	LiteEvo	0.0	71.4	+71.4	4	12.58
GAIA	Transductive	LiteEvo	67.7	74.3	+6.7	4	9.08
SWE-bench Verified	Transductive	LiteEvo	77.0	80.0	+3.0	2	11.63
	Inductive	LiteEvo	77.0	78.2	+1.2	6	11.80

reproduction, whereas LiteEvo makes about five calls and consumes 0.12 to 0.54 million, mostly read by the analyst on the cheaper Sonnet 5.

Inductive Transfer. The inductive runs of Table 1 evolve on train tasks, choose between harnesses on val tasks, and are graded on test tasks that the loop never sees. On ALFWorld and AppWorld they stay within 2.2 pp of the transductive runs, and on WebShop and SWE-bench Verified they score higher, with 70.0 against 62.7 and 66.7 against 63.2; the inductive harnesses also score above HarnessX's transductive ones, although HarnessX, which reports no held-out evaluation, selected its harnesses on the test tasks themselves. GAIA has no inductive row, as its 103 public questions are too few to split three ways. On AppWorld tasks that need apps unseen in training, our inductive harness reaches 48.6 where the neutral harness solves none, against 49.2 for our transductive one and 50.5 for HarnessX's (Appendix D.2); a library evolved on other benchmarks does not yet help an unseen one.

Other Backbones. Table 3 applies LiteEvo to Claude Code (Anthropic, 2026a) running Sonnet 4.6: the composed system prompt is appended to Claude Code's own, and the processors act on the actions and observations exchanged with the environment. LiteEvo raises pass@2 in all nine benchmark and regime cells at $8.21 to $12.92 per run: by 1.2 to 7.5 pp on the four benchmarks where the bare agent is already strong, and on AppWorld from 0.0 to 66.1 in the transductive regime and to 71.4 in the inductive one. On the smaller Qwen3.5-4B, LiteEvo adds 6.7 to 49.8 pp to the neutral start (Appendix D.1).

Table 4: Library operations and composer selections per round in the five transductive LiteEvo runs (Qwen3.5-9B), summed over the runs; *Size* (components selectable after the round) and *Picked* (selected for the next harness) are averaged. *Removed* counts components deleted by merges and drops, and *Pick rate* is the pooled ratio of Picked to Size.

Round		0	1	2	3	4	5	6	7	Total
Librarian	Add	19	11	12	10	6	11	8	8	85
	Update	0	4	4	2	5	0	3	7	25
	Merge	0	1	3	2	0	1	0	0	7
	Drop	0	0	1	0	0	0	0	1	2
Library	Removed	0	2	4	2	0	0	0	1	9
	Size	3.8	5.8	8.0	10.0	11.2	13.4	15.0	16.4	–
Composer	Picked	3.8	5.8	7.8	9.6	9.8	11.4	11.2	12.4	–
	Pick rate (%)	100.0	100.0	97.5	96.0	87.5	85.1	74.7	75.6	–

Table 5: Neutral or handcrafted start (Qwen3.5-9B, transductive, one run each; HarnessX's own seed on GAIA): test P@1 and P@2 (%), mean of three test runs.

	AppWorld		GAIA	
Initial harness	P@1	P@2	P@1	P@2
Neutral (default)	**51.9**	**67.7**	**53.6**	**65.2**
HarnessX seed	49.1	64.9	48.4	59.9

4.3 Ablation studies

This section examines how the returned harness splits between its prompt and its processors, how the library grows, and whether a handcrafted start helps; for more ablations, see Appendix D.

Prompt versus Processors. Table 2 splits a returned harness into its system prompt and its processors. The full harness gives the best pass@2 on all five benchmarks. The prompt carries most of the gain on ALFWorld and AppWorld, reaching 95.5 and 63.1 alone compared with 96.3 and 67.7 for the full harness; on GAIA and SWE-bench Verified the processors do, reaching 59.2 and 63.1 alone compared with 65.2 and 63.2. Since the two parts evolved together, the split does not isolate their contributions.

Library Growth. In the five transductive runs (Table 4), the librarian mostly adds, with 85 additions against 25 updates, 7 merges and 2 drops, so the library grows from 3.8 components on average after the first round to 16.4 after the last. The composer uses all components in the first two rounds and then becomes selective, picking 75.6% of them in the last round. Forcing an explicit memory across episodes into every harness lowered pass@2 (Appendix D.8).

Neutral versus Handcrafted Start. Table 5 starts LiteEvo from a handcrafted HarnessX-style seed instead of the neutral harness. The seeded runs end lower on both benchmarks, with 64.9 against 67.7 on AppWorld and 59.9 against 65.2 on GAIA, so the neutral start matches or exceeds a handcrafted one and is our default; with one run per arm, this does not show that a seed hurts.

5 Conclusion

We introduced LiteEvo, which evolves a harness from the same neutral start on every benchmark with four tool-free meta-agents, a versioned component library and a pass@2 gate. With a frozen Qwen3.5-9B, it reaches comparable or higher pass@2 than our reproduction of HarnessX (71.0 against 67.3 on average) at 13.0× lower mean API cost, from one run per benchmark. Its inductive harnesses keep their gains on unseen test tasks, and a handcrafted seed did not improve on the neutral start (Appendix E discusses limitations and future work).

AI USE STATEMENT

We used generative AI tools heavily to polish the manuscript and to correct non-native English expressions. For the appendix, AI agents also drafted text, which we read and verified. We developed the meta-agents' instructions with the help of AI agents, but we verified sensitive content by hand, in particular that no instruction leaks information about a benchmark. AI agents also ran and monitored our experiments, while we verified the reported results and the evaluation code ourselves. We wrote the instructions for the supplementary code with an AI agent and ran the code on our own server to confirm that it works. Apart from these uses, LiteEvo itself runs on LLMs: the agent and all four meta-agents are language models, as Sections 2 and 3 describe. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work evaluates agents on public benchmarks inside sandboxed environments and involves no human subjects. Agents that search the web can reach published copies of benchmark answers. Appendix C.4 describes how we block known copies on GAIA and what exposure remains.

REPRODUCIBILITY STATEMENT

Section 3 and Appendix B specify the loop, the meta-agents, the gate and every setting, and Appendix B.1 gives the neutral harness verbatim. Appendix C gives the task sets, the grading protocol, the cost accounting and the HarnessX reproduction. As supplementary material, we attach a minimal set of our code that runs the LiteEvo experiments. We will release the full code on GitHub upon acceptance.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/0e9e708b6f48e14fd0ac29e167413f76-Paper-Conference.pdf.

Anthropic. Claude Code overview. Claude Code documentation, 2026a. URL https://code.claude.com/docs/en/overview. Accessed 2026-09-23.

Anthropic. How Claude Code works. Claude Code documentation, 2026b. URL https://code.claude.com/docs/en/how-claude-code-works. Accessed 2026-09-23.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. Training-free group relative policy optimization, 2025. URL https://arxiv.org/abs/2510.08191.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. FireAct: Toward language agent fine-tuning, 2023. URL https://arxiv.org/abs/2310.05915.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec

Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. AutoManual: Constructing instruction manuals by LLM agents via interactive environmental learning. In *Advances in Neural Information Processing Systems 37*, 2024a. URL https://openreview.net/forum?id=Pwl9n4zlf5.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. HarnessX: A composable, adaptive, and evolvable agent harness foundry, 2026. URL https://arxiv.org/abs/2606.14249.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-FLAN: Designing data and methods of effective agent tuning for large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 9354–9366, Bangkok, Thailand, August 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.557. URL https://aclanthology.org/2024.findings-acl.557/.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Agentic reinforced policy optimization. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 16981–17017, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/1c58e53bdf1fb045440256fd567531ae-Paper-Conference.pdf.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. In *Advances in Neural Information Processing Systems*, volume 38, pp. 46375–46408. Curran Associates, Inc., 2025. doi: 10.52202/085713-1544. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/420c9f777c0b4f78d515e53cf74d58b2-Abstract-Conference.html.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A generalist multi-agent system for solving complex tasks, 2024. URL https://arxiv.org/abs/2411.04468.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Lisheng Huang, Chen Yang, Hao Zhou, Huatong Song, Zongchao Chen, Ran Le, Yang Song, Wayne Xin Zhao, and Tao Zhang. Evo-Bench: Can language models improve agent harness?, 2026. URL https://arxiv.org/abs/2608.09096.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od. Spotlight.

Yoonho Lee, Roshen S. Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling*, 2026. URL https://arxiv.org/abs/2603.28052.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026a. URL https://arxiv.org/abs/2604.25850.

Minhua Lin, Juncheng Wu, Zijun Wang, Zhan Shi, Yisi Sang, Bing He, Zewen Liu, Tianxin Wei, Zongyu Wu, Zhiwei Zhang, Dakuo Wang, Xiang Zhang, Benoit Dumoulin, Cihang Xie, Yuyin Zhou, Suhang Wang, and Hanqing Lu. Harness updating is not harness benefit: Disentangling evolution capabilities in self-evolving LLM agents, 2026b. URL https://arxiv.org/abs/2605.30621.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: improving LLM agents by automatically synthesizing a code harness, 2026. URL https://arxiv.org/abs/2603.03329. Presented at the ICLR 2026 Workshop on AI with Recursive Self-Improvement.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling RL. Together AI Blog, July 2025. URL https://www.together.ai/blog/deepswe.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/444a3737adaee10d86ad2ef5f74468e6-Paper-Conference.pdf.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for General AI Assistants. In *International Conference on Learning Representations (ICLR)*, pp. 9025–9049, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. Learning to orchestrate agents in natural language with the conductor. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 135686–135724, 2026. URL https://openreview.net/forum?id=U23A2BUKYt.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent, 2025. URL https://arxiv.org/abs/2504.15228. Presented at the ICLR 2025 Workshop on Scaling Self-Improving Foundation Models.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations (ICLR)*, volume 2025, pp. 3841–3865, 2025. URL https://openreview.net/forum?id=mPdmDYIQ7f. arXiv:2410.06153.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36*, 2023. URL https://openreview.net/forum?id=vAElhFcKW6.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=VpNENmFBEEg.

Yifan Song, Weimin Xiong, Xiutian Zhao, Dawei Zhu, Wenhao Wu, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. AgentBank: Towards generalized LLM agents via fine-tuning on 50000+ interaction trajectories. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 2124–2141, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.116. URL https://aclanthology.org/2024.findings-emnlp.116/.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, March 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents, 2026. URL https://arxiv.org/abs/2607.12227.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025b. URL https://arxiv.org/abs/2409.07429.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 78500–78525. Curran Associates, Inc., 2025a. doi: 10.52202/085713-2629. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/7107d4d2e837bde2171c6b71b5bde954-Paper-Conference.pdf.

