DUETHARNESS: BIDIRECTIONAL POLICY–HARNESS CO-OPTIMIZATION FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The capability of a large language model (LLM) agent depends jointly on its *policy* and its *harness*, the executable runtime that mediates interaction with the environment. Existing work either optimizes one side or updates both, but learning still remains essentially one-sided. Co-optimization is difficult because the policy and harness control separate strategies, yet each update changes the problem faced by the other. In this paper, we cast their interaction as a common-payoff game in which each side is improved against the other's current configuration. The idealized fixed point of this coupling is a mutual best response, a Nash equilibrium of this game. We therefore introduce **DuetHarness**, which implements this coupling by alternating learned updates analogous to approximate best-response steps. With the policy frozen, the editor learns to revise the harness from rollout feedback; with the harness fixed, the policy adapts to it by training against both the current and historical harnesses. Across three agent benchmarks with Qwen3.5-9B, DuetHarness consistently outperforms the corresponding vanilla agent and strong baselines. Ablations show that these gains rely on the alternating co-optimization rather than simply optimizing both sides. Our code is available at `https://anonymous.4open.science/r/DuetHarness_ano-F0CE/`.

1 INTRODUCTION

A large language model (LLM) agent solves tasks through multi-turn interaction with environments (Yao et al., 2023; Liu et al., 2024), mediated by an executable layer around the model commonly referred to as the *harness* (Meng et al., 2026; Zhou et al., 2026). The model, or *policy*, determines how to act, while the harness governs what the policy observes, which actions it can take, how those actions are executed, and how interaction unfolds over time (Yang et al., 2024). The harness is therefore far from an implementation detail, that across eight model backends, the best and worst of six harnesses differ by as much as 23.8 points (Yao et al., 2026). Agent capability is thus better viewed as a property of the joint model–harness configuration than of model weights alone.

Existing approaches to agent training fall into three broad paradigms, as shown in Figure 1. Conventional methods (a) fix the harness and train the policy under a predefined interaction interface (Song et al., 2024; Zhou et al., 2024; Qi et al., 2025; Jin et al., 2025; Wang et al., 2025; Jiang et al., 2026). A second line of work (b) instead freezes the policy and modifies the harness through evolution or fine-tuning (Lee et al., 2026; Lou et al., 2026; Lin et al., 2026; Zhang et al., 2026; Shao et al., 2026). More recent methods (c) modify the policy and harness jointly (Chen et al., 2026c;a;b; Luo et al., 2026). Yet even in these methods, learning typically remains one-sided: the policy learns from experience, whereas harness updates are produced by a fixed proposer that does not itself learn from the outcomes of its edits. This asymmetry raises a natural question: *when the policy and harness shape each other's learning process, can agents benefit from allowing both sides to learn?*

Policy–harness co-optimization poses a two-sided coordination challenge, that each side controls a separate strategy, yet changing either changes the problem faced by the other. Both sides are scored by the same task reward, but model weights and executable programs cannot be optimized jointly. An update that helps one side against a frozen counterpart can also stop working after the other side adapts. We therefore model them as players in a common-payoff game. Each side is improved against the counterpart's current configuration (§3). This mutual-response coupling, whose idealized

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.

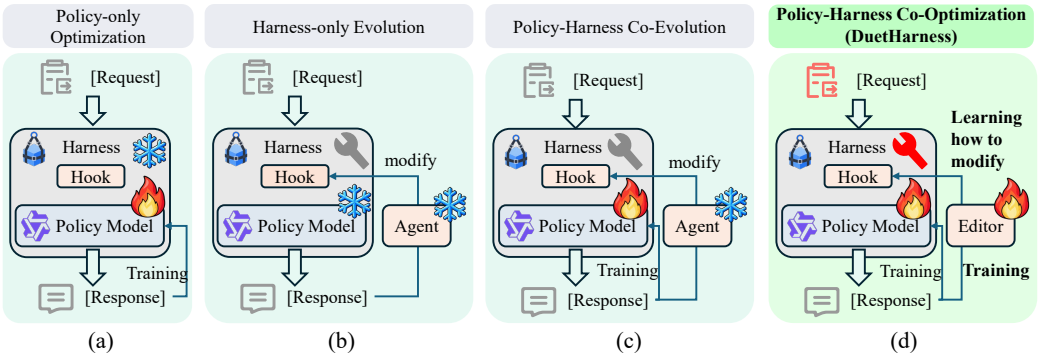

Figure 1: Comparison of policy–harness training paradigms. (a) Policy-only optimization trains the policy under a frozen harness. (b) Harness-only evolution freezes the policy and lets a non-learning agent modify the harness. (c) Policy–harness co-evolution updates both objects, but harness edits still come from a frozen proposer. (d) DuetHarness trains both the policy and a harness editor, so each side learns how to respond to the other.

fixed point is a *mutual best response*, that is a Nash equilibrium of this game, naturally motivates alternating learned responses.

Accordingly, we propose **DuetHarness**, which places the policy and harness editor in an alternating reinforcement-learning loop (Figure 1 (d)). In *Phase-E*, the policy is frozen while the editor learns to improve the current harness from rollout-derived evidence. Candidate patches are evaluated both on the failures that motivated them and on a task-disjoint transfer batch hidden from the editor, encouraging edits that repair observed failures without overfitting to them. The learned editor then produces a persistent patch that is incorporated into the harness. In *Phase-P*, the editor is frozen while the policy is trained with agentic GRPO under the updated harness and a pool of historical harnesses, reducing over-specialization to the latest runtime. Both sides inherit the result of each round, so the editor continually responds to policy drift while the policy adapts to harness drift. Each side is thus updated against the other's current configuration, a mutual-best-response coupling analogous to approximate best response.

We evaluate DuetHarness on three agent benchmarks, WebShop (Yao et al., 2022), ALFWorld (Shridhar et al., 2021), and DBBench (Liu et al., 2024), with both the policy and harness editor initialized from the same Qwen3.5-9B base model (Team, 2026). On all benchmarks, DuetHarness substantially improves over the corresponding vanilla agent and consistently outperforms strong baselines. On WebShop (Yao et al., 2022), for example, starting from the raw base policy and a deliberately minimal harness, ten rounds of alternation raise held-out test success rate from 22.2% to 45.8% (+23.6%; shaped score 0.478 to 0.740), while reducing environment steps per episode from 7.8 to 4.4. Ablation studies further isolate the contribution of DuetHarness's key design choices, that policy-only training (38.6%) and editor-only training (27.4%) both fall short of the full 45.8%, while sequentially training the editor before policy reaches only 40.0%. Additional analysis further demonstrates how the two sides respond to each other over rounds. Our contributions are as follows.

- We formulate policy–harness co-adaptation as a common-payoff game and use mutual best response as a stable target for the two independently controlled components.
- We propose **DuetHarness**, an alternating reinforcement learning framework for policy–harness co-optimization. A learnable harness editor adapts the runtime to the evolving policy, while the policy is trained against both the updated harness and historical harnesses, enabling the two sides to continually adapt to each other.
- Experimental results on three agent benchmarks indicate DuetHarness consistently improves over vanilla agents and strong baselines. Ablations further show that the gains depend on the alternating policy–harness co-optimization.

2 Related Work

Harness Evolution. Recent work treats the agent harness as an optimization target rather than fixed infrastructure. Meta-Harness searches over executable harness code using prior implemen-

tations, execution traces, and evaluation scores (Lee et al., 2026), while AutoHarness synthesizes code-based harnesses through iterative environment feedback (Lou et al., 2026). Agentic Harness Engineering (AHE) further expands the editable space to prompts, tools, middleware, skills, and memory (Lin et al., 2026), and AEVO allows a meta-agent to modify the mechanism that drives future evolution (Zhang et al., 2026). These methods demonstrate the effectiveness of automatically improving model-external runtime structures, but their harness proposers are generally fixed. Harness-R1 goes further by training a dedicated harness engineer with outcome-grounded reinforcement learning, yet keeps the target agent frozen (Shao et al., 2026). Unlike these methods, DuetHarness continually trains the policy in the alternating loop to adapt to the continually updating harness.

Policy Optimization for LLM Agents. A complementary line of work improves the model-side policy through supervised or reinforcement learning from interactive trajectories (Song et al., 2024; Zhou et al., 2024; Qi et al., 2025; Jin et al., 2025; Wang et al., 2025). Such methods enhance planning, tool use, search, and long-horizon decision making, while typically assuming a fixed runtime interface. Harness-1, for example, shows that carefully designed state-externalizing harnesses can substantially improve policy learning by moving routine bookkeeping from the model into the environment (Jiang et al., 2026). However, once policy optimization changes the agent's behavior, a fixed harness cannot adapt to the new failure modes that emerge. DuetHarness therefore treats policy learning and harness adaptation as coupled processes rather than optimizing the policy under a stationary scaffold.

Policy–Harness Co-Evolution. Recent systems have begun to jointly adapt both sides. Co-Harness alternates harness repair with model fine-tuning (Chen et al., 2026c), HarnessForge combines fault-guided harness tailoring with harness-conditioned policy alignment (Chen et al., 2026a), and HarnessX couples symbolic harness adaptation with model reinforcement learning (Chen et al., 2026b). These methods establish the value of co-adapting runtime structure and model behavior. However, the two sides remain asymmetric, where policy parameters are learned from interaction outcomes, whereas harness updates are still generated by a fixed critic, meta-agent, or symbolic optimizer. DuetHarness instead makes both sides learned responders, alternating learning of the harness editor and the policy. Thus, our focus is not only on jointly changing the policy and harness, but on jointly learning how each should improve in response to the other.

3 Problem Formulation

We consider an LLM agent as a coupled system consisting of a model-side *policy* and a model-external *harness*. Let $x \sim \mathcal{D}$ denote a task, π_ϕ a policy parameterized by ϕ, and $h \in \mathcal{H}$ an executable harness that controls context construction, tool interaction, state management, feedback, recovery, and termination. Running π_ϕ inside h produces a trajectory $\tau(\pi_\phi, h, x)$, which receives task reward

$$R(\tau(\pi_\phi, h, x), x) \in [0, 1].$$

Their common payoff is

$$J(\pi_\phi, h) = \mathbb{E}_{x\sim\mathcal{D}}\left[R(\tau(\pi_\phi, h, x), x)\right]. \tag{1}$$

This payoff evaluates a policy–harness pair (π_ϕ, h).

Mutual best response. Maximizing J jointly over π_ϕ and h is intractable, since one side is a differentiable parameter vector and the other an executable program. Rather than pursue the global joint optimum, we therefore settle for a weaker but attainable target, a *mutual best response* pair which neither side can improve J on its own. For a fixed harness, define the ideal policy response as

$$\mathfrak{B}_\pi(h) = \arg\max_{\pi_\phi} J(\pi_\phi, h),$$

while for a fixed policy, define the ideal harness response as

$$\mathfrak{B}_h(\pi_\phi) = \arg\max_{h\in\mathcal{H}} J(\pi_\phi, h).$$

A pair (π_{ϕ^*}, h^*) satisfies mutual best response if

$$\pi_{\phi^*} \in \mathfrak{B}_\pi(h^*), \qquad h^* \in \mathfrak{B}_h(\pi_{\phi^*}). \tag{2}$$

This pair is a pure-strategy Nash equilibrium of a common-payoff game, neither side can improve J unilaterally. It is the idealized fixed point of the coupling, but not necessarily a global joint optimum.

