HERO: AGENT SELF-IMPROVEMENT THROUGH HARNESS–EXECUTOR RECIPROCAL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The ability to learn from experience and continually improve is fundamental to intelligence. We introduce HERO, an agent self-improvement framework that adapts experience collection alongside executor learning. A task-conditioned harness generator and an executor, initialized from the same pretrained model, learn through their own environment interactions. Direct preference optimization trains the generator using preferences derived from verified execution outcomes. Successful action trajectories provide supervision for executor fine-tuning, with student inputs reconstructed under a fixed deployment harness. The updated executor then supplies feedback for subsequent generator adaptation. This alternating process uses self-generated experience without additional teacher-provided solutions or critiques. We evaluate HERO with Qwen3.5-9B on ALFWorld and WebShop. On ALFWorld, the final executor reaches 97.0% success on the official unseen split. On WebShop, three consecutive DPO–SFT stages improve success on the official 500-task test split from 31.2% to 45.6%. The final executors achieve these results under fixed deployment harnesses, requiring no extra generator calls. Analysis shows that the trained generator yields more clean successful trajectories and shorter average rollouts than its untrained counterpart. These findings highlight HERO as a way to shape the experience from which an executor learns.

1 INTRODUCTION

Language-model agents can learn from successful interactions with their environments through self-training Xi et al.. The experience they collect depends on both the executor and its harness—the instructions and runtime controls that guide execution. A harness can change the actions attempted and the paths taken to complete a task, thereby changing the trajectories available for learning.

We view agent self-improvement as learning both how to collect experience and how to use it. A trainable harness generator can adjust task-specific guidance using feedback from the current executor. Successful trajectories then provide action supervision for updating the executor, whose changed behavior motivates fresh generator adaptation. Our objective is to improve the executor's task success under a fixed deployment harness H_0. ScoreFlow Wang et al. (2025) learns a task-conditioned workflow generator from execution scores. Co-Harness Chen et al. (2026) and WHALE Kim et al. (2026) alternate harness revision or search with model training. Experience Funnel Gao et al. (2026) further studies how explicit experience is transferred into model parameters. Our focus is a learned, task-conditioned harness policy whose execution experience is converted into action supervision for a separately trained executor under a fixed deployment interface.

We introduce **HERO (Harness–Executor Reciprocal Optimization)** (Figure 1), which trains a harness generator and an executor from the same pretrained model using separate trainable parameters. At task initialization, the generator uses public task information and fixed design rules to produce a bounded harness specification. We freeze the executor, evaluate candidate harnesses in the environment, and use execution success and failure to train the generator with direct preference optimization (DPO) Rafailov et al. (2023). Successful trajectories provide action supervision for executor self-training. We reconstruct their inputs under H_0, omit generated guidance, and use only information available before each target action. Supervised fine-tuning (SFT) then trains the executor to reproduce the recorded actions through the interface used at deployment. After the executor update, fresh executions supply preferences for the next generator update. DPO therefore learns

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.

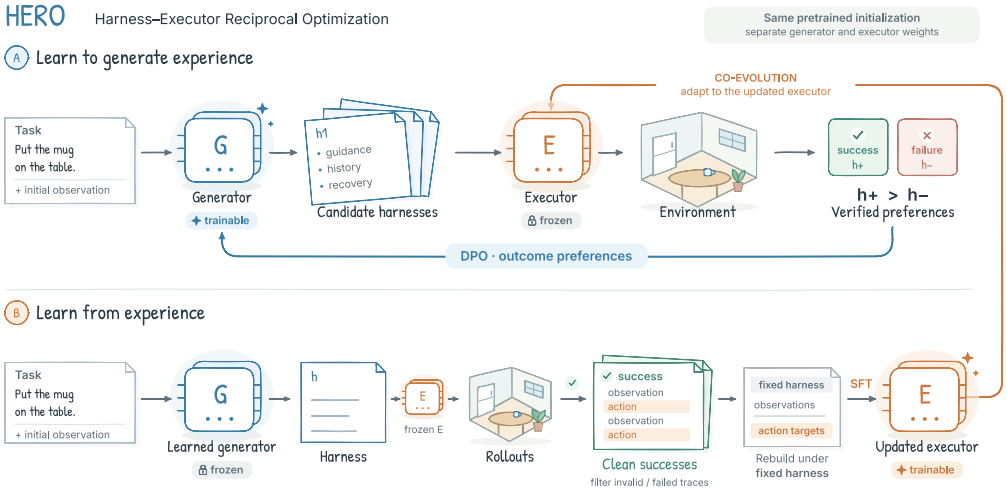

Figure 1: **Overview of HERO. (A) Learning to generate experience.** A task-conditioned generator proposes candidate harnesses, which a frozen executor evaluates through environment interactions. Verified execution outcomes define preferences for training the generator with DPO. **(B) Learning from experience.** The learned generator and current executor collect trajectories with their weights frozen. Successful trajectories are reconstructed under a fixed deployment harness to fine-tune the executor, supervising only recorded action tokens. The updated executor then guides subsequent generator adaptation, forming an iterative co-evolution loop.

from current execution outcomes, while subsequent executor gains are evaluated after self-training under H_0. The training uses the models' own interactions and environment feedback, without additional teacher-provided solutions or critiques.