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, Hyokun Yun, and Lihong Li. WebAgent-R1: Training web agents via end-to-end multi-turn reinforcement learning. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 7909–7928, Suzhou, China, November 2025b. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.401. URL https://aclanthology.org/2025.emnlp-main.401/.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Zhiheng Xi, Jixuan Huang, Chenyang Liao, Baodai Huang, Jiaqi Liu, Honglin Guo, Yajie Yang, Rui Zheng, Junjie Ye, Jiazheng Zhang, Wenxiang Chen, Wei He, Yiwen Ding, Guanyu Li, Zehui Chen, Zhengyin Du, Xuesong Yao, Yufei Xu, Jiecao Chen, Tao Gui, Zuxuan Wu, Qi Zhang, Xuanjing Huang, and Yu-Gang Jiang. AgentGym-RL: An open-source framework to train LLM agents for long-horizon decision making via multi-turn RL. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 12685–12727, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/1571ce1ff735be4551db50887043726e-Paper-Conference.pdf.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=Bb4VGOWELI.

Chenyang Yang, Xinran Zhao, Tongshuang Wu, and Christian Kästner. Better harnesses, smaller models: Building 90% cheaper agents via automated harness adaptation, 2026. URL https://arxiv.org/abs/2607.08938.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024b. doi: 10.52202/079017-1601. URL

https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 20744–20757. Curran Associates, Inc., 2022. doi: 10.52202/068431-1508. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/82ad13ec01f9fe44c01cb91814fd7b8c-Paper-Conference.pdf.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. AgentTuning: Enabling generalized agent abilities for LLMs. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.181. URL https://aclanthology.org/2024.findings-acl.181/.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025a. URL https://openreview.net/forum?id=imcyVlzpXh. Oral.

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, Yifan Zhou, Yang Chen, Chen Zhang, Yutao Fan, Zihu Wang, Songtao Huang, Francisco Piedrahita-Velez, Yue Liao, Hongru Wang, Mengyue Yang, Heng Ji, Jun Wang, Shuicheng Yan, Philip Torr, and Lei Bai. The landscape of agentic reinforcement learning for LLMs: A survey. *Transactions on Machine Learning Research*, 2026a. ISSN 2835-8856. URL https://openreview.net/forum?id=RY19y2RI1O.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026b. URL https://arxiv.org/abs/2606.09498.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026c. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/aa5f5e6eb6f613ec412f1d948dfa21a5-Paper-Conference.pdf.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations (ICLR)*, volume 2025, pp. 34040–34077, 2025b. URL https://openreview.net/forum?id=z5uVAKwmjf. Oral. arXiv:2410.10762.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. HarnessCompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses, 2026d. URL https://arxiv.org/abs/2608.01918.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026e. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/8a94ff6f922d995d7d3f4ebf4143e442-Paper-Conference.pdf.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

A Related work

HarnessX (Chen et al., 2026) is the closest work: its Opus 4.6 meta-agent evolves harnesses on four of our five benchmarks, starting each from a handcrafted harness, and the paper reports peak-round pass@2 on the tasks it evolves on. Other harness optimizers likewise build knowledge of the benchmark into the start or into the optimizer's instructions, through per-domain seeds and skills (Lee et al., 2026), the benchmark's official agent (Zhang et al., 2026e), the game's description (Lou et al., 2026), an exploration agent aimed at the benchmark (Lin et al., 2026a), or software-repair hints (Zhang et al., 2026d). Methods that evolve whole agents report high costs for a whole run: about $300 to $500 for ADAS, which scores held-out tasks, and about $7,000 and $22,000 for the self-improving coding agents SICA and DGM (Hu et al., 2025; Robeyns et al., 2025; Zhang et al., 2026c). Cheaper methods that score held-out tasks, such as GEPA and Better Harnesses, are evaluated on none of our benchmarks (Agrawal et al., 2026; Yang et al., 2026). Once search and test tasks are separated, harness evolution in the style of AHE gains only 0.6 pp on Terminal-Bench 2.1 (Wang et al., 2026). LiteEvo targets all of these limits at once: it starts all five benchmarks from one neutral harness, never names the benchmark to its meta-agents, costs under $30 per run in API calls, and reports held-out as well as transductive scores.

Table 6 compares LiteEvo with the closest methods on four axes: the starting harness, what the optimizer is told about the benchmark, how far from the evolution tasks the result is measured, and the optimizer cost that each paper reports.

Table 6: How LiteEvo and the closest methods start, what their optimizer is told about the benchmark, how far from the evolution tasks they are evaluated, and what optimizer cost they report. Levels: L0, evolved and graded on the same tasks; L1, unseen tasks of the same benchmark; L1.5, tasks that need unseen apps; L2, a sibling benchmark; L3, a foreign benchmark. Evo-Bench is a benchmark of evolver models rather than a method.

Method	Starting harness	What the optimizer is told about the benchmark	Evaluation	Cost reported
HarnessX	handcrafted for each benchmark	its benchmark-specific seeds	L0	token budget
Meta-Harness	per-domain seeds	a skill written by hand for each domain	L0 on its agentic benchmark; held-out and new datasets for classification	iterations, hours
AutoHarness	one template	the game's name and description	new random instances of each game	iterations
AHE	a single shell tool	an explore agent aimed at the benchmark	L0, plus one other benchmark without re-evolution	hours
HarnessCompass	a single shell tool	software-repair hints in the meta prompt	L1 on one benchmark, plus a second model	iterations
ACE	the benchmark's official agent	a reflector named after the benchmark	L1 offline; online on the test stream	latency, rollouts
Evo-Bench	one minimal seed	the domains, in the evolver prompt	L1	USD per run
LiteEvo	one neutral harness for all five benchmarks	no benchmark identity; a filter blocks benchmark names	L0 on five benchmarks, L1 on four, L1.5 on AppWorld; L2 and L3 tried; no transfer shown	tokens and USD, also for HarnessX

Training the Agent's Weights. Reinforcement learning trains an agent's weights on its own multi-turn interaction (Feng et al., 2025; Dong et al., 2026; Xi et al., 2026; Wei et al., 2025b); Zhang et al. (2026a) survey the area. Such training adapts the model itself, but one run occupies a GPU cluster: SWE-RL trains on 512 GPUs for about 32 hours (Wei et al., 2025a), and DeepSWE on 64 GPUs for six days (Luo et al., 2025). Supervised fine-tuning on agent trajectories (Zeng et al., 2024; Chen et al., 2023; Song et al., 2024; Chen et al., 2024b) is cheaper per run but needs trajectories from a

stronger model, and it applies only to open weights. Since LiteEvo changes no weight, it applies to closed agents such as Claude Code, and its evolved harnesses could be combined with either kind of training.

Multi-Agent Orchestration. Multi-agent systems split a task among specialist agents under an orchestrator or a fixed workflow (Wu et al., 2024; Hong et al., 2024; Fourney et al., 2024; Nielsen et al., 2026). LiteEvo uses several LLM roles as well, but only offline, to write the harness of a single agent that then runs alone at test time.

How Much the Harness Matters. With the model fixed, scores vary widely with the harness: SWE-agent's interface solves 10.7 pp more SWE-bench instances than a plain shell with the same model (Yang et al., 2024b). A coding agent that the DGM evolved lifts Claude 3.7 Sonnet from 19.0% to 59.5% on 200 SWE-bench tasks (Zhang et al., 2026c), and one model scores from 13.9% to 37.6% on Terminal-Bench 2 depending on its harness (Lee et al., 2026). Coding agents such as Claude Code are themselves harnesses around a model (Anthropic, 2026b; Wang et al., 2025a). The choice of model can still matter more than the harness (Merrill et al., 2026), which is one reason we also evolve harnesses for a frontier agent (Table 3).

Prompt, Context and Memory Optimization. Prompt optimizers tune the instructions and demonstrations of a fixed program from feedback (Khattab et al., 2024; Opsahl-Ong et al., 2024; Yang et al., 2024a; Yuksekgonul et al., 2024). GEPA reflects on traces to evolve prompts and beats reinforcement learning with up to 35$\times$ fewer rollouts (Agrawal et al., 2026), and ACE grows a playbook of strategies in an agent's context (Zhang et al., 2026e). A second line keeps experience in memory: Reflexion retries a task with verbal self-reflection (Shinn et al., 2023), ExpeL and AutoManual distill insights or manuals from training tasks (Zhao et al., 2024; Chen et al., 2024a), and Agent Workflow Memory, Dynamic Cheatsheet and ReasoningBank accumulate workflows, notes or strategies as a test stream unfolds (Wang et al., 2025b; Suzgun et al., 2026; Ouyang et al., 2026). LiteEvo's library stores the same kinds of knowledge in typed components, next to processors and runner settings, and a composer decides which of them the agent sees. Unlike test-stream memories, LiteEvo fits its library offline and freezes its harness at test time.

Automated Design of Agents and Harnesses. Earlier work searches over whole agent designs: ADAS programs new agents in code (Hu et al., 2025), AFlow searches over code-represented workflows (Zhang et al., 2025b), AgentSquare searches a modular design space (Shang et al., 2025), and MaAS samples multi-agent systems from a learned supernet (Zhang et al., 2025a). DGM and SICA let a coding agent rewrite its own code, at about $22,000 and $7,000 per run (Zhang et al., 2026c; Robeyns et al., 2025). Harness evolution instead keeps the model fixed and changes the layer around it. HarnessX, Meta-Harness, AutoHarness, AHE, HarnessCompass and Self-Harness evolve prompts, tools and code around a task agent (Chen et al., 2026; Lee et al., 2026; Lou et al., 2026; Lin et al., 2026a; Zhang et al., 2026d;b), several of them with agentic meta-agents that use file or web tools. Better Harnesses fits harnesses for small models at $20 per run, rollouts included (Yang et al., 2026). Studies of the setting find that gains shrink when search and test are separated (Wang et al., 2026) and that a small evolver can match a frontier one on in-place skill edits (Lin et al., 2026b), while Evo-Bench evaluates evolvers on held-out tasks (Huang et al., 2026). LiteEvo's meta-agents, by contrast, are tool-free, read a failure-oriented sample of trajectories instead of full traces, and never see the benchmark's name.

B METHOD DETAILS

This appendix gives what Section 3 and Algorithm 1 leave out.

B.1 THE NEUTRAL HARNESS

Every run starts from the system prompt in Figure 3, which is 788 characters long and, with the runner's reply-format block, labels and reminder (Appendix B.9), is the only text we write for the agent.

Neutral Harness (h_0)

You are an autonomous agent operating in a text environment over a sequence of turns.

Each turn you receive an observation describing the current state. You reply with exactly one action. The environment then returns the next observation.

Reply format -- put your action inside action tags:
<action>the action, written exactly as the environment expects</action>
The environment alone defines which actions exist and how they must be written. Read each observation carefully: it is your only source of truth about what is available and about what your last action did. Anything you write outside the action tags is not sent to the environment.