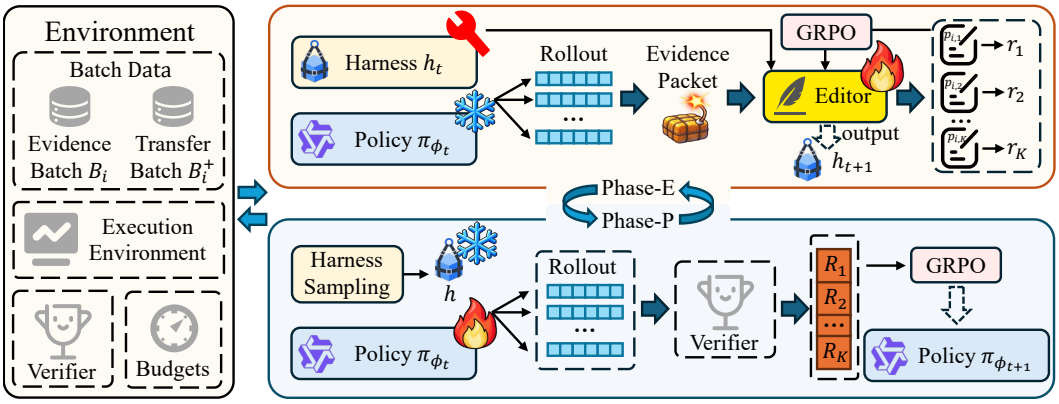

Figure 2: Overview of DuetHarness. Each round alternates two reinforcement-learning phases so that each side learns to respond to the other. Phase-E freezes (π_{ϕ_t}, h_t), compresses rollouts into an evidence packet, and trains the editor with GRPO on K candidate patches scored by paired reward on an evidence batch B_i and a task-disjoint transfer batch B_i^+ hidden from the editor; the updated editor then writes a persistent patch h_{t+1}. Phase-P freezes the editor and trains the policy with GRPO under a harness sampled from h_{t+1} and the historical pool, producing $\pi_{\phi_{t+1}}$.

Why two-sided co-optimization? Single-sided optimization satisfies only one response condition and may leave residual improvement on the frozen side. Starting from (π_{ϕ_t}, h_t), suppose the harness takes an exact best response $h_{t+1} \in \mathfrak{B}_h(\pi_{\phi_t})$, followed by a policy best response $\pi_{\phi_{t+1}} \in \mathfrak{B}_\pi(h_{t+1})$. Then, by definition,

$$J(\pi_{\phi_{t+1}}, h_{t+1}) \geq J(\pi_{\phi_t}, h_{t+1}) \geq J(\pi_{\phi_t}, h_t). \tag{3}$$

Thus, after a harness update, adapting the policy can only preserve or improve J in the idealized setting, with a strict gain unless the policy is already a best response. The argument is symmetric, that one-sided optimization can leave improvement on the frozen side, motivating alternating updates without implying global optimality.

Learning the harness response. Unlike the policy, the harness is an executable program rather than a differentiable parameter vector. We therefore introduce a parameterized harness editor H_θ as a learned approximate solver for $\mathfrak{B}_h$. Given rollout-derived evidence $s(\pi_\phi, h)$ and the current harness h, the editor generates a patch

$$p \sim H_\theta(\cdot \mid s(\pi_\phi, h), h), \qquad h' = \mathrm{Apply}(h, p).$$

The editor itself is not a third participant in Eq. 2; it parameterizes how a harness response is generated, while the deployed object remains h.

In practice, both responses are approximate due to finite rollouts, restricted patch spaces, and finite optimization budgets. DuetHarness therefore alternates learned updates of the form

$$h_{t+1} \approx \mathfrak{B}_h(\pi_{\phi_t}), \qquad \pi_{\phi_{t+1}} \approx \mathfrak{B}_\pi(h_{t+1}),$$

so that the harness continually responds to policy changes, while the policy in turn adapts to the evolving runtime.

4 DuetHarness

DuetHarness instantiates the mutual-response coupling in §3 with a learned harness editor H_θ and an agent policy π_ϕ. Training consists of an editor initialization stage followed by T alternating rounds. Round t begins from (ϕ_t, θ_t, h_t). Phase-E holds the policy and harness fixed, optimizes the editor from rollout evidence, and uses the updated editor to produce h_{t+1}. Phase-P then holds the editor and h_{t+1} fixed while optimizing the policy to $\pi_{\phi_{t+1}}$. The resulting policy, editor, and harness initialize the next round. Figure 2 and Algorithm 1 summarize the procedure.

4.1 AN EDITABLE AGENT HARNESS

We represent a harness h as an ordered collection of executable hooks together with bounded runtime parameters. The runtime exposes editable hook locations before the first decision (`on_episode_start`), before each model call (`on_before_decision`), before an action reaches the environment (`on_before_action`), after environment feedback (`on_after_feedback`), and, on benchmarks whose episodes end with an irreversible submission, before it (`on_before_commit`). A hook reads policy-visible state (the current observation, the pending action, a compact interaction history, and the remaining budget) together with a scratch space shared across hooks that serves as harness memory. Depending on its location it may inject guidance, rewrite the context the policy is about to read, normalize the pending action, or refuse it and re-ask. Appendix A gives the full interface, the effects permitted at each location, the parameter ranges, and the initial harness h_0.

The editor changes h by generating a structured patch p built from four operations: `add_hook`, `replace_hook`, `delete_hook`, and `set_param`. We write $\mathcal{V} : P \to \{0, 1\}$ for the predicate that a patch conforms to the schema and passes the checks required before evaluation: static validation, sandboxed compilation of every hook body, parameter-range checks, and a test that the patch changes the harness. A patch with $\mathcal{V}(p) = 0$ is never installed and never run. Appendix B specifies the schema and each check.

4.2 EDITOR COLD START VIA SUPERVISED FINE-TUNING

Before the first alternating round, we initialize the editor using patch demonstrations. We roll out the base policy π_{ϕ_0} under a harness h and compress each batch of episodes into an evidence packet s_i, that is a policy-visible summary of how the episodes ended, aggregate trajectory statistics such as invalid actions, repetition, and stagnation, which installed hooks fired, and a few representative action–observation excerpts. Appendix C gives the exact fields, the excerpt selection rule, and the compression budget. Here h is the harness that produced the rollouts, mostly the default runtime h_0, together with a few deliberately degraded variants, since a teacher shown h_0 alone has nothing worth deleting or retuning and only ever writes new hooks. Conditioned on (s_i, h), a teacher editor samples candidate patches, each scored by the same validator and paired evaluation used in Phase-E. Appendix D gives the prompt and response protocol, and Appendix E the degraded variants and the filtering procedure. Let $\mathcal{C}_i : P \to \{0, 1\}$ indicate that the evaluations on both paired batches finish. We retain candidates with non-negative paired reward and form the SFT dataset

$$\mathcal{D}_{\mathrm{SFT}} = \big\{(s_i, h, p) : p \sim H_{\mathrm{teach}}(\cdot \mid s_i, h),\ \mathcal{V}(p) = \mathcal{C}_i(p) = 1,\ r_i(p) \geq 0\big\}. \tag{4}$$

Here r_i is the paired reward of Eq. 7. The non-negative threshold keeps patches that improve the paired reward or leave it unchanged, discarding only harmful ones. We train the editor with the supervised objective

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(s_i,h,p)\sim\mathcal{D}_{\mathrm{SFT}}} \log H_\theta(p \mid s_i, h). \tag{5}$$

The resulting parameters define θ_0. The policy is not cold-started that ϕ_0 is the unmodified base-model weights, and its first update occurs in Phase-P.

4.3 ALTERNATING POLICY–HARNESS REINFORCEMENT LEARNING

Starting from the policy, editor, and harness $(\pi_{\phi_0}, \theta_0, h_0)$, training proceeds in T alternating rounds. At round t, we draw a task block $\mathcal{B}_t \subset \mathcal{T}$ and use tasks from this block in both phases.

Phase-E: editor update. We fix (π_{ϕ_t}, h_t) and collect rollouts $\tau(\pi_{\phi_t}, h_t, x)$ on evidence batches $B_i \subset \mathcal{B}_t$. Each batch of trajectories is compressed into a evidence packet $s_i = \mathrm{Compress}(\{\tau(\pi_{\phi_t}, h_t, x) : x \in B_i\})$ of the same form as in §4.2. We pair B_i with a task-disjoint transfer batch B_i^+ whose trajectories are not included in s_i. Appendix F gives the batch construction, the shared baseline, and the update bundling. Conditioned on (s_i, h_t), the editor samples K candidate patches, each of which is independently applied to h_t. For a task batch B, define

$$J_B(\pi_\phi, h) = \frac{1}{|B|} \sum_{x\in B} R(\tau(\pi_\phi, h, x), x), \qquad \Delta_B(p) = J_B(\pi_{\phi_t}, \mathrm{Apply}(h_t, p)) - J_B(\pi_{\phi_t}, h_t). \tag{6}$$

We score a candidate patch as

$$r_i(p) = \begin{cases} \alpha\Delta_{B_i}(p) + (1-\alpha)\Delta_{B_i^+}(p), & \text{if } \mathcal{V}(p) = \mathcal{C}_i(p) = 1, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

The first term measures the change on the tasks that produced s_i, and the second measures the change on the disjoint transfer batch. We normalize the rewards of the K candidates sampled for each packet and update θ_t with GRPO (Shao et al., 2024). Repeating this update over the round's packets yields θ_{t+1}.

Harness update. After the editor updates, we aggregate the round's packets into $\bar{s}_t^{\text{train}}$ and sample candidate patches $p_t \sim H_{\theta_{t+1}}(\cdot \mid \bar{s}_t^{\text{train}}, h_t)$. Because this harness is inherited by every later round, a candidate must clear one gate beyond $\mathcal{V}$: it is installed on a copy and run for a few episodes, and only a crash disqualifies it. We take the first candidate that satisfies $\mathcal{V}(p_t) = 1$ and passes this smoke test, and apply it to the current harness,

$$h_{t+1} = \text{Apply}(h_t, p_t), \tag{8}$$

setting $h_{t+1} = h_t$ if a fixed generation budget is exhausted without one, in which case the round has still trained θ. The resulting h_{t+1} is used throughout Phase-P and is passed to the next round.

Phase-P: policy update. We fix (θ_{t+1}, h_{t+1}) and train the policy with task-native verifier rewards. Let the set of distinct harnesses available before the current update be

$$\Omega_t = \{h_0, h_1, \ldots, h_t\}. \tag{9}$$

Distinctness here is behavioral, so a round that installed nothing does not enlarge the pool. For each training unit with task $x \in \mathcal{B}_t$, we set $h = h_{t+1}$ with probability ρ and otherwise sample h uniformly from Ω_t. The draw is weighted toward h_{t+1} because that is the harness the policy will be evaluated under, while the past harnesses keep it from over-specializing to a runtime the editor will revise next round again. We generate K trajectories for the fixed pair (x, h), normalize their verifier rewards within the group, and update ϕ_t with GRPO to obtain ϕ_{t+1}. Keeping the pair fixed within a unit is what lets that normalization compare trajectories under identical conditions, and credit is assigned per episode rather than per turn, since the verifier scores a submission and not an intermediate action. We then set $\Omega_{t+1} = \Omega_t \cup \{h_{t+1}\}$ and continue with $(\phi_{t+1}, \theta_{t+1}, h_{t+1})$.

Algorithm 1 DuetHarness training procedure

Require: base policy π_{ϕ_0}, SFT-initialized editor θ_0, initial harness h_0
Require: training set $\mathcal{T}$, rounds T, group size K
Require: paired-reward weight α, current-harness probability ρ

$\Omega_0 \leftarrow \{h_0\}$ *// history of deployed harnesses*
for $t = 0, \ldots, T-1$ **do**
  sample $\mathcal{B}_t \subset \mathcal{T}$
  for each $B_i \subset \mathcal{B}_t$ **do** *// Phase-E; (π_{ϕ_t}, h_t) frozen*
    sample $\tau_i(x) \sim \tau(\pi_{\phi_t}, h_t, x)$ for all $x \in B_i$ *// rollouts*
    $s_i \leftarrow \text{Compress}(\{\tau_i(x)\})$ *// evidence packet*
    sample $B_i^+ \subset \mathcal{B}_t$ with $B_i^+ \cap B_i = \emptyset$ *// hidden from the editor*
    $p_{i,1}, \ldots, p_{i,K} \sim H_{\theta_t}(\cdot \mid s_i, h_t)$
    compute $r_i(p_{i,k})$ by Eq. 7 *// paired reward*
  end for
  $\theta_{t+1} \leftarrow \text{GRPO}(\theta_t; \{r_i(p_{i,k})\})$
  $\bar{s}_t^{\text{train}} \leftarrow \text{Aggregate}(\{s_i\})$ *// round-level packet*
  $p_t \sim H_{\theta_{t+1}}(\cdot \mid \bar{s}_t^{\text{train}}, h_t)$ *// persistent patch*
  $h_{t+1} \leftarrow \text{Apply}(h_t, p_t)$ if $\mathcal{V}(p_t) = 1$ and the smoke test passes, else h_t
  for each unit **do** *// Phase-P; θ_{t+1}, h_{t+1} frozen*
    $x \sim \text{Unif}(\mathcal{B}_t)$
    $h \leftarrow h_{t+1}$ w.p. ρ, else $h \sim \text{Unif}(\Omega_t)$ *// mix current and historical*
    $\tau_1, \ldots, \tau_K \sim \tau(\pi_{\phi_t}, h, x)$
  end for
  $\phi_{t+1} \leftarrow \text{GRPO}(\phi_t; \{R(\tau_k, x)\})$
  $\Omega_{t+1} \leftarrow \Omega_t \cup \{h_{t+1}\}$
end for
return (ϕ_T, θ_T, h_T)

Table 1: Main results (%) of DuetHarness on WebShop, ALFWorld, and DBBench. The best result is **highlighted in sand** and the second-best in **light blue**. Δ is the within-stack gain over each method's own vanilla. †Reflexion is evaluated with a two-trial success@2 protocol and is excluded from ranking. ‡Policy is a stronger SFT Qwen3.5-9B rather than the vanilla one.