We evaluate HERO with Qwen3.5-9B on ALFWorld Shridhar et al. (2020) and WebShop Yao et al. (2022). The final executors reach 97.01% success on ALFWorld's official unseen split and 45.60% on the WebShop test set, using H_0 without generator calls at evaluation. We also compare generators with frozen executors and evaluate trajectory sources under matched student initialization and SFT budgets, measuring both success and execution length.

Our contributions are:

- We propose an agent self-evolution paradigm in which agents learn from their own experience and adapt how they collect new experience without additional teacher-provided solutions or critiques.
- To implement this paradigm, we develop HERO, a framework that alternates DPO training of a task-conditioned harness generator with executor SFT on successful trajectories.
- Experiments show that the trained generator yields more clean successful trajectories and reduces average rollout length, providing more verified experience for self-training.
- HERO substantially improves task success and achieves competitive results against reported methods, without generator calls at deployment.

2 Related Work

2.1 Agent Self-Improvement from Self-Generated Experience

Self-training turns feedback on model-generated outputs into supervision. ReST^{EM} Singh et al. (2023) filters generated solutions by correctness and fine-tunes on successes; AgentEvol within AgentGym Xi et al. applies this loop to interactive environments. WebRL Qi et al. (2025) combines failure-driven curricula with experience replay, while ETO Song et al. (2024) learns execution policies from success–failure trajectory preferences. Re-ReST Dou et al. (2024) repairs failed outputs with a trained reflection model, and Self-Challenging Language Model Agents Zhou et al. (2026)

generate tasks and verification functions. SEAL Zweiger et al. (2026) generates self-edits for model adaptation and optimizes them using post-update performance. HERO trains its harness generator from verified execution outcomes and assesses the resulting experience through subsequent executor self-training.

2.2 Automated Harness Design and Optimization

Automated system design optimizes the prompts, programs, and runtime controls around language models. DSPy Khattab et al. (2023) optimizes modular programs, and TextGrad Yuksekgonul et al. (2024) updates text variables using propagated feedback. ADAS Hu et al. (2025), AFlow Zhang et al. (2025), and Meta-Harness Lee et al. (2026) search agent programs, workflows, and executable harnesses, respectively. ScoreFlow Wang et al. (2025) learns a task-conditioned workflow generator through a DPO variant based on execution scores. Harness-R1 Shao et al. (2026) trains a failure-conditioned engineer with SFT and GRPO to edit executable harnesses, keeping the target executor frozen during engineer training. HERO generates bounded specifications from the initial task context and uses the resulting successful actions for subsequent executor learning.

2.3 Joint Optimization of Harnesses and Executors

Joint optimization adapts the harness as the model changes. BetterTogether Soylu et al. (2024) alternates prompt optimization and model fine-tuning; Co-Harness Chen et al. (2026) and WHALE Kim et al. (2026) couple harness revision or search with training on collected trajectories. HASE Luo et al. (2026) combines task actions and harness edits within a trainable policy, while EvoHarness-RL Ning et al. (2026) learns when to act in the environment or operate on external state. Experience Funnel Gao et al. (2026) transfers explicit experience into model parameters. Work on on-policy expert correction Yu et al. (2026) shows that successful teacher trajectories can still disrupt a student's fit to an evolved harness. HERO alternates DPO for a bounded, task-conditioned harness generator with executor self-training on its own successful actions. Student inputs are reconstructed under H_0, enabling learning and evaluation through a fixed deployment interface without additional teacher-provided solutions or critiques.

3 Method

In this section, we first define the learning problem and formalize the harness. We then describe Harness Generator DPO and Executor Self-Training. Finally, we combine these two updates into a harness–executor co-evolution procedure and summarize the complete algorithm.

3.1 Problem Formulation

We consider an agent that interacts with an environment to complete tasks drawn from a distribution $\mathcal{D}$. Each task provides a goal q and an initial public observation o_0, which together form the initial context $x = (q, o_0)$. At each step t, the agent selects an action a_t and receives a new observation o_{t+1}. This interaction produces a trajectory $\tau = (o_0, a_0, o_1, \ldots, a_{T-1}, o_T)$ within a fixed interaction budget B, where $T \leq B$. The environment provides a verifiable outcome $R(x, \tau) \in \{0, 1\}$, indicating whether the task is successfully completed.