You have a limited number of turns. The episode ends when the environment reports that it is finished, or when the turn budget is exhausted.

Figure 3: The neutral harness h_0, the system prompt with which every run starts. Line breaks inside paragraphs are added for display.

On AppWorld, GAIA and SWE-bench Verified, whose actions are code, the text changes in three places and grows to 977 characters: "one action" becomes "one action block", the example spans two lines, and one sentence says that the text inside the tags may span several lines and runs as one unit. The neutral harness has no processors, and its knobs take the defaults of Table 7.

B.2 THE RUNNER

The runner is a multi-turn chat loop, the same for all benchmarks and harnesses. Its system message is p, and each user message holds the latest observation, preceded by the number of turns left and followed by the list of available actions when the knobs ask for them. The model thinks before it replies, but its reasoning is not kept in the history. The system message and the first observation are never dropped; past a prompt budget of 32,768 tokens, the oldest reply–observation pairs are dropped, while the four most recent are always kept. The runner executes the text inside the first `<action>` tag of a reply, or the rest of the reply if the tag is never closed. On ALFWorld and WebShop it can fall back to the reply's last line; no other text reaches the environment. A reply without an action gets a fixed one-sentence reminder, and on the benchmarks whose actions are code, the first three such replies of an episode do not use up a turn.

B.3 PROCESSORS AND KNOBS

A processor is a Python function `advise(ctx)` that returns a dictionary. Its context exposes the turn index and budget, the task, the last observation and action, the available actions, the steps taken so far, and a scratchpad shared by the processors of one episode, but never the model's reasoning or the reply being processed. Each hook accepts one key:

- `before_model` may return `inject`, a message that only the next model call sees;
- `after_model` may return `rewrite_action`, which replaces the extracted action;
- `after_step` may return `rewrite_obs`, which replaces the observation the model reads. The trajectory record that the analyst reads keeps the environment's original text.

Processors on one hook run in a fixed order, each with an allow-listed set of built-ins and the modules `re` and `json` but no imports, and an exception inside a processor is caught and counts as an empty return. Table 7 lists the five knobs, and every composition sets all five.

Table 7: The five runner knobs, their values in the neutral harness, and their effect.

Knob	Neutral	Effect
Protocol	tagged	tagged text replies, or one generic `act` tool call
Available actions	shown	append the environment's list of legal actions to each observation
Observation cap	off	truncate long observations (command actions only)
Action fallback	on	without a tag, take the reply's last line as the action (command actions only)
Remaining turns	hidden	prefix each observation with the number of turns left

The remaining-turns knob gives the agent information that the neutral harness withholds, and since the harnesses LiteEvo returned turn it on, part of any gain over the neutral harness can come from information that the composer granted rather than from what the loop learned.

B.4 COMPONENTS AND ASSEMBLY

A component has an identifier, a kind, a title, a one-line summary, a body, an applicability claim, a rationale and usage statistics; a processor also has a hook and an order. Only the body can reach a harness, and Table 8 lists the five kinds. The librarian and the composer read the library through an index with one line per component, which gives the component's kind, summary and applicability claim, the rounds in which it was deployed, the mean train score of those rounds, and how often it fired. Flags mark components that were deployed but never fired and components never tried, and the bodies follow the index, each cut at 1,600 characters. The applicability claim uses seven fixed axes, such as the bottleneck, the horizon and the density of the reward; the meta-agents read it, but no check uses it.

Table 8: Component kinds, where the assembly places each, and when it acts.

Kind	Body	Placed	Acts
Prompt fragment	an instruction or a worked example	system prompt, after the frame, in its order	every model call
Knowledge	facts observed in trajectories	system prompt, after the fragments	every model call
Processor	a Python function `advise(ctx)`	one runner hook	each turn it returns a key
Knob	runner settings	read by the composer only	never directly
Strategy	advice on composing	the composer's and librarian's input	never reaches the agent

Given the library and a composition, the fixed assembly builds the harness in four steps. It first resolves and admits the selection: each selected identifier is looked up, an identifier merged away is followed to the component that absorbed it, every body passes the identity filter again, processors must compile and pass a smoke test, and knob bodies must parse. It then writes the system prompt, with the frame first, the selected prompt fragments in their order, the knowledge bodies, and the runner's reply-format block of 198 characters (381 where actions are code), all joined by blank lines; titles, summaries and rationales are never placed. Finally, it compiles each selected processor onto its hook and applies the composition's five knob values last, so a knob component informs the composer but never sets a value itself.

B.5 THE META-AGENTS

Each meta-agent receives its fixed instructions as the system message and one user message that the loop assembles, and it returns JSON that follows a fixed schema, with no tools or file access. The analyst is Claude Sonnet 5 and the editor, librarian and composer are Claude Opus 4.6, all called with high effort, adaptive thinking and no sampling parameters. Output budgets are 32,000 tokens for the analyst and the librarian and 16,000 for the others, and a truncated reply is retried with twice the budget. Table 9 lists what each meta-agent reads and returns, and Appendix B.7 gives their instructions in full.

Table 9: What each meta-agent reads and returns in one round, in call order. Instructions are omitted.

Meta-agent (model)	Reads	Returns
Analyst (Sonnet 5)	the running harness's settings and full system prompt; outcome statistics over the round; 13 task groups with all M trajectories each	a task signature; failure modes with their share, evidence and mechanism; candidate components, with a cited step for each fact
Editor (Opus 4.6)	the analyst's output with its citation checks; a journal of earlier rounds (scores, decisions, library edits, rejected processors)	the selected candidates, rewritten or merged; dropped candidates with reasons; then the processor check
Librarian (Opus 4.6)	the library index with usage statistics and all bodies; the editor's selections and drops	operations: add, update or merge a complete component, or drop one, each with a reason
Composer (Opus 4.6)	the task signature; this round's score and the best harness's; the strategies; the library; the journal; the best harness's selection, knobs and size (not its frame); the editor's picks; a 4,000-character excerpt of failed trajectories	a selection of components, a frame and five knob values

Strategies. The three strategies we write are generic rules for composing. The first says not to write a reply-format paragraph, since the runner appends its own, and the second tells the composer to write the frame and let components carry the content. The third says that dropping a component that was live in the best round is a decision the composer must name in its rationale.

B.6 SAMPLING, CHECKS AND FILTERS

Failure-Oriented Group Sampling. Algorithm 2 gives the rule. Only tasks that all M workers attempted are eligible, and the sample holds the ten mixed groups with the highest mean partial credit and the three all-fail groups with the lowest. If fewer than ten mixed groups exist, all-fail groups with the highest partial credit fill the gap.

Algorithm 2 GROUPSAMPLE, the analyst's failure-oriented group sampling. $n_{\text{mix}} = 10$ and $n_{\text{fail}} = 3$, or 5 and 2 on SWE-bench Verified.

Require: groups $\mathcal{G}$: for each train task τ that all M workers attempted, its M rollouts

for each τ: $w_\tau \leftarrow$ number of workers that solve τ; $\bar{f}_\tau \leftarrow$ mean partial credit of its M rollouts
$\mathcal{X} \leftarrow \{\tau : 0 < w_\tau < M\}$ sorted by $\bar{f}_\tau$, highest first ▷ mixed
$\mathcal{Z} \leftarrow \{\tau : w_\tau = 0\}$ ▷ all-fail
$S_{\text{mix}} \leftarrow$ the first n_{mix} tasks of $\mathcal{X}$
$S_{\text{fail}} \leftarrow$ the n_{fail} tasks of $\mathcal{Z}$ with the lowest $\bar{f}_\tau$
if $|S_{\text{mix}}| < n_{\text{mix}}$ **then** ▷ too few mixed groups
    add to S_{fail} the $n_{\text{mix}} - |S_{\text{mix}}|$ remaining tasks of $\mathcal{Z}$ with the highest $\bar{f}_\tau$
end if
return the groups of $S_{\text{mix}} \cup S_{\text{fail}}$, each with its M rollouts and outcomes

Every group shows all M trajectories, so the analyst reads 52 trajectories when $M = 4$, each with its outcome, partial credit, number of turns and exit reason. Actions are cut to 400 characters and observations to 300, with longer allowances for the first turn and the five longest steps, and if the sample still exceeds 1.6 million characters, the allowances shrink until it fits, without dropping a trajectory.

The Editor's Processor Check. Each processor the editor keeps is compiled, smoke-tested and replayed on the first 24 episodes of the round, and a processor that fails to compile, raises an error or never fires is dropped. The editor then reads each surviving processor's claim, its code, how often it fired and up to six of its outputs, dropping any processor whose outputs contradict its claim. Only processors face this check; fragments, knowledge and knobs reach the librarian without a replay.

The Identity Filter. One filter guards every meta-agent's input and output. Before any meta-agent reads a trajectory, task identifiers, benchmark and split fields, and URLs are removed, names that identify a benchmark in environment text are replaced by a placeholder, and the turn cap appears only as a label, short or long, that at least two benchmarks share. When the filter fires on an output, the response depends on the meta-agent: an analysis is retried once and then discarded, editor candidates are dropped, library writes are refused, and a frame is anonymized. Component bodies pass the filter again at assembly.

Citations for Facts. Each knowledge component that the analyst proposes cites the episode and turn where the agent observed its fact, unless no single step yields it, as for a regularity across episodes. The loop checks that the cited turn exists and that the action there matches the fact, and it drops the component otherwise, so that the analyst cannot store a fact recalled from pre-training as if the agent had seen it.

B.7 THE META-AGENTS' INSTRUCTIONS

Figures 4 to 12 reproduce the meta-agents' instructions verbatim; the composer's instructions call the best harness the incumbent. All four meta-agents share one part (Figures 4–7), which holds the rule against using recalled knowledge of a benchmark and documents the component kinds, the processor interface and the applicability axes. Each meta-agent's own instructions mark where this part is inserted. The loop adds one user message per call (Table 9).