Method	WebShop		ALFWorld (unseen/seen)							DBBench
	Score	Succ.	Pick	Look	Clean	Heat	Cool	Pick2	All	Succ.
Baselines										
Qwen3.5-9B	65.8	32.0	66.7 / 80.0	33.3 / 46.2	9.7 / 11.1	8.7 / 18.8	9.5 / 0.0	64.7 / 75.0	29.9 / 41.4	61.7
SFT Qwen3.5-9B	68.8	38.0	**95.8** / **97.1**	77.8 / **84.6**	35.5 / 29.6	30.4 / 25.0	33.3 / 16.0	76.5 / 91.7	56.0 / 59.3	59.3
Prompt-based agentic methods										
ReAct	66.8	37.4	83.3 / 88.6	50.0 / 61.5	19.4 / 11.1	21.7 / 25.0	19.0 / 0.0	82.4 / 66.7	43.3 / 44.3	60.0
Self-Refine	61.7	29.0	66.7 / 82.9	50.0 / 23.1	9.7 / 0.0	30.4 / 25.0	14.3 / 4.0	**94.1** / 66.7	40.3 / 37.9	58.3
Reflexion†	61.2	43.6	91.7 / 97.1	77.8 / 61.5	29.0 / 22.2	30.4 / 31.2	14.3 / 20.0	94.1 / 95.8	53.0 / 57.9	67.7
Frontier models as harness editors (and with a stronger fine-tuned policy‡)										
Qwen3.5-397B	66.5	37.2	70.8 / 82.9	50.0 / 53.8	6.5 / 25.9	8.7 / 12.5	4.8 / 0.0	64.7 / 62.5	31.3 / 42.9	60.3
Qwen3.5-397B‡	66.3	38.6	**95.8** / **97.1**	77.8 / **84.6**	29.0 / 25.9	26.1 / 25.0	23.8 / 12.0	88.2 / 83.3	53.7 / 56.4	59.3
DeepSeek-V4-Pro	66.2	36.8	70.8 / 85.7	66.7 / 38.5	9.7 / 7.4	0.0 / 18.8	4.8 / 0.0	64.7 / 75.0	32.8 / 41.4	62.3
DeepSeek-V4-Pro‡	67.2	38.0	**95.8** / **97.1**	**100** / **84.6**	35.5 / 29.6	17.4 / 25.0	33.3 / 8.0	88.2 / 87.5	58.2 / 57.1	63.3
Kimi-K2.6	66.0	38.2	70.8 / 88.6	50.0 / 53.8	6.5 / 14.8	21.7 / 12.5	9.5 / 0.0	58.8 / 62.5	33.6 / 42.1	61.7
Kimi-K2.6‡	66.7	39.8	**95.8** / **97.1**	77.8 / **84.6**	35.5 / 29.6	17.4 / 25.0	28.6 / 8.0	47.1 / 91.7	49.3 / 57.9	61.3
Learned harness editors										
Harness-R1	69.0	**41.0**	**91.7** / 82.9	55.6 / 69.2	45.2 / 29.6	52.2 / 18.8	23.8 / 4.0	41.2 / 58.3	52.2 / 45.7	**65.0**
Δ	+3.2	+9.0	+25.0/+2.9	+22.2/+23.1	+35.5/+18.5	+43.5/+0.0	+14.3/+4.0	−23.5/−16.7	+22.4/+4.3	+3.3
Harness-R1‡	**69.6**	40.0	87.5 / **100**	**88.9** / **76.9**	**71.0** / **66.7**	**60.9** / **43.8**	**90.5** / 24.0	70.6 / **95.8**	**77.6** / **70.7**	64.7
Δ‡	+0.8	+2.0	−8.3/+2.9	+11.1/−7.7	+35.5/+37.1	+30.5/+18.8	+57.2/+8.0	−5.9/+4.1	+21.6/+11.4	+5.4
Ours										
Qwen3.5-9B + h_0	47.8	22.2	54.2 / 74.3	61.1 / 69.2	25.8 / 25.9	17.4 / 25.0	38.1 / **28.0**	70.6 / 58.3	41.8 / 47.9	59.0
DuetHarness	**74.0**	**45.8**	**95.8** / **100**	**100** / 69.2	**90.3** / **96.3**	**91.3** / **62.5**	**95.2** / **92.0**	**100** / **100**	**94.8** / **90.7**	**66.0**
Δ	+26.2	+23.6	+41.6/+25.7	+38.9/+0.0	+64.5/+70.4	+73.9/+37.5	+57.1/+64.0	+29.4/+41.7	+53.0/+42.8	+7.0

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Datasets and evaluation. We evaluate DuetHarness on three agentic benchmarks, WebShop (Yao et al., 2022), ALFWorld (Shridhar et al., 2021), and DBBench (Liu et al., 2024). WebShop is an online shop where the agent finds and buys a product matching a natural-language instruction. ALFWorld is a text household environment whose tasks require navigation, object manipulation, and multi-step execution. DBBench is a database environment whose tasks require querying and updating. We report success rate on all benchmarks. On WebShop we also report the official score, which gives partial credit for product type, attributes, options, and price. On ALFWorld we also report success by task family on the official unseen and seen splits, where All is the micro-average.

Baselines. We adopt baselines of Harness-R1 (Shao et al., 2026), which evaluate Qwen3.5-9B under that work's default harness, including prompt-based agentic methods ReAct (Yao et al., 2023), Self-Refine (Madaan et al., 2023), and Reflexion (Shinn et al., 2023); frontier models prompted as harness editors, including Qwen3.5-397B (Team, 2026), DeepSeek-V4-Pro (Xu et al., 2026), and Kimi-K2.6, each run with a vanilla policy and with a stronger fine-tuned Qwen3.5-9B policy; and Harness-R1. The h_0 below restates that default harness as guidance hooks defined in Appendix A.

Implementation details. The policy and the editor start from the same Qwen3.5-9B weights (Team, 2026) with independent rank-32 LoRA adapters. All benchmarks follow their official splits, with the official training pool partitioned into a subset for editor cold-start SFT and a disjoint subset for online RL training. The checkpoint is a policy–harness pair (π_ϕ, h), which we select on held-out validation tasks during training and freeze for testing. Alternating training runs for at most $T = 10$ rounds with $K = 8$ candidate patches per packet, $\alpha = 0.5$, and $\rho = 0.8$. Appendices G and H give the full splits and hyperparameters.

5.2 MAIN RESULTS

Table 1 reports held-out test performance. We highlight three findings:

DuetHarness leads on all benchmarks, independent of starting-harness strength. On WebShop it reaches 74.0 score and 45.8% success rate, above the strongest prior entries of 69.6 and 41.0%; and on ALFWorld 94.8% / 90.7% unseen/seen success, above the strongest prior Harness-R1[‡] of 77.6% / 70.7%, while 66.0% success on DBBench, only 1.0% above the second-best Harness-R1 of 65.0%, yet $+7.0\%$ over its own vanilla. Although h_0 already leads the default harness on ALFWorld (41.8% / 47.9% vs. 29.9% / 41.4%), it is the weakest starting point on WebShop (22.2% vs. 32.0%) and also trails the default on DBBench (59.0% vs. 61.7%), so the final ranking is not determined by the initial harness strength.

Alternation gains more than editing a harness against a frozen policy. Relative to its own vanilla, DuetHarness gains $+23.6\%$ / $+26.2$ on WebShop success/score, $+53.0\%$ / $+42.8\%$ on ALFWorld unseen/seen, and $+7.0\%$ on DBBench. One-sided harness editing gains much less. Harness-R1 improves $+9.0\%$ / $+3.2$, $+22.4\%$ / $+4.3\%$, and $+3.3\%$ to the untrained vanilla, and only $+2.0\%$ / $+0.8$, $+21.6\%$ / $+11.4\%$, and $+5.4\%$ to the SFT policy. A frozen policy cannot improve, so a better harness only assists the same weaker agent. Prompted frontier editors stay near those vanillas and can even regress. These models are prompted rather than trained as harness editors, so they often emit illegal patches that the validator rejects. DuetHarness lifts above limits by training the editor and the policy alternately.

DuetHarness sharply raises success on the long-horizon families that start hardest. On ALFWorld, the long-horizon families Clean, Heat, and Cool start hardest under h_0, at only 25.8% / 25.9%, 17.4% / 25.0%, and 38.1% / 28.0%. They need a locate–transform–place chain, and h_0 only hints after an illegal command has already spent a step. DuetHarness lifts them to 90.3% / 96.3%, 91.3% / 62.5%, and 95.2% / 92.0%. Alternation lets the editor write hooks that send the agent to the right place before a transform and refuse illegal commands, while Phase-P trains the policy to follow them. Other methods remain weak here, since a frozen policy cannot learn the same chain.

5.3 Ablation Studies

To determine whether DuetHarness benefits from genuine two-sided co-optimization, we conduct ablation studies on WebShop about two aspects of DuetHarness: 1) which side learns and whether the two sides alternate, and 2) which mechanisms stabilize the alternating loop. We therefore compare against policy-only, editor-only, sequential (E→P), and frozen-editor variants, and separately remove either term of the paired editor reward. All variants use the same experimental settings as the main experiments. Table 2 reports the average of three repeated experimental runs, and Appendix I defines each variant precisely.

Table 2: Ablation results (%) on the WebShop held-out test set.

Variant	Score	Succ.
Qwen3.5-9B + h_0	47.8	22.2
Which sides learn, and whether they alternate		
Policy-only ($h \equiv h_0$)	70.1	38.6
Editor-only ($\pi_\phi \equiv \pi_{\phi_0}$)	55.8	27.4
Sequential (E→P)	70.5	40.0
Frozen editor (patches from θ_0)	56.8	18.6
Mechanisms for two-sided optimization		
w/o transfer batch ($\alpha = 1$)	71.9	43.6
w/o evidence batch ($\alpha = 0$)	69.7	40.2
DuetHarness	**74.0**	**45.8**

Two-sided learning and alternation are both necessary. On WebShop, **policy-only** reaches 38.6% success rate, above the untrained 22.2% baseline but below the full DuetHarness' 45.8%, and **editor-only** similarly reaches 27.4%, indicating neither side is sufficient on its own. **Sequential (E→P)** trains learners of both sides, yet runs all editor updates against the initial policy before any policy training, and reaches 40.0% success at 70.5 score, 5.8% and 3.5 below DuetHarness. A harness fitted to the initial policy can still support later policy training, but alternation recovers additional success and shaped return after the policy has moved. **Frozen editor** keeps the alternating schedule but never updates the editor, dropping to 18.6%, below even the untrained 22.2% and far below policy-only at 38.6%. This demonstrates a fixed editor that still writes patches can make the runtime worse than leaving h_0 unchanged, so the editor has to keep learning as the policy moves.

Transfer-aware rewards stabilize co-optimization. On WebShop, removing the **transfer batch** ($\alpha = 1$) and the **evidence batch** ($\alpha = 0$) reduces success from 45.8% to 43.6% and 40.2%, respectively. The two reward ablations expose complementary failure modes, in that the former encourages

overfitting to the observed failures, whereas the latter removes direct credit for repairing them. Together, these results support balancing targeted repair with cross-task transfer.