A harness specifies the instructions and runtime controls that guide the executor's interaction with the environment. HERO consists of a harness generator G_ϕ and an executor E_θ, initialized from the same pretrained language model and maintained with separate trainable parameters. At the beginning of a task, the generator observes x and samples a specification $z \sim G_\phi(\cdot \mid x; \mathcal{I})$, where $\mathcal{I}$ denotes fixed generation instructions. A deterministic compiler $\mathcal{C}$ composes a valid specification with the fixed base harness H_0, yielding $H = \mathcal{C}(H_0, z, q) \in \mathcal{H}(H_0)$. This admissible harness space preserves the task, environment interface, success criterion, and interaction budget. The executor then interacts with the environment under H. We denote the resulting trajectory distribution by $p_\theta(\tau \mid x, H)$.

Our objective is to improve the executor's expected task success under the fixed deployment harness H_0:

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D},\, \tau \sim p_\theta(\cdot \mid x, H_0)} \left[R(x, \tau) \right]. \tag{1}$$

To support this objective, the generator learns to produce harness specifications that improve the current executor's task success, and the executor learns from successful trajectories collected under their compiled harnesses. The generator's immediate learning signal is verified execution success; its contribution to subsequent executor learning is assessed through $J(\theta)$ under the common deployment harness. Training uses recorded interactions and environment feedback from training tasks, without additional teacher-provided solutions or critiques.

3.2 HARNESS FORMALIZATION

We implement a task-conditioned harness as a bounded specification composed with the fixed base harness H_0. The base provides the executor instructions, tool interface, observation format, and a state tracker that records progress from public feedback. The generated specification controls how this interface is adapted to a particular task. As illustrated in Fig. 2, we represent it as

$$z = (m, w, k, f, u, v), \tag{2}$$

where m selects whether to retain the base harness or enable task-specific guidance; w controls the recent-history window; k sets the threshold for a recovery reminder; f selects the reminder focus; and u and v provide task and recovery guidance, respectively. In our implementation, $w \in \{4, 6, 8\}$, $k \in \{1, 2, 3\}$, and $f \in \{\texttt{subgoals}, \texttt{exploration}, \texttt{commands}\}$. The guidance strings u and v are limited to 900 and 450 characters.

At the beginning of an episode, the generator receives the initial context $x = (q, o_0)$, together with fixed generation instructions $\mathcal{I}$ containing the base executor prompt, output schema, and design rules. These rules ask for guidance grounded in the public task information, including task prerequisites, search discipline, and checks against environment feedback. The generator emits a JSON specification, which is validated against the schema and composed with H_0 by a deterministic compiler $\mathcal{C}$:

$$z \sim G_\phi(\cdot \mid x; \mathcal{I}), \qquad H = \mathcal{C}(H_0, z, q). \tag{3}$$

Compilation maps valid specifications z to runtime harnesses H in the admissible space $\mathcal{H}(H_0)$. The `passthrough` mode returns H_0. In `bounded_hints` mode, the compiler adds a public goal checklist derived from q, appends u to the system instructions, and configures history and recovery behavior. Unsupported goal forms fall back to H_0.

During execution, a fixed renderer $\mathcal{R}$ constructs the executor context from the harness, current observation, and recorded interaction prefix:

$$c_t = \mathcal{R}(H, x, \tau_{<t}, o_t), \qquad a_t \sim E_\theta(\cdot \mid c_t). \tag{4}$$

The context includes recent interaction history, the public state summary, current admissible commands, and a reminder selected by f. Recovery guidance v is displayed after k consecutive identical failed action–result pairs. The specification z remains fixed throughout the episode, while the rendered context changes with new observations. The executor selects the actions; the generated specification changes its guidance and context management.

Figure 2(a) illustrates the harness structure using a recorded example for the goal "put a clean soapbar in cabinet." The generated specification selects a history window of six, a recovery threshold of two, and a subgoal-focused reminder. Figure 2(b) shows how the initial task context and fixed design rules lead to this specification and its composition with H_0.

3.3 HARNESS GENERATOR DPO

We train the generator using preferences derived from task execution. At the start of each generator update stage, we freeze the current executor E_θ and retain a frozen copy $G_{\phi_{\text{ref}}}$ of the generator. For each training context x, we sample K specifications $z_i \sim G_{\phi_{\text{ref}}}(\cdot \mid x; \mathcal{I})$. Each valid specification is compiled into H_i and evaluated under the same task reset, executor settings, and interaction budget, producing a trajectory τ_i and an outcome $r_i = R(x, \tau_i)$. In co-evolution rounds, we additionally evaluate a deterministic `passthrough` specification z_0 that reproduces the fixed harness H_0.