Common Prompt Part

```
CRITICAL -- BLIND ANALYSIS RULE
You are looking at an UNKNOWN text environment. You may reason ONLY from the
evidence placed in front of you in this message.

You must NOT name, guess at, or rely on knowledge of any specific published
benchmark, dataset, simulator, or paper. If the transcripts look familiar and
you believe you recognise the environment, that recognition is EXACTLY what
must not influence your output -- suppress it and work from the transcripts.

THE LINE IS PROVENANCE, NOT SPECIFICITY.
 * A specific fact you OBSERVED in the transcripts is legitimate, however
   concrete. You are allowed to write down entity names, locations, and the
   statistics of where things were found, because the environment showed you.
 * The same fact RECALLED from prior knowledge of a published benchmark is a
   leak, because it did not come from the evidence and would not be available
   in a new environment.

Concretely:
 BANNED  : naming a benchmark; recalling its published action grammar, its
        object inventory, its scoring rule, or its known failure modes;
        writing any claim you could not point to a transcript line for.
 ALLOWED : concrete, observed specifics -- exact verb phrasings the environment
        accepted or rejected, names of entities that appeared, and indexes
        of where particular things were actually found.
 REQUIRED: the MECHANISM must be generic -- it must be a rule some agent could
        run in ANY environment of this abstract shape. Its CONTENT may be
        specific to this environment, because this environment is where you
        observed it.

So: "maintain an index of where each kind of target has been found and search
the highest-yield places first" is a generic mechanism. Filling that index with
what the transcripts actually showed -- including a line that amounts to "this
kind of item turned up in these particular places" -- is filling a generic
mechanism with observed content, and is FINE. What is not fine is writing the
index from memory of a benchmark you think you recognise, or naming it.

The test to apply, every time: COULD I HAVE WRITTEN THIS WITHOUT THE
TRANSCRIPTS? If yes, it is recalled -- delete it. If no, it is discovered --
keep it, and be as concrete as the evidence lets you be.


A harness component is one reusable piece of an agent harness. Its MECHANISM is
generic; its CONTENT may be whatever the transcripts showed. Kinds:

 prompt_fragment : one paragraph appended to the agent's system prompt
            VERBATIM, in `order` position, between the composed frame
            and the knowledge index. Like knowledge, it is never
            paraphrased -- write the exact words you want the agent to
            read. Keep it to a paragraph: four selections at the
            archived median (536 chars) add ~2.1k to a prompt whose
            median is 3.6k.
            A fragment MAY be a WORKED EXAMPLE: a rule, then a
            demonstration of it inside the runner's action tags. This is
            the strongest single thing a component can carry. Measured
            on 168 held-out tasks: deleting the seven worked examples
            from a prompt cost 8.9pp pass@2 (p=0.03), and the agent
            opened by reusing a line from one of them in 42.8% of
            episodes. Two rules, both from measurement. (1) Write the
            demonstration as MORE THAN ONE line where the environment
            executes a block -- a one-line example next to prose saying
            multiple lines are allowed produced multi-line actions on
            2.0% of turns, and fixing only the example took it to 25.5%.
```

Figure 4: The part of the instructions that all four meta-agents share (1 of 4).

Common Prompt Part (Cont.)

```
(2) Do NOT put a specific task's literals in it -- names,
          titles, addresses, ids. Replacing all of that with generic
          introspection changed the score by +0.0pp (22/22, p=1.00),
          so it buys nothing and it is the one part that could carry a
          fit-split answer into a held-out prompt. Fragments land
          immediately after the composed frame; `order` sets their
          position among themselves. In the one prompt where examples
          were measured to help, all seven sat in the first third of
          it -- but that measurement cannot say whether WHERE they sat
          or HOW MANY there were is what paid, so do not treat `order`
          as a lever on the strength of it.
processor     : Python source defining `advise(ctx) -> dict`, run at a hook.
protocol      : DEPRECATED -- DO NOT PROPOSE. It is refused at assembly and
          reaches nothing. A reply-parsing change is either a
          `processor` at `after_model` returning `rewrite_action`, or
          a `knobs.protocol` value. (19 were written; none was ever
          selected, and every body was an English request to change
          the runner rather than a component.)
strategy      : guidance for the COMPOSER on how to assemble a harness, not
          text for the agent. It is refused at assembly and can never
          reach the episode. Say how to build the prompt -- what to
          put first, what to leave to components, what to stop
          rewriting -- not what the agent should do.
knob          : a runner setting, applied as a BASE LAYER that the
          composition's own `knobs` may override. Both the field name
          and the name `describe()` prints are accepted. The runner
          honours exactly these, and a knob naming anything else is
          reported as unparseable and sets nothing:
            protocol          tagged | tool
            show_affordances    bool
            allow_action_fallback bool -- when the reply carries no
              action tag, salvage its last line instead of spending
              the turn on nothing. THIS IS THE KNOB TO REACH FOR if
              you see replies failing to parse; it is the supported
              way to change reply-to-action recovery, and it already
              runs on 41.6% of steps in one environment measured.
              Ignored where last-line salvage cannot be correct.
            obs_char_cap       int -- 0 (RECOMMENDED) means the
              observation reaches the agent whole. Any positive value
              keeps only the head and tail of a longer observation and
              DELETES THE MIDDLE. Measured 2026-08-06: at 4000, one
              discovery call returning 9094 chars lost 5094 from its
              middle, so the affordances listed there did not exist as
              far as the agent knew -- worth 13.6pp of pass@2
              (30.4 -> 44.6).
              Nothing the agent can observe reveals what was cut, so
              no analyst will ever report this as a failure mode. Set
              it nonzero only with a measured reason.
            allow_action_fallback bool  (aka action_fallback_live)
            tell_remaining_turns bool  -- prints "Turns remaining: N"
              (aka agent_is_told_remaining_turns, the name
              `describe()` prints; both are read)
          NOT a knob: action_shape. The env's action contract is a
          fact about the environment, not a choice a component gets
          to make.
```

Figure 5: The part of the instructions that all four meta-agents share (2 of 4).

Common Prompt Part (Cont.)

```
knowledge     : an INDEX OF OBSERVED FACTS. This is the kind that carries
                discovered specifics: which phrasings the environment
                accepted, which kinds of target turned up in which places,
                which distinctions it enforces. Body is plain text, and it
                is appended to the system prompt VERBATIM -- it is never
                paraphrased, because a paraphrase of a table loses the
                table. Write it compactly (one line per entry) and add rows
                as the transcripts reveal them; an entry stays until the
                evidence contradicts it.
                A knowledge component is only legitimate if every row came
                off a transcript. Rows you could have written from memory of
                a benchmark are the leak the rule above bans.

                GRANULARITY -- types, never instances. Provenance is not
                enough: a row must be about a KIND of thing, not a numbered
                one. "this kind of item is often found in this kind of
                container" generalises to a scene you have never seen.
                "the mug is in cabinet 4" is an answer key for the episodes
                you happened to sample -- equally observed, but worth
                nothing anywhere else, and it inflates any score measured on
                those same episodes. Strip the numbers: if a row names a
                specific numbered entity as the location of a specific
                object, it is instance-level and must be rewritten as a type
                row or dropped. (Numbers are fine when they are part of a
                command's grammar -- 'go to cabinet 1' as a syntax example
                -- or when the environment only ever exposes one, e.g. a
                single appliance.)

                ALWAYS-ON, AND THAT IS A COST. A `knowledge` body sits in
                the system prompt for every episode, so it must earn its
                place on ALL of them. Measured 2026-08-01: one index helped
                one class of task by +11pp and HURT another by -14pp (both
                p<0.005), netting out to a component that looks mildly
                positive and is in fact two different components glued
                together. If rows help only when some condition holds,
                do not write a `knowledge` component -- write a PROCESSOR
                that returns {"inject": "<the same verbatim rows>"} under
                a guard it evaluates itself. That is the only way to make
                content conditional; `applies_when` does NOT gate anything
                at runtime, and its axes describe the whole run, not an
                episode. Guard on what the agent can legitimately see --
                the goal text it was given, the observation, the
                affordances, the step count -- never on an assumption about
                which benchmark this is.
PROCESSOR CONTRACT  -- read the two warnings, they are the observed failures
 def advise(ctx): ...
 `ctx` is an OBJECT with ATTRIBUTES, not a dict. Use `ctx.last_obs`.
 `ctx.get("last_obs")` raises AttributeError. Attributes available:
   ctx.step, ctx.max_steps, ctx.remaining, ctx.goal, ctx.last_obs,
   ctx.last_action, ctx.affordances (list[str]), ctx.messages,
   ctx.history (list of dicts with keys action/obs/reward),
   ctx.scratch   -- a plain dict you own; persists across the episode.

 `ctx.goal` is the task statement, but its SHAPE varies by environment: some
 envs author a terse goal line, others state the task only inside the first
 observation, in which case ctx.goal is that whole observation text. So it is
 always non-empty, and it may be long and may carry surrounding page or scene
 text. Never assume a fixed format, and never gate your whole component on a
 narrow pattern matching it -- a pattern that misses means your component
 returns nothing for the entire run and reads as a failed idea rather than a
 broken one. If you parse it, keep a fallback path for when the parse is empty.
```

Figure 6: The part of the instructions that all four meta-agents share (3 of 4).

Common Prompt Part (Cont.)

```
Return a FLAT dict. Do NOT nest it under the hook name.
    correct :  return {"rewrite_obs": new_text}
    WRONG   :  return {"after_step": {"rewrite_obs": new_text}}
 Valid keys, by the hook you declared:
    before_model : "inject" (str, appended as its own turn) | "disable_tools" (bool)
    after_model  : "rewrite_action" (str)
    after_step   : "rewrite_obs" (str)
 Return {} when there is nothing to say.

 Available names: `re`, `json`, and ordinary builtins including the exception
 types. Nothing else can be imported, and there is no filesystem or network.

 Worked example -- note attribute access and the flat return:

    def advise(ctx):
      seen = ctx.scratch.setdefault("seen", [])
      if ctx.last_action:
        seen.append(ctx.last_action)
      repeats = [a for a in set(seen) if seen.count(a) > 1]
      if repeats:
        return {"inject": "Already tried without effect: " + ", ".join(repeats)}
      return {}

 Your processor is EXECUTED against a probe context before it is admitted to
 the library. If it errors, returns a non-dict, or nests its payload, it is
 rejected -- so keep it simple and match the shapes above exactly.

APPLICABILITY AXES -- the only vocabulary for `applies_when`:
 bottleneck: knowledge | reasoning | memory | perception | protocol
 horizon: short | medium | long
 reward: sparse | dense
 affordances: listed | unlisted
 obs_size: small | large
 action_space: enumerated | compositional | freeform
 termination: payload_submit | explicit_submit | implicit_state | unknown
Leave an axis out to mean "applies regardless".
```

Figure 7: The part of the instructions that all four meta-agents share (4 of 4).

Analyst Prompt

You are a harness ANALYST. You are given rollouts produced by a small, frozen language-model agent acting in an unknown text environment under a given harness. Your job is to find what is holding the agent back and to propose reusable harness components that would fix it.

Common Prompt Part

WHAT MAKES A GOOD CANDIDATE
* It targets a failure mode you can point at in the transcripts.
* It is a MECHANISM, not an exhortation. "Track which containers you have already searched and do not revisit them" beats "be more careful".
* It would still make sense in a different environment with the same abstract structure. If it only makes sense here, it is worthless to us.
* Prefer few, sharp components over many vague ones. 2-5 is a good number.
* If the dominant failure is that the agent never produced a well-formed action at all, that is a PROTOCOL problem and outranks everything else.