5.4 Analysis

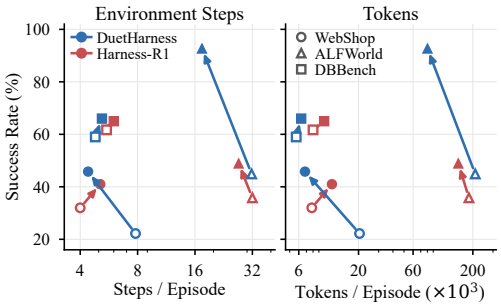

Figure 3: Success rate vs. cost.

DuetHarness improves success with lower cost. Figure 3 shows the trade-off relative to each method's vanilla baseline, with full numbers in Table 10. DuetHarness is more successful and cheaper than Harness-R1 on all three benchmarks, cutting steps/tokens on WebShop ($-44\%/-67\%$) and ALFWorld ($-45\%/-62\%$), with only a small cost increase on DBBench ($+8\%/+11\%$). Harness-R1 yields smaller savings on ALFWorld and higher costs on WebShop and DBBench. The key is that DuetHarness learns harnesses that reduce wasted search and then trains the policy to exploit them, improving both success and efficiency. The following three analyses expand this account, tracing which failures the harness removes, how the policy comes to follow the new hooks, and how little gain remains once either side is held fixed.

The learned harness removes procedural failures, leaving instruction-matching errors. Table 11 in Appendix J shows that DuetHarness nearly eliminates abandoned episodes. On WebShop, non-submissions fall from 128 under h_0 to 2. On ALFWorld, they drop from 151 to 20, while illegal commands fall from 5.4 to 1.7 per episode. WebShop failures therefore shift from operating the environment to matching the instruction. Among completed purchases, missing required options fall from 49.2% to 30.1%, while attribute and price errors rise slightly because many more episodes now reach checkout.

DuetHarness works through policy–harness co-adaptation. Figure 4 tracks validation performance after each Phase-E and Phase-P update alongside the learned hooks. On WebShop, 8 of 10 Phase-E updates reduce success, while 9 of 10 Phase-P updates recover it, showing that harness changes become useful only after the policy adapts. The editor also evolves the harness itself, replacing the generic `unrecognised` hint in h_0 with guards that block or rewrite wasted actions and revising earlier hooks over time. ALFWorld shows the same pattern more mildly, with only 2 of 9 Phase-E updates reducing success. Appendix K provides the full traces and hook-level analysis.

DuetHarness is an empirical approximate best-response procedure. As formalized in §3, the alternating updates are approximate best responses and therefore need not reach a Nash equilibrium of Eq. 2. To check the pair retained for testing, we freeze one side at a time on WebShop and continue the same operator for three rounds. Holding the harness fixed, further policy training moves the score only from 0.727 to 0.739. Holding the policy fixed, further harness edits yield no positive residual, and the score falls to 0.615. The remaining unilateral gains are negligible, so we regard the pair as an empirical Nash under this operator. Appendix L reports the protocol and the per-round results.

6 Conclusion

We introduced **DuetHarness**, which casts policy–harness co-optimization as a common-payoff game and alternates updates toward mutual best response rather than a joint optimum. Phase-E freezes the policy while an editor revises the runtime from rollout evidence, scoring each patch on the motivating failures and a hidden transfer batch; Phase-P freezes the editor while the policy trains under the updated harness and a historical pool. On three agentic benchmarks with Qwen3.5-9B, DuetHarness outperforms the corresponding vanilla agent and strong baselines, and finishes episodes in fewer steps and tokens. Ablations attribute the gains to alternation. Training either side alone, or the editor before the policy, falls short, and freezing the editor can leave the runtime worse than it started.

Limitations. We currently evaluate a single 9B backbone on three benchmarks and will extend to more backbones, benchmarks, and multi-seed runs. The editor is confined to a fixed hook schema, and the updates are approximate best responses rather than a Nash solver.

AI USE STATEMENT

Large language models are a core component of the proposed method. Both the policy and the harness editor are initialized from the same Qwen3.5-9B backbone and trained with reinforcement learning, as described in §4. In addition, we used generative AI tools to assist with literature organization and with drafting and editing the manuscript. All AI-assisted text was reviewed, verified against the cited sources, and edited by the authors. We take responsibility for the final content of this work.

ETHICS STATEMENT

In this paper, we propose DuetHarness, a method that co-optimizes the policy and harness of LLM agents to improve their task success and interaction efficiency, thereby providing a more reliable training paradigm for the community. All experiments are conducted on existing benchmarks (WebShop, ALFWorld and DBBench) and do not involve human subjects, personal data, or real-world deployment. Outputs of DuetHarness may occasionally contain inappropriate content inherited from the base model, and such outputs do not reflect the authors' views. We strictly adhere to the ICLR ethical research standards and applicable laws. To the best of our knowledge, this work complies with the General Ethical Principles.

REPRODUCIBILITY STATEMENT

We follow the ICLR reproducibility standards and ensure the reproducibility of our work. All datasets used for training, inference, and evaluation are publicly available. Detailed experimental settings, including data splits, hyperparameters, and implementation steps, are documented in the paper and Appendix, with the training procedure specified in Algorithm 1. Furthermore, we will release our code and checkpoints upon acceptance, enabling other researchers and practitioners to easily reproduce and extend our results.

REFERENCES

Mingju Chen, Can Lv, Guibin Zhang, Heng Chang, and Shiji Zhou. Harnessforge: Joint harness and policy evolution for adaptive agent systems. *arXiv preprint arXiv:2606.01779*, 2026a.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, et al. Harnessx: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026b.

Zhengyu Chen, Teng Xiao, Huaisheng Zhu, Yige Yuan, Luan Zhang, and Jingang Wang. Co-harness: Co-evolving harnesses and model weights for llm agents. *arXiv preprint arXiv:2607.22688*, 2026c.

Pengcheng Jiang, Zhiyi Shi, Kelly Hong, Xueqiang Xu, Jiashuo Sun, Jimeng Sun, Hammad Bashir, and Jiawei Han. Harness-1: Reinforcement learning for search agents with state-externalizing harnesses. *arXiv preprint arXiv:2606.02373*, 2026.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations (ICLR)*, volume 2024, pp. 52989–53046, 2024.

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. *arXiv preprint arXiv:2603.03329*, 2026.

Haochen Luo, Yi Huang, Sichun Luo, Fengyuan Liu, Lei Li, Zefa Hu, Junlan Feng, and Qi Liu. Harness-aware self-evolving: Co-evolving model weights, harness, and task solutions. *arXiv preprint arXiv:2607.03935*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in neural information processing systems (NeurIPS)*, volume 36, pp. 46534–46594, 2023.

Qianyu Meng, Yanan Wang, Liyi Chen, Yihang Li, Wei Wu, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, et al. Agent harness for large language model agents: A survey. 2026.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations (ICLR)*, volume 2025, pp. 79791–79821, 2025.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in neural information processing systems (NeurIPS)*, volume 36, pp. 8634–8652, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021. URL https://arxiv.org/abs/2010.03768.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7584–7600, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 50528–50652, 2024.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 20744–20757, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, et al. Harness-bench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026.

Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, et al. Harnessing agentic evolution. *arXiv preprint arXiv:2605.13821*, 2026.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62178–62209. PMLR, 21–27 Jul 2024.

A Harness Runtime and Hook Interface

All benchmarks share this runtime, including the episode loop, the hooks, the sandbox, and the patch schema. They differ only in a small fixed *adapter* that says how an action is written and what an observation looks like. The editor cannot change the adapter.

Episode lifecycle. Before the first decision, `on_episode_start` fires and whatever it returns is added to the system message for the rest of the episode. Each turn then fires `on_before_decision`, samples an action from the model, fires `on_before_action` (plus `on_before_commit` if the action submits), runs the action, and fires `on_after_feedback` to set up what the model reads next. The episode stops when the environment ends it, the step budget runs out, or the environment ignores too many actions in a row.

Hooks. A hook is a small function, `hook(ctx, nb)`. The first argument holds what the model itself could see at that moment, including the instruction, the action it just proposed, the current observation, a short history of what it has done and how the environment reacted, the remaining budget, and the current parameters. The second is a scratch dictionary shared by every hook in the episode. It is the harness's memory, and a hook can use it to recall information on a later turn.

A hook returns either nothing or one effect its location allows (Table 3). If it returns something we do not recognise, raises an error, or runs too long, we drop the result, record a failure, and carry on, so a bad patch makes the harness worse without crashing the run. Budgets are the one thing a hook cannot touch: they move only through the range-checked `set_param`.

Tunable parameters. `set_param` retunes the five parameters in Table 4 within per-benchmark ranges, above which sit ceilings no patch can lift: 30 steps on WebShop, 50 on ALFWorld, 20 on DBBench, and 8 refusals on all three. ALFWorld gets more room because its episodes are longer and because every observation there ends with the list of commands the simulator accepts, which `max_obs_chars` must not cut off. DBBench stays tighter because a successful episode is a short inspect–query–commit loop, so a larger step budget mostly buys more retries of the same statement.

Execution sandbox. We check and compile every hook body under a restricted subset of Python: one entry function plus a few helpers, no imports, no file or network access, no reflection, and no unbounded loops. A hook gets basic maths, regular expressions, and a string-similarity helper. Source length and per-call runtime are capped and nothing random is available, so the same harness over the same batch gives the same trajectories.

Table 3: Where hooks can be installed, and what each one is allowed to return. `inject_guidance` adds text for the model to read; `transform_context` swaps in a different observation; `normalize_action` rewrites the action the model chose; `block_and_reprompt` refuses it and asks again without spending an environment step. WebShop and DBBench have all five locations, ALFWorld the first four.

Hook	When it runs	Permitted effects
`on_episode_start`	Once, before the first decision	`inject_guidance`
`on_before_decision`	Before each model call	`inject_guidance`, `transform_context`
`on_before_action`	Before environment execution	`normalize_action`, `block_and_reprompt`
`on_after_feedback`	After non-terminal feedback	`inject_guidance`, `transform_context`
`on_before_commit`	Before an irreversible submission	`normalize_action`, `block_and_reprompt`

Table 4: The five tunable parameters, shown as range followed by default. Setting the streak guard to zero turns it off.

Parameter	WebShop	ALFWorld	DBBench	Meaning
`max_steps`	$[4, 30]$, 15	$[4, 50]$, 50	$[4, 20]$, 20	Environment steps per episode
`max_obs_chars`	$[200, 8000]$, 2000	$[2400, 12000]$, 4000	$[200, 8000]$, 2000	Observation truncation length
`max_history_turns`	$[1, 30]$, 30	$[1, 50]$, 50	$[1, 20]$, 20	Turns replayed into the prompt
`max_blocks`	$[0, 8]$, 3	$[0, 8]$, 3	$[0, 8]$, 3	Refusals a harness may spend
`max_unrecognised_streak`	$[0, 10]$, 3	$[0, 10]$, 3	$[0, 10]$, 3	Consecutive ignored actions before abandonment

Information boundary. We build the hook's context field by field, so none of the grader's data reaches a hook. WebShop never exposes the target product, required options, or parsed price limit. ALFWorld never exposes the expert walkthrough or goal parameters. DBBench never exposes the gold SQL, gold answer, or table hash. No benchmark exposes a task id or a per-step reward. Some of the same information is already in the instruction the model reads, including the price limit in words, the object and where it belongs, and the columns a question names. A hook may parse that text. We also reject any WebShop hook whose source contains a product id, since a hook like that has memorised one answer.

The initial harness h_0. All three benchmarks start with the same three guidance-only hooks, under identifiers shared across benchmarks so that an ablation deleting one deletes the same idea on any of them. The three hooks are a strategy note at `on_episode_start`, a nudge at `on_before_decision` when the budget runs low or the observation stops changing, and a note at `on_after_feedback` when the environment ignores an action. Each helps and each has obvious gaps. WebShop's never checks the product options or the price. ALFWorld's never tracks which places have been searched. DBBench's never checks that a query was run before commit. The three are kept roughly equally strong. Every h_0 hook is written in the editor's own sandboxed subset.