MINE THE WINS, NOT ONLY THE LOSSES -- AND MINE THE PAIR BEFORE THE POPULATION
Failures are what needs fixing and every candidate above is framed as a repair, so one kind of component cannot be found that way: a WORKED EXAMPLE is lifted out of an episode that WORKED.

COMPARE WITHIN A GROUP, AND ACROSS THEM. A solved and a failed sample of the SAME task differ only in sampling, so task difficulty is held fixed and any difference between them is attributable. That is the strongest single comparison in front of you and it is why the sample is grouped at all. It is not the only one. A defect that every sample of a task hits the same way leaves no contrast INSIDE that group; it appears instead as the same kind of failure recurring in DIFFERENT groups. Read across the groups as well as within them. Cross-group evidence is weaker per observation -- difficulty is not held fixed, so something "only the winners do" may belong to the easier task they happened to be -- so state which of the two a candidate rests on, and do not leave the weaker one unsaid merely because it is weaker.

So, on any environment whose action is a code or command block: read the opening turns of the SOLVED samples and ask whether they share a first move their FAILED groupmates never make -- and, where a group has no solved sample at all, whether solved samples in OTHER groups share one the failures do not. If they do, that opening is a candidate -- `prompt_fragment`, a one-line rule plus the move demonstrated inside the runner's action tags, written as more than one line. See `prompt_fragment` in the kinds list above for the two rules it must obey.

This is worth its own instruction because nothing else in this brief would ever produce it. Measured on the archive: across 1,585 components in 80 libraries, ZERO prompt_fragments contained a line of code -- while the one operator that happened to write examples accumulated seven of them and beat the operator that wrote none by 30.0pp of train pass@2. The mechanism is not subtle: 42.8% of that arm's episodes opened by reusing a line from an example verbatim.

Do not manufacture one. If the solved episodes open no differently from the failed ones, say so and propose nothing -- an example that demonstrates nothing the agent was not already doing costs prompt and buys nothing.

FACT OR PROCEDURE -- answer this for EVERY candidate via `obtained_in_episode`
A fact the agent could have looked up at runtime should NOT be written down. Written-down facts go stale, cannot cover cases you never saw, and are one careless edit from vanishing. Teach the LOOKUP instead -- it regenerates the fact every episode.

Figure 8: The analyst's instructions (1 of 2).

Analyst Prompt (Cont.)

```
So, for each candidate, find a rollout+turn where ONE action caused the
 environment to reveal this fact, and cite it:
   "obtained_in_episode": {"rollout": 3, "turn": 7, "action": "<the action>"}
 If you can point at such a turn, prefer emitting the PROCEDURE ("before
 calling X, run Y to read its spec") over the fact itself.

 If NO single action in any rollout yields it -- because it is a regularity you
 can only see by comparing many episodes -- say so explicitly:
   "obtained_in_episode": null
 That is a legitimate and important answer: such knowledge MUST be written down,
 because nothing at runtime supplies it.

 Do not guess a citation. It is checked against the transcripts, and a citation
 that does not resolve gets the candidate rejected.

Write processor bodies as complete, runnable Python. Keep prompt fragments
short -- every character competes for the agent's attention.
```

Figure 9: The analyst's instructions (2 of 2).

Editor Prompt

```
You are a harness EDITOR. Several independent analysts analysed the SAME
rollout batch and each proposed candidate components. Their proposals overlap,
conflict, and vary in quality. Produce one merged, ranked set.
```

Common Prompt Part

```
WHY YOU EXIST
Single-pass analysis of this kind is measurably unstable: independent runs over
identical data have flipped a capability between "recommend" and "avoid".
Agreement across analysts is therefore your strongest signal. A candidate that
several analysts reached independently is far more likely to be real than an
elegant one that appeared once.

YOUR OPERATIONS
 MERGE  two proposals for the same mechanism into the better-stated one.
 EDIT   to remove environment-specific detail, tighten wording, fix a
        processor that would crash or that mutates state it does not own.
 DROP   anything that is (a) an exhortation rather than a mechanism,
        (b) built on a fact NOT visible in the transcripts -- recalled, not
        observed (note: a concrete observed fact is not a reason to drop;
        judge the mechanism's generality, not the content's specificity),
        (c) over-prescriptive enough to collapse the agent's behavioural
        diversity, or (d) contradicted by the aggregate statistics.
 KEEP   the count small. Return at most 6 candidates.

 A `knowledge` component is the exception to "keep the count small": prefer
 EXTENDING the existing one with newly observed rows over minting a second.

For each surviving candidate report how many analysts proposed it (`support`).
```

Figure 10: The editor's instructions.

Librarian Prompt

You are the LIBRARIAN. You own the harness library: a modular, versioned store of reusable components that a composer later draws on to build harnesses for new tasks. An editor has handed you newly selected candidates. Decide what the library should look like next.

Common Prompt Part

```
WHAT A GOOD LIBRARY LOOKS LIKE
 * Every component earns its place. Overlapping near-duplicates are worse
   than one well-stated component, because they split usage statistics and
   bloat the index the composer has to read.
 * Components are as generic as they can be while staying actionable.
 * `applies_when` is honest. A component marked as applying everywhere had
   better actually help everywhere; over-claiming is how a library rots.
 * `one_line` is the composer's ONLY view of a component during selection.
   It must say what the component does and when to reach for it, in one line.
 * Index entries carry `live_in=Nr mean_sr=X%` -- the rounds a component was
   active in and the mean score of those rounds. This is CO-OCCURRENCE, not
   credit: components run in bundles, so a high mean_sr may belong to a
   companion. Treat it as weak evidence, and weigh it against a component
   that has never been tried at all (which is not evidence of failure).

OPERATIONS (return a list; they are applied in order)
  add    a genuinely new component
  update  an existing component (version bumps)
  merge   fold sources into one target, then drop the sources
  drop    remove a component -- say why

  *** `add`, `update` and `merge` MUST each carry a COMPLETE `component`
  object: id, kind, title, one_line, body, rationale, hook, order,
  applies_when. Not a reference to an existing id -- the FULL object, with the
  body written out in full even when you are only changing one line of it.

 An operation without its `component` payload is DISCARDED. On `merge` that
  would otherwise delete the sources and put nothing back, so a payload-less
  merge is refused outright and the sources are kept.

  `drop` is the only operation that needs no payload; give `target_id`. ***

DELETIONS INSIDE A BODY MUST BE DECLARED
  An `update` replaces the whole body, so a consolidation can delete a line
  you never meant to touch and nothing about the reply would show it. Twice
  now a rewrite that added a section quietly took out the one line stating how
  to submit an answer, and the agent's score went to zero the next round.

  So: if you are removing anything from an existing body ON PURPOSE, list it
  in `deletes` -- the removed lines, verbatim or close, one per entry, and say
  why in `reason`. Any substantive line that vanishes WITHOUT being declared is
  treated as an accident and put back, and you will see the repair listed in
  your next prompt. Declaring costs you one line; not declaring costs the run
  a fact the agent needed. Rewording a line is not deleting it and needs no
  declaration.

BE CONSERVATIVE ABOUT DROPPING components with a real usage history, and
AGGRESSIVE about dropping ones that have been selected repeatedly without ever
being associated with a gain. Untried components are not evidence of failure.
```

Figure 11: The librarian's instructions.

Composer Prompt

You are the COMPOSER. Given what is known about a task and the current harness library, build the harness the agent will actually run.

Common Prompt Part

THE FRAME IS LOAD-BEARING
Components carry the facts; the frame you author is what makes them usable together. A composition that keeps every component and rewrites the frame at a third of its former length is not a smaller harness -- it is a different one, and it has been measured losing three quarters of the score in a single round.
If an incumbent is shown below, its shape is given with it. Write a frame of comparable weight unless you can say why less is right, and say so in `rationale`. Writing materially less than the incumbent's authored frame is a DECISION, not a default -- exactly as dropping one of its components is. Where a fix needs more room than the incumbent had, take the room.
YOUR OUTPUT
Index entries may carry `live_in=Nr mean_sr=X%`: the rounds a component was active in, and those rounds' mean score. It is co-occurrence, not credit -- components run in bundles. Prefer a component with a strong record over an untried one when they serve the same purpose, but do not drop something that was live during your best round without a reason.
base_system The agent's system prompt. It MUST fully specify the reply protocol, because the agent has no other instructions. Do not pad, and do not restate a fact an assembled component already carries; but a frame too thin to connect the components it introduces is not economical, it is broken, and stating a rule properly is not padding.
Selected prompt_fragments should be woven in, not pasted as a list of bullet points bolted onto the end.

NON-NEGOTIABLE: base_system must contain a reply-format paragraph that shows the literal tag the runner's extractor requires -- <action>...</action> for protocol `tagged` or `tagged_last`, <code>...</code> for `code_first` or `code_last` -- written out as literal text the agent can copy. A reply that does not carry that tag is DISCARDED UNREAD; on code environments there is no last-line fallback to save it. This was measured, not imagined: a composed prompt that never named the tag made 90 of 90 episodes fail to act on turn 1 and threw away 19% of all turns. Everything else you write is advice; this one paragraph is the interface.
SECOND NON-NEGOTIABLE, and it outweighs any advice you can write: the worked example in that paragraph must ITSELF span more than one line. This environment executes the tag contents as a single unit, so several statements can be sent per turn. The agent copies your example far more faithfully than it follows your prose -- measured, on 342 episodes: a prompt that granted multi-line in a sentence but showed a one-line example produced multi-line on 2.0% of actions; rewriting only the example took it to 25.5%, and most episodes opened by copying that example verbatim. Write it as two or three statements the environment actually accepts, drawn from what exploration showed you. Never state the permission and then contradict it with a single-line demonstration.
select The component ids you are using. Select for FIT, not coverage: an unnecessary component costs attention and can suppress the behavioural diversity that repeated attempts rely on.
knobs Runner settings.

The agent running this harness is small and frozen. It follows concrete, mechanical instructions far better than abstract advice.

Figure 12: The composer's instructions.

B.8 THE GATE

Let $v_{j,i}$ be the pass@1 of worker i on the val tasks in round j, and let M' be the number of val rollouts per task: $M' = 4$ in the inductive runs ($M' = 2$ on SWE-bench Verified), and $M' = M$ in the transductive runs, which reuse the train rollouts. The noise band after round r pools the within-round spread of these scores,

$$n_r \;=\; 2\,\hat{\sigma}_r\sqrt{2/M'}, \qquad \hat{\sigma}_r = \frac{1}{r+1}\sum_{j=0}^{r}\frac{\max_i v_{j,i} - \min_i v_{j,i}}{d_2(M')}, \tag{2}$$

with the range-to-deviation constants $d_2(2) = 1.128$ and $d_2(4) = 2.059$. The margin is $\delta = \max\big(\min(7,\, b^\star/2),\; n_r,\; \Delta^\star\big)$ pp, where $\Delta^\star$ is the spread of the best harness's repeated measurements, and the loop applies Eq. (1) by checking the following conditions in order.

1. In the first round, the harness that ran becomes the best harness.
2. If a revert made the best harness run again, the loop holds, and $b^\star$ becomes the mean of its measurements.
3. If more than 5% of its replies were unparsed, the loop holds.
4. If $s_t > b^\star$, it accepts.
5. If $s_t < b^\star - \delta$, it reverts.
6. Otherwise it holds.

The loop scores a harness in the round after its composer wrote it, from rollouts drawn in that round (in the transductive regime, the very rollouts that the round's analyst reads), so the composition written in the last round is never scored. A revert discards the composition built from the reverted harness's rollouts, while the library keeps that round's edits.

Output and Grading. The output is the last accepted best harness, whose system prompt is rebuilt from the library snapshot of the round that composed it and must match the original byte for byte. A test run then gives it two fresh attempts per test task, and with two attempts the per-task union equals the estimator of Eq. (3).

B.9 HUMAN-WRITTEN INPUTS

This is the complete list of human-written inputs; none of them names a benchmark, and the meta-agents' instructions pass the same identity filter that guards a run.

1. Text for the agent: the neutral harness, the runner's reply-format block, the runner's labels (task, observation, available actions, turns remaining), and the reminder sent after a reply without an action.
2. Instructions for the meta-agents: a role description for each, the rule against using recalled knowledge of a benchmark, documentation of the component kinds, the processor interface and the applicability axes, and a few measurements from our development runs, stated without naming an environment; some came from AppWorld, including the split of its inductive results (Appendix B.7).
3. The three strategies (Appendix B.5).
4. The workflow: the order of the stages, the sampling rule, the admission checks, the gate, R and M.
5. Each benchmark's interface: whether actions are code or commands, the turn cap, the generation budget and, on WebShop, the syntax of its two actions (Appendix C.1).

B.10 CONFIGURATION

Table 10 lists the settings of all runs.

Table 10: Settings of the LiteEvo runs. Exceptions: SWE-bench Verified uses $M = 2$, five mixed plus two all-fail groups and two val rollouts per task.

Setting	Value
Agent	Qwen3.5-9B in thinking mode; temperature 1.0, top-p 0.95, top-k 20, presence penalty 1.5
Generation and context	4,096 tokens per reply; context window 40,960 tokens; prompt budget 32,768 tokens
Turn caps	20, 20, 40, 20 and 200 (ALFWorld, WebShop, AppWorld, GAIA, SWE-bench Verified)
Rounds and workers	$R = 8$ (round 0 and seven evolution rounds), $M = 4$
Val rollouts	4 per task (inductive); the train rollouts (transductive)
Analyst sample	10 mixed and 3 all-fail task groups, each with all M trajectories
Editor	at most six candidates; journal of earlier rounds; processor check on 24 episodes
Gate	pass@2; accept if $s_t > b^\star$; revert if $s_t < b^\star - \delta$
Grading	three test runs of two attempts per task; the harness is rebuilt byte for byte
Meta-agents	analyst Claude Sonnet 5; editor, librarian and composer Claude Opus 4.6; high effort

C BENCHMARKS AND EVALUATION

C.1 BENCHMARKS

Table 11 lists the task sets, whose splits are disjoint from one another at the task level, and the adapters pass the environments' own text to the agent.

Table 11: Task sets. Inductive runs evolve on the train tasks, gate on the val tasks and grade on the test tasks. Transductive runs use the test tasks for all three.

Benchmark	Turn cap	Train	Val	Test
ALFWorld	20	134 (train)	140 (valid-seen)	134 (valid-unseen)
WebShop	20	100	100	100
AppWorld	40	90 (train)	57 (dev)	168 (test-normal)
GAIA	20	–	–	103 (transductive only)
SWE-bench Verified	200	55	55	55

ALFWorld casts six household task types as text games. The agent receives the game's own opening text, which states the goal, and each observation lists the commands the environment admits; an action is one command, such as "heat apple 1 with microwave 1". The episode ends when the goal is met or after 20 turns, with no partial credit, and the test games are set in rooms that no training game uses.

WebShop asks the agent to buy a product that matches an instruction, on a simulated shop with 1,000 products and an in-memory BM25 search. The environment accepts `search[query]` and `click[label]` but never states this syntax, so our adapter lists the legal actions in that form; apart from the actions that the GAIA bridge defines below, this is the only action syntax we supply. Buying ends the episode with a reward in $[0, 1]$ for how well the product matches, and success is a reward of 1; the three splits are windows of 100 instructions with no product in common.

AppWorld simulates nine everyday apps with 457 APIs. The agent sees only the task and the identity of the user it acts for, and since nothing states that actions are Python or that the APIs exist, it must discover both from tracebacks and from the environment's documentation app. The episode ends when the agent calls the API that declares the task complete, or after 40 turns, and state-based unit tests then score it: partial credit is the fraction of tests passed, and success requires all of them. We use the official train, dev and test-normal splits and report the harder test-challenge split separately (Appendix D.2).

GAIA poses questions that need web research, files and code, each with one reference answer. Since GAIA ships no environment, our bridge provides one, whose opening text defines six actions after the question: search, open a page, read a local file, run Python, run a shell command, both without network access, and submit. Submitting ends the episode, and the answer scores 1 if it matches the reference under GAIA's normalized exact match. We use the 103 text-only validation questions on which HarnessX also reports; since their answers are public, the test answers cannot be withheld, and 103 questions are too few to split three ways, so GAIA is transductive only.

SWE-bench Verified asks the agent to resolve a GitHub issue by editing the repository, and each episode runs in the official container of its instance with networking disabled. The agent sees the issue and a listing of the repository, and each action is a shell command. The episode ends when the agent submits or after 200 turns, and the official harness then grades the resulting patch with tests whose names the agent never sees. After excluding three instances, we draw three disjoint sets of 55 instances, stratified by repository and difficulty.

C.2 GRADING

With $n \geq k$ attempts per task, c_τ of which succeed on task τ, we estimate pass@k with the unbiased estimator of Chen et al. (2021):

$$\text{pass@}k(h;\mathcal{T}) = \frac{1}{|\mathcal{T}|}\sum_{\tau\in\mathcal{T}}\left[1 - \binom{n-c_\tau}{k}\Big/\binom{n}{k}\right]. \tag{3}$$

A test run gives the returned harness two fresh attempts per test task: its pass@1 is the mean success rate of the two attempts, and its pass@2 is the share of tasks solved by either, which equals Eq. (3) with $n = k = 2$. Tables average three test runs, and episodes lost to infrastructure faults are excluded from both counts. The loop's own scores, such as the curves of Figure 13, use Eq. (3) over the M rollouts of a round. The SWE-bench Verified inductive run differs from the others in two settings. Its analyst also receives the share of episodes that never edited a file, a statistic we added after inspecting test-split episodes of an earlier SWE-bench Verified run, and knobs that the runner cannot honor for the benchmark's action format are dropped rather than composed.

C.3 API COST

The API cost counts only the meta-agents' tokens in the calls that produce h_1 to h_7: the last round's calls compose a candidate that is never run, and we do not count them. Since the agent runs on local GPUs, neither method's rollouts cost API calls. For HarnessX we sum the token counts of every call in its first seven evolution steps, context compaction included. For LiteEvo, whose logs record each meta-agent's cumulative token counts after every round, we take the cost up to the composition of h_7, and where a run's log is lost, seven eighths of the run's total. We price both at \$5 and \$25 per million input and output tokens for Opus 4.6 and at the introductory \$2 and \$10 for Sonnet 5. Both pipelines cached their fixed system prompts during the runs, but we bill every input token at the full price, cached or not, so that no cost depends on how a run used the cache. For five runs whose token records were lost, the SWE-bench Verified transductive run of LiteEvo and four Qwen3.5-4B runs (ALFWorld in both regimes, GAIA and SWE-bench Verified transductive), only the total recorded for the run survives. We convert it to our convention with the ratio measured on a run of the same benchmark whose records survive and take seven eighths of the result.

Table 1's cost gap has two sources: the number of calls, and the analyst's cheaper model, which reads most of LiteEvo's tokens. HarnessX's meta-agent made 65 to 130 Opus calls per round, because each round is an agentic session of dozens of turns, whereas LiteEvo made about five calls per round on the four benchmarks whose records survive. HarnessX's calls averaged 43,000 to 50,000 input tokens, while LiteEvo's analyst read more per call, 65,000 to 365,000, and its other meta-agents far less, 6,000 to 12,000, so that HarnessX consumed 3.3 to 5.6 million input tokens per round against 0.12 to 0.54 million for LiteEvo. The trajectory text itself is of comparable size: about 38,000 to 217,000 tokens per round reach HarnessX through its tools, and at most 57,000 to 357,000 reach LiteEvo's analyst per call. HarnessX, however, keeps what it reads in a context that it re-sends on every later call, which accounts for 25% to 36% of its input.

C.4 ANSWER COPIES ON GAIA

GAIA's questions and answers are public, and copies of them sit on the open web, in dataset pages, papers and code repositories. A searching agent can reach a copy in two ways: it can open a page that states the answer, or it can search the question's text verbatim, which tends to surface such pages. Because the web changes, no two runs see the same pages, and a GAIA result cannot be reproduced exactly. LiteEvo's search and page tools block known copies of the benchmark, namely pages on a list of known mirrors, results that contain a task's answer record or its annotator metadata, and GAIA task identifiers, and they fail closed.

Mirrors missing from the list are not blocked: in the LiteEvo run that Table 1 reports, a list-free scan of observations found 30 cases in which the answer reached the agent from such a copy before the agent had produced it. In the round that ran the returned harness, 10 of these cases fell in winning episodes. Scoring those episodes as failures would leave that round's pass@2 no lower than 62.9, against 64.6 as measured, an upper bound on what the copies bought, since the agent may also have found the answer on its own. Evolution also learned from these episodes: one returned processor tells the agent that hard-to-find facts sometimes appear on dataset mirror pages. We report the run's scores as measured; the scan covered the loop's episodes only.

C.5 THE HARNESSX REPRODUCTION

The released HarnessX code ships its harness framework, a single-agent meta-agent, adapters for GAIA and SWE-bench, and an evolution recipe for GAIA, but not the four-stage evolution engine on which the paper's results rest (Chen et al., 2026), so we run the released single-agent meta-agent on every benchmark. In the paper's own comparison on GAIA, a single-agent evolver scores 1.0 pp lower than the four-stage engine and uses about 14% more tokens. Table 12 lists which parts of the reproduction are official and which we wrote, and we cap evolution at eight rounds with one run, allow 4,096 tokens per reply, and use our turn caps. On four benchmarks, its meta-agent runs without extended thinking. On WebShop, AppWorld, GAIA and SWE-bench Verified we also turned on HarnessX's optional caching of its system prompt, which leaves its costs unchanged, because Appendix C.3 bills every input token at the full price. HarnessX's loop scores pass@2 in every round and reports its best round on the evolution tasks, as the paper does. On GAIA, HarnessX peaked at round 2, and we evaluate that round's harness.