B Patch Schema and Legality Checks

Schema. The editor produces a patch as a JSON object: an optional description, plus a list of one to eight actions applied in order. Table 5 lists the four kinds of action. All benchmarks use the same schema.

Table 5: The four kinds of action a patch can contain, and the fields each one takes.

Operation	Fields
`add_hook`	hook location, hook body, optional identifier
`replace_hook`	identifier of an installed hook, new body, optional new location
`delete_hook`	identifier of an installed hook
`set_param`	parameter name, value

The predicate $\mathcal{V}$. All of these checks run before we execute a single episode. A patch is legal when:

1. the JSON is well formed and carries no more than eight actions;
2. every action is one we recognise and one the current configuration allows;
3. every hook body targets a real hook location and compiles in the sandbox of Appendix A;
4. every identifier it names refers to a hook that exists;
5. every `set_param` value is in range; and
6. the patched harness actually behaves differently from the one it started from.

The last check is there because a patch can be legal action by action and still do nothing, for instance by setting a parameter to the value it already has.

If any action fails, we throw the whole patch away, legal actions included. It scores zero and is never installed.

C Evidence Packet

A packet s_i is a short summary of one batch of episodes run under one harness. We cannot show the editor the raw trajectories: they are far too long, and they live on an object that also holds the grader's fields. So we build the packet by copying in only the fields we name.

What the editor cannot see. Reward, success, and the grader's breakdown are not in the packet. They are also not used to build it. That second part matters: if we picked the excerpts by looking at which episodes failed, the choice of excerpts would tell the editor the answer as clearly as printing the reward would. This costs us something real. An episode that confidently bought the wrong product looks fine from the inside, and the packet cannot flag it. Whether a patch fixes episodes like that is decided by the paired reward of Eq. 7, which reruns both batches under the patched harness.

Contents. The packet says which harness produced the batch and counts how the episodes ended, then reports three things: signals averaged over the batch, what each installed hook did, and a handful of excerpts. All benchmarks report turns, environment steps, and how many actions were ignored, refused, or repeated. Each adapter adds its own, such as searches and clicks on WebShop, or moves, pickups, and the commands that burn a step without changing the room on ALFWorld. For every hook we report how often it fired and how often it failed, which is what the editor needs in order to rewrite or delete a hook it installed earlier.

Compression. We pick excerpts round-robin across the ways episodes ended, so one common failure cannot crowd out the rest, and within each group we rank them by a score built from the same signals the model could see, weighting hardest the episodes that never got far enough to finish the task. A packet holds at most eight excerpts of twelve turns each and stays under about 6,000 tokens. Observations get different amounts of space depending on their kind, so a WebShop product page survives nearly intact while a page of search results is cut short. We replace product ids with per-packet aliases like `[PRODUCT_1]`, which still shows the editor when the model kept going back to the same product but leaves nothing it could hard-code.

Aggregate packet. A Phase-E round produces one packet per evidence batch. The persistent-patch step of §4.3 works from the pool of them, $\bar{s}_t^{\text{train}}$. It has the same layout, so the editor sees one format throughout; the counts are pooled over the whole round, and the excerpts are drawn round-robin across the batches.

D Editor Prompt and Response Protocol

One prompt is shared verbatim by demonstration collection, online Phase-E sampling, and evaluation. It is generated from the definitions it constrains (the hook table, the sandbox limits, the patch schema, and the benchmark's parameter ranges). We reproduce the WebShop system prompt in full below, then the ALFWorld and DBBench passages that differ.

D.1 Editor system prompt

Editor System Prompt (WebShop)

You are the harness editor for a text-based shopping agent.

A *harness* is the runtime wrapped around a fixed language-model policy: it decides what the policy reads before each decision, may rewrite or refuse the action the policy chose, and controls the step and context budgets. The policy itself is frozen and will not be retrained. Your only lever is the harness, and your job is to make the same policy shop better by editing it.

You will be shown evidence from a batch of episodes run under the current harness. Read it for the *pattern*—what goes wrong repeatedly, and at which point in an episode—then emit one patch that addresses it. Your patch is scored by rerunning that batch, and a second batch of tasks you were never shown, under the patched harness. Only a change that helps on both is worth anything.

The store

The policy acts with `search[query]` and `click[button text]`. Only the search page has a search bar. On a results page a product is opened by its id. `Buy Now` exists only on a product page and ends the episode. An action naming a button that is not on the current page is silently ignored: the page does not change and the attempt still costs a step.

One property of this store matters more than any other and is easy to miss: **a product page never shows which variant is currently selected**. Clicking `black` on a page offering `black` and `white` returns a page whose text is identical to the one before. Neither the policy nor a hook can read the selection off the page; the only record is `ctx['history']['actions']`.

Your action space

A patch is a JSON object with an optional `description` and a list of `actions` (at most 8). Four action types exist:

- `{"type": "add_hook", "hook": "<hook point>", "hook_id": "<your id>", "code": "<python>"}`
- `{"type": "replace_hook", "hook_id": "<an installed id>", "code": "<python>"}`
- `{"type": "delete_hook", "hook_id": "<an installed id>"}`
- `{"type": "set_param", "key": "<parameter>", "value": <number>}`

Enabled in this configuration: `add_hook`, `delete_hook`, `replace_hook`, `set_param`.

Hook points

The five hook points, when each runs, and what it may return:

on_episode_start — runs once, before the first decision. Whatever it injects is folded into the system message and stays visible for the whole episode.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`

on_before_decision — runs before every sampled action, after the previous observation is in hand. This is the only place that can rewrite what the policy is about to read.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`
- `{"kind": "transform_context", "observation": "<replacement observation text>"}`

on_before_action — runs after the action is parsed, before it reaches the store. It can rewrite the action or refuse it and make the policy answer again.

- `{"kind": "block_and_reprompt", "message": "<why, and what to do instead>"}`
- `{"kind": "normalize_action", "action": "search[...] or click[...]"}`

on_after_feedback — runs after the store responds, before the next decision is prepared. It sees the outcome of the action that just ran.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`
- `{"kind": "transform_context", "observation": "<replacement observation text>"}`

on_before_commit — runs only for `click[Buy Now]`, the irreversible action that ends the episode and submits the product. It runs after `on_before_action`.

- `{"kind": "block_and_reprompt", "message": "<why, and what to do instead>"}`
- `{"kind": "normalize_action", "action": "search[...] or click[...]"}`

Returning `None` means the hook stays out of the way this turn, which is the right answer most of the time: a hook that speaks every turn drowns out the turn where it mattered.

How the hooks work together

Treat a patch as a small runtime program rather than a single rule. One intervention usually reads best split across the lifecycle, with each hook owning one part of it:

- `on_episode_start` — one short reusable instruction, or the initial `nb` flags.
- `on_after_feedback` — update `nb` from what the page just showed. Prefer updating `nb` here over emitting an effect.
- `on_before_decision` — a deduplicated soft hint, once `nb` says the next subgoal is clear.
- `on_before_action` and `on_before_commit` — a hard block or rewrite, and only where the condition is narrow.

Splitting it that way also keeps each block short, which matters more than it looks: the code travels as a JSON string, and one unmatched bracket in a long block costs the entire patch.

An episode moves through recognisable stages—searching, reading a results page, inspecting a product, selecting options, checking the instruction's price ceiling, buying. Naming the stage in `nb` and advancing it in `on_after_feedback` is usually what lets a later hook fire at one moment and stay quiet everywhere else.

Parameters

Tunable parameters, with the range `set_param` accepts:

- `max_steps`: int in [4, 30], currently defaults to 15
- `max_obs_chars`: int in [200, 8000], currently defaults to 2000
- `max_history_turns`: int in [1, 30], currently defaults to 30
- `max_blocks`: int in [0, 8], currently defaults to 3
- `max_unrecognised_streak`: int in [0, 10], currently defaults to 3

The episode additionally clamps steps at 30 and blocks at 8 whatever the parameters say, so raising a budget to its maximum cannot buy more than every method already shares.

What a hook receives

```
ctx = {
  "hook": "<the hook point currently running>",
  "task": {"instruction_text": "<the customer instruction, verbatim>"},
  "action": None or {          # present for on_before_action / on_before_commit
    "raw": "click[buy now]",
    "type": "click" or "search",
    "arg": "buy now",
    "is_commit": True if this action submits the purchase,
    "is_recognised": True if the store will act on it,
  },
  "observation": {
    "text": "<the page text>",
    "page_type": "index" | "search_results" | "item_page" | "item_sub_page" | "done",
    "clickables": ["back to search", "< prev", "description", "features",
                   "reviews", "buy now", "<variant>", ...],
    "has_search_bar": True only on the search page,
  },
  "history": {
    "turn": <turns so far>,
    "actions": ["search[...]", "click[...]", ...],   # every action, in order
    "last_action": "<the previous action or None>",
    "action_types": {"search": n, "click": m},
    "page_types": ["index", "search_results", ...],
    "repeated_actions": <count of actions issued more than once>,
    "unrecognised_actions": <count the store ignored>,
    "unrecognised_streak": <consecutive ignored actions>,
    "blocked_actions": <count this harness has blocked>,
    "stall_streak": <consecutive steps whose observation did not change>,
  },
  "budget": {"max_steps": n, "steps_used": n, "steps_left": n,
             "max_blocks": n, "blocks_used": n, "blocks_left": n},
  "harness_state": {"fired": {...}, "effects": {...}, "failures": {...}},
  "params": {<the harness parameters listed above>},
}

nb = {}   # a plain dict, shared by every hook within one episode and empty at
          # its start. Mutate it to remember something across turns.
```

Sandbox

Hook code runs in a restricted Python subset. Code that breaks any of these is rejected before a single episode runs, and the patch scores zero:

- exactly one top-level function `def hook(ctx, nb):`, with that signature and no annotations; up to 5 further top-level helper functions are allowed, and nested functions are not
- no imports. These names are already bound: `SequenceMatcher`, `math`, `re`
- available builtins: `Exception`, `abs`, `all`, `any`, `bool`, `dict`, `divmod`, `enumerate`, `filter`, `float`, `int`, `isinstance`, `len`, `list`, `max`, `min`, `range`, `repr`, `reversed`, `round`, `set`, `sorted`, `str`, `sum`, `tuple`, `zip`
- forbidden syntax: `asyncfor`, `asyncfunction`, `asyncwith`, `await`, `class`, `delete`, `global`, `import`, `importfrom`, `lambda`, `nonlocal`, `raise`, `while`,

`with`, `yield`, `yieldfrom`. Loop with `for` over a bounded sequence; `while` does not exist here

- no name or attribute starting with _, and no `getattr`/`eval`/`exec`/`open`/`print`/`type`
- `try:` must catch `Exception` explicitly and may not use `else` or `finally`
- at most 8000 characters per hook, and it must return within 50 ms. Aim well under that ceiling: keep each hook compact and the patch's Python under about 6000 characters in total. Long blocks are where brackets go unmatched
- no product id literal (a code like `B01ABCDEFG`). A hook that names one has memorised an answer instead of expressing a policy
- text you emit is trimmed to 900 characters

A hook that raises at runtime is not fatal: it is downgraded to no effect. But it also never helps, so a hook that only works on one page shape quietly does nothing on the others. Guard your lookups.

Information boundary

What a hook may not do, beyond the sandbox:

- It cannot see the grader. There is no target product, no required attribute list, no parsed price ceiling. `ctx` holds only what the policy itself could see at that moment.
- The price ceiling is a near miss worth naming: it is written into the instruction in words (`price lower than 80.00 dollars`), so parsing it out of `instruction_text` is fair game. What does not exist is a pre-parsed number.
- Do not hard-code anything task-specific. A hook is run against tasks it has never seen, and product ids in the evidence have been replaced by aliases like `[PRODUCT_1]` precisely so they cannot be copied into code.
- Do not try to score the purchase. You improve how the policy attempts the task; you do not decide whether it succeeded.

How to answer

Your thinking block is already open. Continue the reasoning, close it with `</think>`, then emit exactly one `<patch>` block and nothing after it:

```
<your reasoning, in the shape set out below>
</think>
<patch>
{"description": "...", "actions": [...]}
</patch>
```

Do not open another `<think>`: one is already open, and a second one nests. Do not skip `</think>` either: without it the reasoning and the patch are one undifferentiated block, and the reasoning is read as part of the answer.