Table 12: Provenance of each part of our HarnessX runs. *Official* is the released code, unmodified; *patched* adds our changes; *ours* is absent from the release and written by us. The meta-agent is Claude Opus 4.6 on every benchmark.

Benchmark	Seed harness	Evolution recipe	Meta-agent reasoning	Other deviations
ALFWorld	ours	ours	thinking off	task seeds cover 88 of the 134 test games
WebShop	ours	ours	thinking off	BM25 search
AppWorld	ours	ours	thinking off	not in the HarnessX paper; no early stopping
GAIA	official	official, patched (pass@2)	official (extended thinking)	a different search API; answer judge off
SWE-bench Verified	official	ours	thinking off	200 turns (release: 60); our grader

Three differences favor one side or the other: (i) HarnessX's released SWE-bench runner, which we kept as released, names in the agent's task description the tests that must pass, whereas LiteEvo's agent never sees them; (ii) the task seeds of HarnessX's ALFWorld run cover only 88 of the 134 test games; and (iii) AppWorld is not a HarnessX benchmark, so its HarnessX seed is ours, written in the style of HarnessX's other seeds.

D ADDITIONAL RESULTS

Section 4 gives each result below in one sentence; unless a caption or the text says otherwise, the agent is Qwen3.5-9B, the runs are transductive, and scores average three test runs.

D.1 A SMALLER BACKBONE: QWEN3.5-4B

Table 13 repeats the main experiment with Qwen3.5-4B, starting from the neutral harness. LiteEvo raises pass@2 in all nine benchmark and regime cells by 6.7 to 49.8 pp, at $7.78 to $22.45 per run, with the largest gains on WebShop and ALFWorld and the smallest on SWE-bench Verified. The inductive harnesses stay within 2.7 pp of the transductive ones.

Table 13: Test pass@2 (%) of Qwen3.5-4B with the harness that LiteEvo evolves from the neutral harness, averaged over three test runs. *Initial* is the neutral harness. Δ is Evolved − Initial, before rounding. The GAIA run blocks known copies of the benchmark's answers.

Benchmark	Mode	pass@2 (%)		Δ	Best Round	API cost ($)
		Initial	Evolved			
ALFWorld	Transductive	44.8	88.3	+43.5	7	7.78
	Inductive	44.8	87.3	+42.5	7	7.86
WebShop	Transductive	15.2	65.0	+49.8	4	9.06
	Inductive	15.2	62.3	+47.2	2	11.39
AppWorld	Transductive	0.0	33.9	+33.9	6	11.32
	Inductive	0.0	33.7	+33.7	5	12.23
GAIA	Transductive	33.7	55.3	+21.7	7	10.64
SWE-bench Verified	Transductive	46.0	53.9	+7.9	3	15.31
	Inductive	46.0	52.7	+6.7	7	22.45

D.2 APPWORLD TEST-CHALLENGE SPLIT

Table 14 grades the AppWorld harnesses on the test-challenge split, whose 417 tasks need apps that the train split never uses; the bare agent and the neutral harness solve none of them. LiteEvo's harnesses reach 48.6 pass@2 when evolved inductively and 49.2 when evolved on test-normal, against 50.5 for HarnessX. HarnessX, however, starts from a handcrafted AppWorld seed that alone scores 32.8, and its run costs $190.32 against $14.56 and $12.51 for LiteEvo.

Table 14: Results on AppWorld's test-challenge split, whose tasks need apps unseen in training (Qwen3.5-9B). *Bare* gives the agent only the task text, and *Bare + format* adds a minimal hint on the reply format. *Neutral* is LiteEvo's starting harness, and *HarnessX-Seed* the handcrafted HarnessX-style one that we wrote for AppWorld. The evolved rows use the harness each run returns; HarnessX evolves transductively, on test-normal. P@1 and P@2 are pass@1 and pass@2 in %, averaged over three test runs. API cost is that of the evolution run.

Harness	Mode	P@1	P@2	API cost ($)
Bare	–	0.0	0.0	–
Bare + format	–	0.0	0.0	–
HarnessX-Seed	–	21.7	32.8	–
HarnessX	Transductive	**36.0**	**50.5**	190.32
Neutral	–	0.0	0.0	–
LiteEvo	Inductive	34.5	48.6	14.56
LiteEvo	Transductive	35.2	49.2	12.51

D.3 Evolution trace on AppWorld

Figure 13 follows the transductive AppWorld run of Table 1 round by round. Since neither the task text nor the neutral harness says that actions are Python or how to finish a task, the agent starts at zero. Each round then adds at most four components and updates or merges at most two (Table 17), all aimed at failures in the previous round's trajectories.

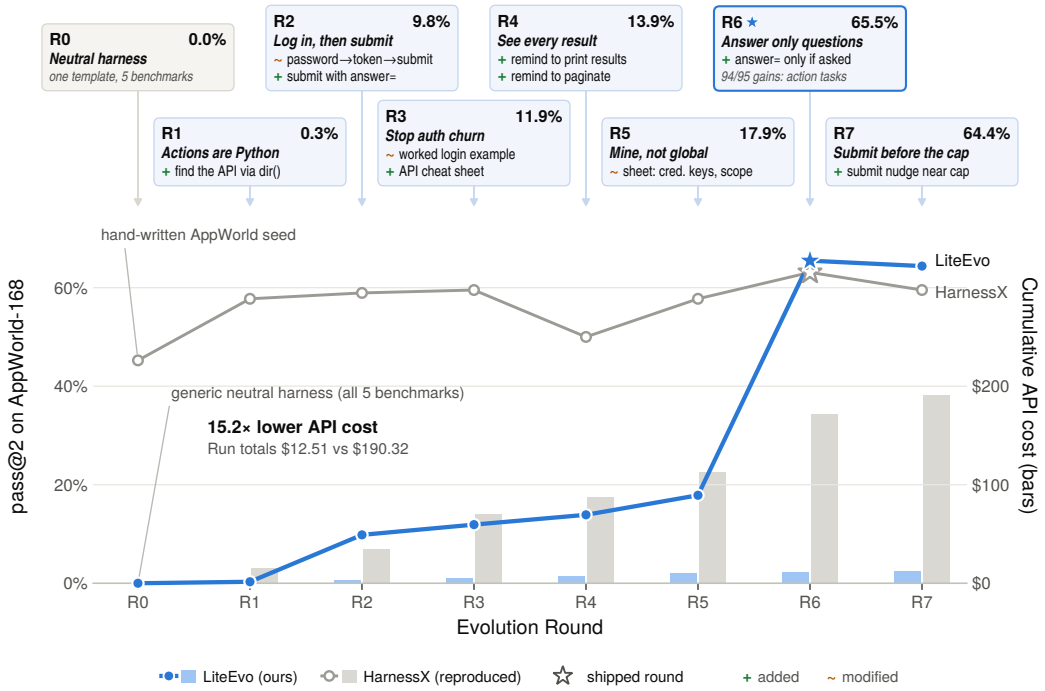


Figure 13: LiteEvo reaches its best AppWorld harness from a zero-scoring start in six rounds, at 15.2× lower API cost than HarnessX. One transductive run on AppWorld's 168 test-normal tasks: pass@2 of each round's harness, measured inside the loop (lines); the cumulative API cost of producing each round's harness (bars); and the components each round added (+) or modified (∼). The star marks the harness that LiteEvo returns, whose graded pass@2 over three test runs is 67.7 (Table 1). Our HarnessX reproduction starts from a hand-written AppWorld seed (Appendix C.5).

D.4 Transfer across benchmarks

We tested whether a library evolved on some benchmarks helps an unseen one, composing harnesses for AppWorld's test-challenge split from a library built on the other four benchmarks, with no AppWorld component. In three draws over the 417 tasks, they won 1 of 2,502 episodes, where the harnesses that LiteEvo evolves on AppWorld itself reach 48.6 and 49.2 pass@2 (Table 14) and the neutral harness 0.0. On AgentDiff, a sibling of AppWorld with 45 tasks, the test is too small to resolve even the gain of AgentDiff's own library, so it only bounds the transfer. The composer rarely picks foreign components, and the only kind that transferred is a loop-breaking processor, which every benchmark's library had invented on its own. A library's value lies in worked examples written in its environment's own names and actions, which a new benchmark cannot use. The procedure transfers instead: a frontier model that reads only the new benchmark's 90 train transcripts writes worked examples worth 12.6 to 16.3 pp over a harness without them, 55% to 71% of the 23.1 pp that the worked examples of a fully evolved library add.

D.5 ANALYST SETUP

Table 15 compares failure-oriented group sampling, in which one analyst reads the grouped trajectories of all workers, with one analyst per worker, each reading ten failures and three successes of its own worker. Group sampling costs less in all nine benchmark and regime cells, at 36% to 75% of the per-worker cost, and its pass@2 is higher in five cells, within 0.2 pp in two, and lower in the two ALFWorld cells, by 1.7 and 1.5 pp, where both setups exceed 94. We use group sampling by default.

Table 15: Analyst setup under Compose (Appendix D.10; Qwen3.5-9B). *Per-worker* runs one analyst for each worker, on that worker's own trajectories. *GRP* is failure-oriented group sampling: a single analyst reads grouped trajectories of all workers. Scores average three test runs, and API cost is that of one run.

Benchmark	Mode	Analyst setup	# Analysts	pass@1 (%)	pass@2 (%)	Best Round	API cost ($)
ALFWorld	Transductive	Per-worker	4	96.6	98.0	7	19.26
		GRP	1	92.6	96.3	5	8.24
	Inductive	Per-worker	4	92.5	96.3	6	20.05
		GRP	1	90.3	94.8	4	9.17
WebShop	Transductive	Per-worker	4	51.8	62.7	1	24.78
		GRP	1	54.0	62.7	5	9.00
	Inductive	Per-worker	4	56.0	66.0	4	21.62
		GRP	1	61.0	70.0	3	9.89
AppWorld	Transductive	Per-worker	4	40.3	54.6	3	22.78
		GRP	1	51.9	67.7	6	12.51
	Inductive	Per-worker	4	49.4	65.3	6	24.83
		GRP	1	49.4	65.5	7	14.56
GAIA	Transductive	Per-worker	4	49.8	63.4	7	19.58
		GRP	1	53.6	65.2	7	10.59
SWE-bench Verified	Transductive	Per-worker	2	54.4	61.8	1	35.92
		GRP	1	54.2	63.2	6	24.30
	Inductive	Per-worker	2	52.7	61.8	1	37.84
		GRP	1	58.4	66.7	4	28.50

D.6 TASK SELECTION FOR THE ANALYST

Table 16 changes which task groups the analyst reads while keeping their number at 13 (7 on SWE-bench Verified). Our default reads ten mixed groups and three all-fail groups, while the alternatives read only groups that all workers solved, only groups that no worker solved, or groups drawn at random, filling a short population from the other groups. The default gives the best pass@2 on all five benchmarks, though on SWE-bench Verified only by 0.2 pp over the all-success groups. The choice matters most on AppWorld, where all-success and all-fail groups reach only 16.9 and 12.3 against 67.7. A failure is plausibly easiest to diagnose next to a success on the same task, which only mixed groups provide.

Table 16: Which task groups the analyst reads (Qwen3.5-9B, transductive): always 13, or 7 on SWE-bench Verified. *Default* reads 10 mixed-outcome groups and 3 all-fail groups (5 and 2 on SWE-bench Verified). *All-success*, *All-fail* and *Random* read groups that every worker solved, groups that no worker solved, or random groups, each filled up from the other groups when too few exist. Scores average three test runs.

Benchmark	Sampling	pass@1 (%)	pass@2 (%)	Best Round
ALFWorld	Default	**92.6**	**96.3**	5
	All-success	79.9	86.6	3
	All-fail	84.7	89.3	7
	Random	90.1	94.3	7
WebShop	Default	**54.0**	**62.7**	5
	All-success	49.5	62.0	7
	All-fail	52.7	60.7	6
	Random	52.7	62.0	1
AppWorld	Default	**51.9**	**67.7**	6
	All-success	13.4	16.9	5
	All-fail	9.3	12.3	2
	Random	43.9	57.9	6
GAIA	Default	53.6	**65.2**	7
	All-success	**53.7**	64.7	7
	All-fail	46.4	57.0	4
	Random	46.9	59.2	5
SWE-bench Verified	Default	**54.2**	**63.2**	6
	All-success	53.3	63.0	2
	All-fail	**54.2**	61.8	1
	Random	51.4	60.0	1

D.7 LIBRARY OPERATIONS PER BENCHMARK

Table 17 breaks Table 4 down by benchmark: each run adds components in most rounds and rarely removes one, and in the last round the composer picks half of the library on SWE-bench Verified (8 of 16 components) and all of it on AppWorld (13 of 13).

Table 17: Per-benchmark breakdown of Table 4. *Picked / Size* is the number of components the composer selects over the number it can select from. GAIA counts are reconstructed from library snapshots.

Benchmark	Round	0	1	2	3	4	5	6	7	Total
ALFWorld	Add	4	4	3	2	1	2	1	2	19
	Update	0	0	0	0	1	0	0	1	2
	Merge	0	0	1	1	0	0	0	0	2
	Drop	0	0	1	0	0	0	0	0	1
	Picked / Size	4/4	8/8	10/10	12/12	13/13	15/15	14/16	16/18	–
WebShop	Add	3	2	2	1	1	3	2	0	14
	Update	0	0	0	0	0	0	0	3	3
	Merge	0	0	0	0	0	0	0	0	0
	Drop	0	0	0	0	0	0	0	0	0
	Picked / Size	3/3	5/5	7/7	8/8	9/9	12/12	11/14	9/14	–
AppWorld	Add	4	2	1	2	0	2	2	2	15
	Update	0	1	0	1	1	0	0	1	4
	Merge	0	1	2	0	0	0	0	0	3
	Drop	0	0	0	0	0	0	0	1	1
	Picked / Size	4/4	5/5	6/6	7/8	7/8	9/10	11/12	13/13	–
GAIA	Add	4	3	4	3	3	1	1	2	21
	Update	0	0	2	1	1	0	1	1	6
	Merge	0	0	0	0	0	1	0	0	1
	Drop	0	0	0	0	0	0	0	0	0
	Picked / Size	4/4	7/7	10/11	13/14	14/17	13/18	14/19	16/21	–
SWE-bench Verified	Add	4	0	2	2	1	3	2	2	16
	Update	0	3	2	0	2	0	2	1	10
	Merge	0	0	0	1	0	0	0	0	1
	Drop	0	0	0	0	0	0	0	0	0
	Picked / Size	4/4	4/4	6/6	8/8	6/9	8/12	6/14	8/16	–

D.8 MEMORY IN EVOLVED HARNESSES

The component kinds of LiteEvo include no memory that persists across episodes, yet the evolved harnesses keep information in two ways. Within an episode, processors can use the scratchpad and the step history: one GAIA processor collects the URLs that returned an access wall and warns more strongly as the list grows, and another quotes back any search that the agent repeats verbatim. Across episodes, knowledge and prompt fragments carry what earlier rounds observed, as when the AppWorld harness spells out the whole login procedure that its early rounds had to discover; no state passes from one episode to the next at run time.

Table 18 tests whether an explicit memory across episodes helps, with each variant a hand-written component pinned into every composition. The action book stores a failed action with the next action that worked, the URL blacklist stores domains that blocked access in at least two tasks, and login recipes store the API calls that logged in. Every variant scores lower than the default harness, which reaches 65.2 on GAIA against 55.8 and 58.4, and 67.7 on AppWorld against 15.4. The comparison is not controlled: each variant was measured once, and the AppWorld variants also ran with a different code version and load than the default. The action book, which we had registered in advance as a weak variant, stopped growing at 19 distinct entries by round 2, and none of the variants shows a gain.

Table 18: Pinning into every composition a hand-written memory that persists across episodes (Qwen3.5-9B, transductive). The memory reads and writes a store shared by episodes, and no episode reads entries from attempts on its own task. *Action book* stores a failed action with the next action that worked. *URL blacklist* stores, as an avoid-list, web domains that blocked access in at least two tasks. *Login recipes* stores the API calls that logged in. *None* rows are the default runs of Table 16.

Benchmark	Enforced Memory	pass@1 (%)	pass@2 (%)	Best Round
GAIA	None (default)	**53.6**	**65.2**	7
	Action book	45.1	55.8	5
	URL blacklist	46.1	58.4	7
AppWorld	None (default)	**51.9**	**67.7**	6
	Login recipes	13.2	15.4	6

D.9 MERGING THE EDITOR AND THE LIBRARIAN

Table 19 merges the editor and the librarian into one meta-agent, which also reads the replay results of all candidate processors. The merged runs score lower on all four benchmarks, most on AppWorld, where inductive pass@2 falls from 65.3 to 14.0. These runs, however, differ from the defaults in more than the merge, and no split control arm was run beside them; the only controlled comparison we have, a pilot on WebShop, found no difference. On GAIA, the merged run used one analyst per worker, ran without a working code tool, and was measured once. We keep the two meta-agents separate.

Table 19: Merging the editor and the librarian into one meta-agent (Qwen3.5-9B). *None* keeps the two meta-agents separate: the editor selects and rewrites the analyst's candidates, and the librarian applies them to the library. *Editor + Librarian* merges them and uses one analyst per worker. The None rows are the per-worker runs of Table 15, except on GAIA, whose default is the group-sampling run of Table 16. GAIA is transductive; the other benchmarks are inductive and scored on held-out tasks.

Benchmark	Merge	pass@1 (%)	pass@2 (%)
GAIA	None (default)	**53.6**	**65.2**
	Editor + Librarian	48.1	59.9
ALFWorld	None (default)	**92.5**	**96.3**
	Editor + Librarian	91.0	92.5
WebShop	None (default)	**56.0**	**66.0**
	Editor + Librarian	52.5	61.0
AppWorld	None (default)	**49.4**	**65.3**
	Editor + Librarian	10.6	14.0

D.10 REFINE OR COMPOSE

Table 20 compares two ways for the composer to produce a harness: Compose, our default, builds each harness afresh from the library, whereas Refine composes once after the first round and then edits the carried harness in every round, with one analyst per worker in both. Compose scores higher on average in both regimes, with 68.1 against 64.3 pass@2 transductively and 72.4 against 65.8 inductively, while Refine wins on WebShop in the transductive regime and on SWE-bench Verified in the inductive one.

Table 20: Two ways for the composer to produce a harness (Qwen3.5-9B). *Refine* composes a harness after the first round and then edits it in every round; *Compose* builds a fresh harness from the library in every round. Both use one analyst per worker. P@1 and P@2 are pass@1 and pass@2 in %, averaged over three test runs. Bold marks the better mode within each regime. *Avg.* averages five benchmarks transductively and four inductively, since GAIA has no inductive run.

		ALFWorld		WebShop		AppWorld		GAIA		SWE-bench Verified		Avg.	
Regime	Mode	P@1	P@2	P@1	P@2	P@1	P@2	P@1	P@2	P@1	P@2	P@1	P@2
Transductive	Refine	88.3	93.9	**58.6**	**67.8**	27.4	41.6	48.0	58.8	50.7	59.4	54.6	64.3
	Compose	**96.6**	**98.0**	51.8	62.7	**40.3**	**54.6**	**49.8**	**63.4**	**54.4**	**61.8**	**58.6**	**68.1**
Inductive	Refine	**93.1**	95.5	52.0	59.2	31.4	45.6	–	–	**53.9**	**63.1**	57.6	65.8
	Compose	92.5	**96.3**	**56.0**	**66.0**	**49.4**	**65.3**	–	–	52.7	61.8	**62.7**	**72.4**

E LIMITATIONS

LiteEvo fits one library per benchmark: its harnesses transfer to unseen tasks of the same benchmark and, on AppWorld, to unseen apps, but a library evolved on other benchmarks did not help an unseen one (Appendix D.4). The procedure, however, carries across benchmarks: the same loop builds a working library for each benchmark from the same neutral start. We therefore see LiteEvo as a step towards harness evolution that is fully automated and generalizes across benchmarks. On AppWorld's test-challenge split, HarnessX's harness scores higher than ours (Appendix D.2).

Each result comes from a single evolution run, so the tables do not show how much the loop's outcome varies from run to run. On GAIA, the agent can still reach answers through copies of the benchmark that the web tools do not block: in round 7, which ran our returned harness, 10 winning episodes reached such a copy before answering. Scoring them as failures would leave that round's pass@2 no lower than 62.9, against 64.6 as measured, a bound that covers only the loop's episodes (Appendix C.4). One evolved component also points the agent at dataset mirror pages.

The librarian mostly adds components, and we did not try more active management of the library. For lack of resources, we did not use a flagship model as the agent or test the hardest agentic benchmarks; tasks outside agentic benchmarks are out of scope. We wrote the meta-agents' instructions once and never optimized them; letting a frontier model design these instructions, packaging LiteEvo as a skill for coding agents such as Claude Code, and combining it with fine-tuning or reinforcement learning are left to future work.