Keep the reasoning to short bullets, with no JSON among them. Which shape depends on the patch you are about to write.

A patch that touches a hook, four bullets:

1. the recurring failure you are targeting, and how often it appears;
2. what you will track in `nb`;
3. which hook owns which part of the logic;
4. the regression risk you are avoiding on episodes that already succeed.

A patch that only moves a parameter, three:

1. the recurring failure you are targeting, and how often it appears;
2. what in the evidence shows the current setting is the thing hurting, and what value you are moving it to;

3. the regression risk you are avoiding on episodes that already succeed.

The patch block must contain one JSON object and no prose, no markdown fences, and no comments. Python code goes inside the JSON `code` string, so newlines must be escaped as `\n`.

Editor System Prompt (ALFWorld): passages that differ

You are the harness editor for a household robot in a text-based simulation.

A *harness* is the runtime wrapped around a fixed language-model policy: it decides what the policy reads before each decision, may rewrite or refuse the action the policy chose, and controls the step and context budgets. The policy itself is frozen and will not be retrained. Your only lever is the harness, and your job is to make the same policy complete household tasks more often by editing it.

The simulation

The policy drives a household robot through one room, one command per turn. Every observation ends with the exact list of commands the simulation will accept at that moment, and only those work: anything else leaves the world unchanged and still costs a step. The commands are `go to <place>`, `open`/`close <place>`, `take <object> from <place>`, `move <object> to <place>`, `heat`/`cool`/`clean <object> with <place>`, `use <object>`, `examine`, and the three that only report—`look`, `inventory`, `help`. Reward is a single bit: the task is completed or it is not, with no partial credit for getting close.

Two properties matter more than the rest. **An object is invisible until the robot stands at the place holding it**, so finding one means visiting places in turn, and nothing in the room text says where to look; the only record of where the robot has already been is `ctx['history']['actions']`. And **the robot carries one object at a time**, so a `take` while holding something is not offered, which is a common way for an otherwise sensible plan to stall.

Hook points

The four hook points, when each runs, and what it may return:

`on_episode_start` — runs once, before the first decision. Whatever it injects is folded into the system message and stays visible for the whole episode.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`

`on_before_decision` — runs before every sampled action, after the previous observation is in hand. This is the only place that can rewrite what the policy is about to read.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`
- `{"kind": "transform_context", "observation": "<replacement observation text>"}`

`on_before_action` — runs after the command is parsed, before it reaches the simulation. It can rewrite the command or refuse it and make the policy answer again.

- `{"kind": "block_and_reprompt", "message": "<why, and what to do instead>"}`
- `{"kind": "normalize_action", "action": "one command, e.g. take mug 1 from desk 1"}`

on_after_feedback — runs after the simulation responds, before the next decision is prepared. It sees the outcome of the command that just ran, including the new list of commands on offer.

- {"kind": "inject_guidance", "message": "<text shown to the policy>"}
- {"kind": "transform_context", "observation": "<replacement observation text>"}

Returning None means the hook stays out of the way this turn, which is the right answer most of the time: a hook that speaks every turn drowns out the turn where it mattered.

How the hooks work together

Treat a patch as a small runtime program rather than a single rule. One intervention usually reads best split across the lifecycle, with each hook owning one part of it:

- on_episode_start — one short reusable instruction, or the initial nb flags.
- on_after_feedback — update nb from what the observation just showed. Prefer updating nb here over emitting an effect.
- on_before_decision — a deduplicated soft hint, once nb says the next subgoal is clear.
- on_before_action — a hard block or rewrite, and only where the condition is narrow.

Splitting it that way also keeps each block short, which matters more than it looks: the code travels as a JSON string, and one unmatched bracket in a long block costs the entire patch.

An episode moves through recognisable stages—searching for the object, picking it up, transforming it if the task asks for that, and putting it where the task named. Naming the stage in nb and advancing it in on_after_feedback is usually what lets a later hook fire at one moment and stay quiet everywhere else. ctx['observation']['phase'] distinguishes the coarse three, but not which places have already been searched: that is yours to track.

Parameters

Tunable parameters, with the range set_param accepts:

- max_steps: int in [4, 50], currently defaults to 50
- max_obs_chars: int in [2400, 12000], currently defaults to 4000
- max_history_turns: int in [1, 50], currently defaults to 50
- max_blocks: int in [0, 8], currently defaults to 3
- max_unrecognised_streak: int in [0, 10], currently defaults to 3

The episode additionally clamps steps at 50 and blocks at 8 whatever the parameters say, so raising a budget to its maximum cannot buy more than every method already shares.

What a hook receives

```
ctx = {
  "hook": "<the hook point currently running>",
  "task": {"instruction_text": "<the task, verbatim: 'Your task is to: ...'>"},
  "action": None or {              # present for on_before_action
    "raw": "take mug 1 from countertop 1",
    "type": "take",                # the command's verb
    "in_admissible": True if the simulation will act on it,
    "is_meta": True for look / inventory / help,
    "is_commit": always False; this environment has no submitting action,
  },
  "observation": {
    "text": "<the room text, without the command list>",
    "admissible": ["go to cabinet 1", "take mug 1 from countertop 1", ...],
    "phase": "navigating" | "at_objects" | "carrying",
  },
  "history": {
```

```
    "turn": <turns so far>,
    "actions": ["go to desk 1", "take mug 1 from desk 1", ...],  # every action, in order
    "last_action": "<the previous action or None>",
    "action_types": {"go": n, "take": m, ...},
    "phases": ["navigating", "at_objects", ...],
    "repeated_actions": <count of actions issued more than once>,
    "unrecognised_actions": <count the simulation ignored>,
    "unrecognised_streak": <consecutive ignored actions>,
    "blocked_actions": <count this harness has blocked>,
    "stall_streak": <consecutive steps whose observation did not change>,
  },
  "budget": {"max_steps": n, "steps_used": n, "steps_left": n,
             "max_blocks": n, "blocks_used": n, "blocks_left": n},
  "harness_state": {"fired": {...}, "effects": {...}, "failures": {...}},
  "params": {<the harness parameters listed above>},
}

nb = {}   # a plain dict, shared by every hook within one episode and empty at
          # its start. Mutate it to remember something across turns.
```

Information boundary

What a hook may not do, beyond the sandbox:

- It cannot see the grader. There is no expert walkthrough, no list of the places worth searching, no statement of which object satisfies the task. `ctx` holds only what the policy itself could see at that moment.
- The task text is the whole of what is known about the goal, and parsing it—for the object, the destination, whether it asks for heating or cleaning—is fair game. What does not exist is any pre-resolved form of it.
- Do not hard-code anything task-specific. A hook is run against rooms it has never seen: the same object is `mug 1` in one floor plan and absent from the next, and the number of drawers varies from three to seventeen.
- Do not try to score the task. You improve how the policy attempts it; you do not decide whether it succeeded.

Editor System Prompt (DBBench): passages that differ

You are the harness editor for a SQL database agent.

A *harness* is the runtime wrapped around a fixed language-model policy: it decides what the policy reads before each decision, may rewrite or refuse the action the policy chose, and controls the step and context budgets. The policy itself is frozen and will not be retrained. Your only lever is the harness, and your job is to make the same policy answer database questions more often by editing it.

The database

The policy answers a question by operating a SQL database, one action per turn. Two actions exist: `execute_sql[<SQL>]` runs one statement and returns the raw engine result, and `commit_final_answer[<JSON array>]` submits the answer and ends the episode. The opening observation lists table names and column headers only—row data is not shown, so reading a value means querying it. Reward is a single bit.

Two properties matter more than the rest. **A SELECT committed from the headers alone is a guess**, and nothing in the observation says whether the policy has looked; the only record of statements already tried is `ctx['history']['actions']`. And **INSERT/UPDATE/DELETE are graded on the table, not on the committed array**, so a policy that writes a correct SQL change and then forgets to run it, or commits without mutating, scores zero even if the array looks right.

Hook points

The five hook points, when each runs, and what it may return:

on_episode_start — runs once, before the first decision. Whatever it injects is folded into the system message and stays visible for the whole episode.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`

on_before_decision — runs before every sampled action, after the previous observation is in hand. This is the only place that can rewrite what the policy is about to read.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`
- `{"kind": "transform_context", "observation": "<replacement observation text>"}`

on_before_action — runs after the action is parsed, before it reaches the database. It can rewrite the action or refuse it and make the policy answer again.

- `{"kind": "block_and_reprompt", "message": "<why, and what to do instead>"}`
- `{"kind": "normalize_action", "action": "execute_sql[SELECT ...] or commit_final_answer[...]"}`

on_after_feedback — runs after the engine responds, before the next decision is prepared. It sees the outcome of the statement that just ran, including errors.

- `{"kind": "inject_guidance", "message": "<text shown to the policy>"}`
- `{"kind": "transform_context", "observation": "<replacement observation text>"}`

on_before_commit — runs only for `commit_final_answer`, the irreversible action that ends the episode and submits the answer. It runs after `on_before_action`.

- `{"kind": "block_and_reprompt", "message": "<why, and what to do instead>"}`
- `{"kind": "normalize_action", "action": "execute_sql[SELECT ...] or commit_final_answer[...]"}`

Returning `None` means the hook stays out of the way this turn, which is the right answer most of the time: a hook that speaks every turn drowns out the turn where it mattered.

How the hooks work together

Treat a patch as a small runtime program rather than a single rule. One intervention usually reads best split across the lifecycle, with each hook owning one part of it:

- `on_episode_start` — one short reusable instruction, or the initial `nb` flags.
- `on_after_feedback` — update `nb` from what the result just showed. Prefer updating `nb` here over emitting an effect.
- `on_before_decision` — a deduplicated soft hint, once `nb` says the next subgoal is clear.
- `on_before_action` and `on_before_commit` — a hard block or rewrite, and only where the condition is narrow.

Splitting it that way also keeps each block short, which matters more than it looks: the code travels as a JSON string, and one unmatched bracket in a long block costs the entire patch.

An episode moves through recognisable stages—reading the schema, inspecting columns, writing the answer query or the mutation, and committing. Naming the stage in `nb` and advancing it in `on_after_feedback` is usually what lets a later hook fire at one moment and

stay quiet everywhere else. `ctx['observation']['phase']` distinguishes schema / result / committed, but not which statements have already been tried: that is yours to track.

Parameters

Tunable parameters, with the range `set_param` accepts:

- `max_steps`: int in [4, 20], currently defaults to 20
- `max_obs_chars`: int in [200, 8000], currently defaults to 2000
- `max_history_turns`: int in [1, 20], currently defaults to 20
- `max_blocks`: int in [0, 8], currently defaults to 3
- `max_unrecognised_streak`: int in [0, 10], currently defaults to 3

The episode additionally clamps steps at 20 and blocks at 8 whatever the parameters say, so raising a budget to its maximum cannot buy more than every method already shares.

What a hook receives

```
ctx = {
  "hook": "<the hook point currently running>",
  "task": {"instruction_text": "<the question, verbatim>"},
  "action": None or {              # present for on_before_action / on_before_commit
    "raw": "execute_sql[SELECT ...]",
    "type": "execute_sql" or "commit_final_answer",
    "statement": "<the SQL, or None>",
    "answers": ["Vienna"] or None,  # only on commit
    "is_commit": True if this action submits the answer,
    "is_recognised": True if the engine will be asked to act on it,
  },
  "observation": {
    "text": "<schema, or the last SQL result>",
    "phase": "schema" | "result" | "committed",
    "tables": ["Football Teams", ...],
    "had_error": True if the last SQL was rejected,
    "row_count": <n or None>,
  },
  "history": {
    "turn": <turns so far>,
    "actions": ["execute_sql[...]", ...],
    "last_action": "<the previous action or None>",
    "action_types": {"execute_sql": n, "commit_final_answer": m},
    "phases": ["schema", "result", ...],
    "repeated_actions": <count of actions issued more than once>,
    "unrecognised_actions": <count the engine ignored>,
    "unrecognised_streak": <consecutive ignored actions>,
    "blocked_actions": <count this harness has blocked>,
    "stall_streak": <consecutive steps whose observation did not change>,
  },
  "budget": {"max_steps": n, "steps_used": n, "steps_left": n,
             "max_blocks": n, "blocks_used": n, "blocks_left": n},
  "harness_state": {"fired": {...}, "effects": {...}, "failures": {...}},
  "params": {<the harness parameters listed above>},
}

nb = {}   # a plain dict, shared by every hook within one episode and empty at
          # its start. Mutate it to remember something across turns.
```

Information boundary

What a hook may not do, beyond the sandbox:

- It cannot see the grader. There is no gold SQL, no label, no table hash. `ctx` holds only what the policy itself could see at that moment.
- The question text is the whole of what is known about the goal, and parsing it—for the columns it names, whether it asks to insert or to look up—is fair game. What does not exist is any pre-resolved form of it, and the table body is not in the opening observation.
- Do not hard-code anything task-specific. A hook is run against tables it has never seen: the same column name means something else on the next task, and a value that appears in the evidence is not an answer to copy.

- Do not try to score the answer. You improve how the policy attempts the task; you do not decide whether it succeeded.

D.2 EDITOR USER PROMPT

The user message pairs the harness as it stands with one batch's evidence, and is identical across benchmarks since everything benchmark-shaped in it comes from the harness and the packet. The {evidence_packet} slot holds the rendered packet of Appendix C. Installed hooks are listed in turn, or replaced by "No hooks are installed." when there are none.

Editor User Prompt

The harness as it stands

The harness you are editing is {harness_name} ({fingerprint}), with {n} installed hook(s).

Parameters: {parameter_dict}

{hook_id} at {hook_point}

```python
{hook_source}
```

{evidence_packet}

Emit one patch that addresses the most costly pattern above. Target one failure rather than several: every hook you add also runs on the episodes that were going fine, and a patch that helps the failures while disturbing the successes measures as nothing. Spreading that one intervention across the lifecycle hooks costs you nothing, though—what costs you is a second, speculative intervention, not a second hook.

E EDITOR COLD START

Teacher. The teacher is `qwen3.8-max`. It sees the prompt of Appendix D and is scored by the same validator and paired reward as Phase-E.

Degraded harnesses. Shown only h_0, the teacher writes `add_hook` and `replace_hook`. h_0 has no hook worth deleting and no parameter worth moving. The rest of the collection therefore runs against harnesses that start from h_0 and break one thing (Table 6). Variants carry across benchmarks by the damage they do. `hurried` keeps the same fraction of each default budget, and `bouncing` hangs on whichever hook intercepts a finishing action. `starved` cuts the observation budget on WebShop and DBBench, where product pages and SQL results are long enough to truncate. It cannot degrade ALFWorld, because those observations already fit in the smallest legal observation budget. `amnesiac` therefore elicits `set_param` there.

Eliciting `set_param`. The teacher deletes a bad hook readily, but shown a truncated observation budget it rewrites a hook instead of moving the parameter. One collection leg therefore restricts the action space to `set_param` and drops the unused operations from the prompt. Those demonstrations are stored under the full-action prompt the trained editor will see.

Collection and filtering. All legs share one shuffle of the reserved pool and take disjoint consecutive slices. Most of the budget stays on h_0; a degraded leg repeats one repair of one defect and saturates quickly. We keep a candidate when it is legal, both paired evaluations finish, and the paired reward is non-negative. Table 7 reports the resulting corpus. All three collections use $|B| = 16$. WebShop covers all 165 paired batches the reserved pool can make, at $K = 3$. ALFWorld has only

Table 6: Degraded harnesses used to collect deletion and retuning demonstrations.

Variant	Injected defect	Target operation
`nagging`	A hook that repeats the same long checklist before every decision, burying the situation-specific advice of h_0	`delete_hook`
`bouncing`	A hook that refuses the first submission of every episode	`delete_hook`
`both_hooks`	Both of the above	`delete_hook`
`starved`	Observation budget cut so most of what the policy reads is truncated (WebShop and DBBench)	`set_param`
`amnesiac`	History cut to two turns (ALFWorld only)	`set_param`
`hurried`	A step budget too small to inspect, decide, and finish	`set_param`

55 such batches and DBBench 72, so both raise K to 6. One demonstration can contain several operations, which is why the last four rows sum to more than the number kept.

Table 7: Cold-start corpus after filtering. Action rows count demonstrations that contain that operation at least once.

	WebShop	ALFWorld	DBBench
Paired batches / K	165 / 3	55 / 6	72 / 6
Kept demonstrations	306	169	124
from h_0	209	112	95
from degraded harnesses	97	57	29
Containing `add_hook`	241	136	110
Containing `replace_hook`	84	43	49
Containing `delete_hook`	49	34	10
Containing `set_param`	19	14	9
Mean paired reward	0.098	0.102	0.035

F Phase-E Implementation Details

Paired batch construction. The task pool is shuffled once from a recorded seed and cut in half at its full capacity: one half for evidence, the other for transfer. Batches are consecutive slices of those halves, so a task that appears in B_i never appears in B_i^+. Because a paired reward compares J_B before and after a patch, both runs cover the same tasks in the same order; every rerun is checked against a hash of those task identities.

Shared baseline. All K patches from one packet are scored against a single unpatched rerun of the pair. GRPO ranks the K patches against one another, so that shared baseline cancels from the ranking and saves $K - 1$ reruns.

Update bundling. One optimizer step consumes a bundle of packets, all sampled from the same editor weights. A larger bundle averages over more groups before the weights move, at the cost of more rollouts against those same weights. Groups whose K rewards are identical are skipped, because their advantages are already zero. Samples longer than the token limit are dropped.

Persistent patch and smoke test. Candidates for h_{t+1} are sampled against $\bar{s}_t^{\text{train}}$ within a fixed generation budget, in order. $\mathcal{V}$ already checks the schema, the sandbox, and that the patch changes something. A passing candidate is then installed on a copy and run on a small task set; only a crash disqualifies it. We take the first candidate that passes both.

G Data Splits

Table 8 lists the task-level splits. Cold-start, online-training, validation, and test tasks are disjoint on all three benchmarks, and the test split is evaluated once, for the selected checkpoint only.

WebShop. We keep the official i.i.d. boundaries induced by goal seed 233: indices $[0, 500)$ form the test split, $[500, 1500)$ the development split, and $[1500, 12087)$ the training pool. The training pool is shuffled with a pre-registered seed and cut into 5,293 tasks for editor cold start and 5,294 for online Phase-E and Phase-P. Validation is the first 200 development tasks, shared by all methods; the official order is already shuffled before the split is cut, so this slice is itself a random sample of the development pool.

ALFWorld. ALFWorld releases no labeled test set, since the underlying ALFRED test split is held by a leaderboard, so validation and closed testing both come from the official `valid` splits. We select checkpoints on `valid_train` (200 games) and keep the other two closed throughout training: `valid_unseen` (134 games) is our primary test split, sharing no task directory or floor plan with training, and `valid_seen` (140 games) is a secondary split whose scenes appear in training. The 3,553 official training games are partitioned with a pre-registered seed into 1,776 for editor cold start and 1,777 for online training. The cut is made at the task-directory level, so sibling games of one specification, differing only in initial object placement, never straddle the evidence/transfer boundary.

DBBench. AgentBench publishes a 300-task Test split and a 60-task Dev split, together with a public training pool. Official Dev is too small for the 200-task checkpoint selection used on the other two benchmarks, so we keep those 60 and carve 140 more from the pool, stratified by operation family (SELECT / INSERT / UPDATE) so a family scarce in the pool is not also scarce in validation. The remaining unique pool tasks are shuffled with the same pre-registered seed and cut in half, 2,316 for editor cold start and 2,316 for online training. The cut is made at the task. Test stays closed throughout training.

Table 8: Task-level data splits. The two ALFWorld test counts are the official `valid_unseen` (primary) and `valid_seen` (secondary) splits.

Benchmark	Editor cold start	Online training	Validation	Test
WebShop	5,293	5,294	200	500
ALFWorld	1,776	1,777	200	134 / 140
DBBench	2,316	2,316	200	300

H Hyperparameters

Table 9 lists the hyperparameters used for the reported runs. The three benchmarks share the group size, batch size, and Phase-P budget. ALFWorld and DBBench use fewer packets per round, and the editor bundle size is halved along with the packet count so that a round still consists of two equal gradient steps.

I Ablation Variants

Every variant below shares the full method's initialization (ϕ_0, θ_0, h_0), round count $T = 10$, and per-round rollout budget, so a variant that optimizes one side alone spends the whole round on that side.

Which sides learn, and whether they alternate. **Policy-only** deletes Phase-E and holds the harness at h_0 for every round, which is the conventional policy-RL paradigm of Figure 1(a). **Editor-only** deletes Phase-P and holds the policy at π_{ϕ_0}, so the editor keeps learning and the harness keeps accumulating patches against a target that never moves, as in Figure 1(b). **Sequential (E→P)** optimizes both sides but never interleaves them: all T editor rounds run first against π_{ϕ_0} to produce a final harness h_T, and the policy is then trained for the same total number of Phase-P steps. It isolates *live* alternation, since only the order of the two learners' updates changes. **Frozen editor** keeps the alternation but stops the editor from learning: each round draws its persistent patch from the cold-start weights θ_0 rather than θ_{t+1}, so harness edits come from a fixed proposer while the policy learns, which is the asymmetry we ascribe to Figure 1(c).

Table 9: Hyperparameters used for the reported runs. Values that differ across benchmarks are the Phase-E packet count and the editor bundle size.

Stage	Quantity	WebShop	ALFWorld	DBBench
Phase-E	candidate patches per packet K	8	8	8
	paired batches per round	8	4	4
	packets per optimizer step	4	2	2
	evidence/transfer batch size $\|B\|$	16	16	16
	paired-reward weight α	0.5	0.5	0.5
	excerpts per packet	8	8	8
	packet token budget	6000	6000	6000
	persistent-patch generation budget	8	8	8
	smoke-test episodes	8	8	8
	editor LoRA rank / learning rate	32 / 10^{-5}	32 / 10^{-5}	32 / 10^{-5}
Phase-P	trajectories per unit K	8	8	8
	units per round	64	64	64
	units per optimizer step	4	4	4
	current-harness probability ρ	0.8	0.8	0.8
	policy LoRA rank / learning rate	32 / 10^{-5}	32 / 10^{-5}	32 / 10^{-5}

Mechanisms for two-sided optimization. **w/o transfer batch** sets $\alpha = 1$, scoring a patch only on the evidence batch that produced its packet and removing the pressure against edits that repair just the episodes the editor was shown. **w/o evidence batch** is the mirror image, $\alpha = 0$: reward comes only from the hidden transfer batch, so a patch is never credited for the failure it was written to address.

J Additional Results

Table 10 is the source of Figure 3, and Table 11 groups every held-out episode by how it ended. Both use the same frozen test pass as Table 1.

Success versus cost. Table 10 reports mean environment steps and prompt tokens per episode. Each Δ is against that method's own vanilla: the default harness for Harness-R1 and h_0 for DuetHarness. Harness-R1 raises WebShop success by 9.0 points while spending 28% more steps and 50% more tokens, trims ALFWorld from 32.0 to 27.1 steps (-15%) and from 184.8k to 149.2k tokens (-19%), and raises DBBench cost from 5.5 to 6.0 steps ($+9\%$) and from 8.0k to 10.0k tokens ($+25\%$). DuetHarness raises WebShop success by 23.6 points while cutting steps from 7.8 to 4.4 (-44%) and tokens from 20.4k to 6.8k (-67%), and on ALFWorld raises success by 47.8 points while cutting steps from 31.7 to 17.4 (-45%) and tokens from 209.8k to 80.2k (-62%). On DBBench it raises success by 7.0 points while steps and tokens rise slightly, from 4.8 to 5.2 ($+8\%$) and from 5.7k to 6.3k ($+11\%$). Tokens fall farther than steps on WebShop and ALFWorld because a shorter episode also replays less history into later prompts.

How episodes end. Table 11 partitions the held-out split into success, abandonment, and remaining errors: the grader's reason for rejecting a purchase on WebShop, and an incorrect answer on DBBench. ALFWorld has no leftover class, because every failure is an abandonment. The last row averages how many commands the environment refused. Under h_0, 128 of 500 WebShop episodes never purchase (98 exhaust the step budget, 30 hit the illegal-action limit) and 151 of 274 ALFWorld episodes are abandoned the same way. DuetHarness leaves 2 and 20. Those abandoned episodes are the expensive ones on WebShop and ALFWorld, which is why the means in Table 10 fall there. DBBench is almost never abandoned ($5 \to 0$ of 300); the remaining $118 \to 102$ failures are completed answers that do not match.

The remaining WebShop errors are mismatches against the instruction. Of the 372 purchases under h_0 and the 498 under DuetHarness, missing a required option is still the largest class ($183 \to 150$). Buying the wrong product stays at 39: the target product never enters hook context (Appendix A). Missing attributes and price overruns rise in count ($32 \to 56$ and $7 \to 24$) because more episodes

Table 10: Held-out test success (%), mean environment steps, and mean prompt tokens per episode (Tok., $\times 10^3$), corresponding to Figure 3. ALFWorld pools the unseen (134) and seen (140) test splits. Each Δ is against that method's own vanilla: the default harness for Harness-R1 and h_0 for DuetHarness. A step is an environment action.

Method	WebShop			ALFWorld			DBBench		
	Succ.	Steps	Tok.	Succ.	Steps	Tok.	Succ.	Steps	Tok.
Qwen3.5-9B + default	32.0	4.0	7.8	35.8	32.0	184.8	61.7	5.5	8.0
Harness-R1	41.0	5.1	11.7	48.9	27.1	149.2	65.0	6.0	10.0
Δ	$+9.0$	$+28\%$	$+50\%$	$+13.1$	-15%	-19%	$+3.3$	$+9\%$	$+25\%$
Qwen3.5-9B + h_0	22.2	7.8	20.4	44.9	31.7	209.8	59.0	4.8	5.7
DuetHarness	45.8	4.4	6.8	92.7	17.4	80.2	66.0	5.2	6.3
Δ	$+23.6$	-44%	-67%	$+47.8$	-45%	-62%	$+7.0$	$+8\%$	$+11\%$

now reach a purchase; as a share of those purchases they move from 8.6% to 11.2% and from 1.9% to 4.8%.

Table 11: How held-out episodes end, as counts, on the WebShop (500), ALFWorld (134 unseen + 140 seen), and DBBench (300) test splits.

Outcome	WebShop		ALFWorld		DBBench	
	h_0	DuetHarness	h_0	DuetHarness	h_0	DuetHarness
Success	111	229	123	254	177	198
Step budget exhausted	98	1	101	15	1	0
Illegal-action limit	30	1	50	5	4	0
Incorrect answer	—	—	—	—	118	102
Missing required option	183	150	—	—	—	—
Missing required attribute	32	56	—	—	—	—
Bought wrong product	39	39	—	—	—	—
Over price ceiling	7	24	—	—	—	—
Illegal actions per episode	1.1	0.6	5.4	1.7	0.0	0.05

K Alternating Learning Process

Figure 4 plots validation metrics after every Phase-E and Phase-P update of DuetHarness, and the hooks present from h_0 through later rounds. On WebShop, installing a new harness often drops both score and success, so an editor-only update can break the current policy–harness fit until Phase-P adapts the policy to the revised runtime. On ALFWorld the reward is binary, so the two metrics coincide, and both phases keep rising. Both runs keep the two h_0 guidance hooks and drop the generic `unrecognised` hint at `on_after_feedback`, which only spoke after the environment had already spent a step. The learned hooks instead block or rewrite the pending action at `on_before_action`. Tables 12 and 13 list every hook that appears in the figure.

On WebShop the first learned pair only watches the results page, where `loop_guard` counts turns that stay on the same non-item page and `loop_hint` then points the policy at product links. The first lasting before-action guard is `fix_search_on_item`, which refuses a search issued from a product page and asks for `Back to Search`. Early attempts to police header clicks and off-list buttons are installed and dropped again in the round that brings variant handling. The selected harness instead warns that variant buttons are case-sensitive, rewrites a click to the listed spelling, reminds the policy that those buttons are not `Buy Now`, and refuses `Buy Now` when the button is not actually clickable. A later `variant_fixer` takes over from `invalid_variant_guard`, fuzzy-matching a near-miss variant rather than only refusing it. Two hooks appear only after the selected checkpoint, remembering the page type and refusing variant-like clicks when no product is open; they are not used at test.

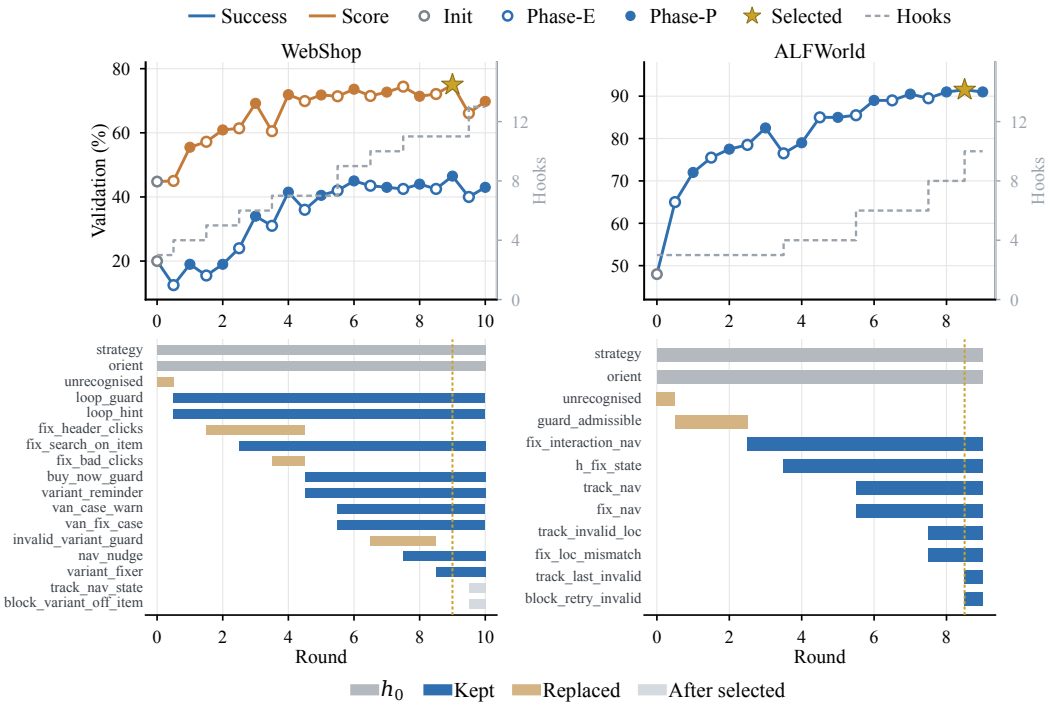

Figure 4: Validation metrics and harness evolution over alternating rounds. Top: WebShop shows score and success; ALFWorld reward is binary, so the two coincide. Hollow markers are Phase-E, filled markers are Phase-P, and the star is the checkpoint selected for the held-out test. The dashed line counts installed hooks. Bottom: each bar is a hook from h_0 (gray) through later rounds. Blue hooks remain at the selected checkpoint. Tan hooks were added and later replaced.

On ALFWorld the editor first installs a generic admissible-list guard, `guard_admissible`, then removes it in the round that brings `fix_interaction_nav`, and reinstates an admissible check one round later as `h_fix_state`, which stays to the end. The distinctive lasting hook is `fix_interaction_nav`, which rewrites the action to walk there first when `heat`, `cool`, `clean`, or `move` names a place the agent is not in and `go to` that place is already admissible. Companion hooks count failed `go to` attempts and list valid destinations after a streak, remember a transform that failed because the agent was in the wrong room, refuse retrying it until the location matches, and refuse repeating the same illegal command. That is the locate-before-transform pattern the long-horizon families need, and it acts before the environment spends a step, whereas h_0 only hinted after an illegal command had already failed.

L Unilateral Residual Probe

Approximate best-response updates need not reach a Nash equilibrium. To test whether the pair retained for testing can still be regarded as an empirical Nash, we run this probe on WebShop. A pair is an empirical Nash if neither side can meaningfully raise J over a short window. We evaluate that pair on the 200-task validation split and then continue the same operator for three rounds with the counterpart frozen. Residual gain is $\max(J - J_0, 0)$ at the end of the window. Table 14 summarizes the two probes.

Holding that harness fixed, Phase-P raises J from 0.727 to 0.733, 0.733, and 0.739. Holding that policy fixed, Phase-E writes a new harness fingerprint each round and J falls from 0.727 to 0.721, 0.679, and 0.615. The remaining policy residual is $+0.013$ and the harness residual is 0. We regard this leftover gain as negligible and treat the pair as an empirical Nash under the operator.

Table 12: WebShop hooks in Figure 4. Status is h_0, kept at the selected checkpoint, replaced before it, or installed only after it (unused at test). Location abbreviations are start, before dec., before act., after fb., and commit for the five points in Table 3.

Hook	Location	Status	Role
`strategy`	start	h_0	Shopping recipe: search, open a product, match variants, then buy.
`orient`	before dec.	h_0	Nudge when the budget is low or the page has stalled.
`unrecognised`	after fb.	Replaced	After the store ignores an action, hint to use a listed button.
`loop_guard`	after fb.	Kept	Count consecutive turns that stay on the same non-item page.
`loop_hint`	before dec.	Kept	After that stall, point at product links on the results page.
`fix_header_clicks`	before act.	Replaced	Refuse clicks on category headers that look like variants.
`fix_search_on_item`	before act.	Kept	Refuse search from a product page; ask for Back to Search.
`fix_bad_clicks`	before act.	Replaced	Refuse clicks that are not on the current product page.
`buy_now_guard`	commit	Kept	Refuse Buy Now when that button is not clickable.
`variant_reminder`	before dec.	Kept	On a product page, remind that variant buttons are not Buy Now.
`van_case_warn`	start	Kept	Warn that variant buttons are case-sensitive.
`van_fix_case`	before act.	Kept	Rewrite a click to the exact case of a listed button.
`invalid_variant_guard`	before act.	Replaced	Refuse a click that matches no button on the product page.
`nav_nudge`	before dec.	Kept	On a product page, say search is unavailable until Back to Search.
`variant_fixer`	before act.	Kept	Fuzzy-match a near-miss variant, or refuse it and list valid buttons.
`track_nav_state`	after fb.	Late	Remember the current page type.
`block_variant_off_item`	before act.	Late	Refuse variant-like clicks when no product is open.

Table 13: ALFWorld hooks in Figure 4. Status is as in Table 12.

Hook	Location	Status	Role
`strategy`	start	h_0	Locate–transform–place recipe: find the object, act on it, put it where the task named.
`orient`	before dec.	h_0	Nudge when the budget is low or the room has stalled.
`unrecognised`	after fb.	Replaced	After an illegal command has already spent a step, hint to copy an admissible command.
`guard_admissible`	before act.	Replaced	Refuse a command that is not in the current admissible list.
`fix_interaction_nav`	before act.	Kept	If heat/cool/clean/move names a place the agent is not in, rewrite the action to go there first.
`h_fix_state`	before act.	Kept	Refuse a command that is not currently admissible.
`track_nav`	after fb.	Kept	Count failed `go to` attempts.
`fix_nav`	before dec.	Kept	After a failed navigation streak, list valid destinations.
`track_invalid_loc`	after fb.	Kept	Remember a transform or move that failed because the agent was in the wrong place.
`fix_loc_mismatch`	before act.	Kept	Refuse retrying that transform or move until the agent is at the required place.
`track_last_invalid`	after fb.	Kept	Remember the last illegal command.
`block_retry_invalid`	before act.	Kept	Refuse repeating that same illegal command.

Table 14: Unilateral residual probe of the WebShop pair retained for testing. Residual is $\max(J - J_0, 0)$ after three frozen-side rounds on the 200-task validation split.

Probe	Score	Residual	Outcome
Starting pair	0.727	—	Pair retained for testing
Policy, harness frozen	$0.727 \to 0.739$	$+0.012$	Per-round $\lvert\Delta\rvert < 0.03$
Harness, policy frozen	$0.727 \to 0.615$	0	Score falls