We construct a preference pair (x, z^+, z^-) when two valid specifications share the same initial prompt and their recorded executions yield $r^+ = 1$ and $r^- = 0$. Successful candidates must also pass a conservative lexical filter over transformation and appliance terms in their guidance.

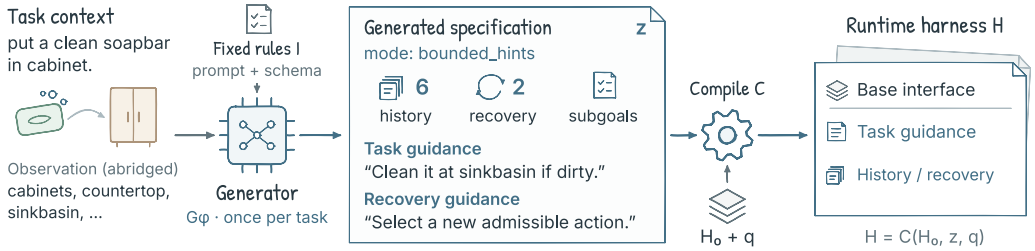

Figure 2: **Task-conditioned harness generation.** At task initialization, G_ϕ generates z from the task context and fixed rules. The compiler combines z with H_0 and the goal q to construct the runtime harness H. The example uses recorded values and verbatim guidance excerpts.

We remove duplicate policy pairs and retain at most two pairs per task. When a fixed-harness control is included, one eligible pair involving z_0 is prioritized; remaining pairs are selected by text similarity, execution cost, and deterministic tie-breaking. Tasks without an eligible success–failure pair contribute no DPO examples. The retained triples form the preference dataset $\mathcal{P}$.

We optimize the generator with direct preference optimization Rafailov et al. (2023). In the following expressions, $G_\phi(z \mid x)$ denotes the probability of the canonical JSON response encoding z, with the fixed instructions $\mathcal{I}$ omitted for clarity. We define the preference margin relative to the frozen reference as

$$\Delta_\phi(x, z^+, z^-) = \log \frac{G_\phi(z^+ \mid x)}{G_{\phi_{\text{ref}}}(z^+ \mid x)} - \log \frac{G_\phi(z^- \mid x)}{G_{\phi_{\text{ref}}}(z^- \mid x)}. \tag{5}$$

The DPO objective is

$$\mathcal{L}_{\text{DPO}}(\phi) = -\mathbb{E}_{\mathcal{P}} \left[\log \sigma\big(\beta \Delta_\phi(x, z^+, z^-)\big) \right], \tag{6}$$

where the expectation assigns equal weight to each retained pair, σ is the logistic sigmoid, and $\beta > 0$ controls the preference scale. We use $\beta = 0.1$. Minimizing this loss favors successful specifications over failed ones relative to their probabilities under the reference generator. The reference retains all generator updates from preceding stages and remains fixed during the current stage.

Sequence log-probabilities are summed over response tokens and the end-of-sequence token, excluding prompt and padding positions. Reference log-probabilities are computed once and reused throughout training. Gradient updates change only the generator parameters using the collected preference dataset, while the executor remains frozen. Environment interaction thus supplies discrete supervision for a differentiable generator objective; the executor's actions and environment transitions are outside the gradient computation.

3.4 Executor Self-Training

We improve the executor by learning from its successful interactions. During trajectory collection, the generator G_ϕ and the current executor E_θ are held fixed. The generator samples a specification z for each training task, and the executor interacts with the environment under its compiled harness H. In co-evolution rounds, collection uses the generator obtained from the preceding DPO stage. We retain complete trajectories with environment-verified success and consistent execution records. Episodes containing invalid or rewritten actions, duplicate tool calls, missing model turns, or runtime errors are excluded. The retained task–trajectory pairs form a set $\mathcal{T}^+$.

To train the executor for deployment under H_0, we reconstruct its input at every recorded action step. For each $(x, \tau) \in \mathcal{T}^+$, the recorded observations and actions are replayed through the fixed harness, yielding

$$c_t^{(0)} = \mathcal{R}(H_0, x, \tau_{<t}, o_t). \tag{7}$$

This reconstruction uses the base instructions, history window, public state tracker, and tool interface of H_0, while preserving the recorded interaction prefix. Each context contains only information

available before its target action. Generated task and recovery guidance are omitted. Replay operates on saved records and requires no new model generation or environment interaction.

The target y_t is the canonical tool-call serialization of the recorded action a_t, including the end-of-sequence token. We pair it with $c_t^{(0)}$ to form an action-level training example. Examples exceeding the context limit are excluded, and all retained examples preserve the complete action target. Let $\mathcal{S}$ denote the resulting dataset. We minimize

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(c,y)\sim\mathcal{S}}\left[\frac{1}{|y|}\sum_{j=1}^{|y|}\log E_\theta(y_j \mid c, y_{<j})\right], \tag{8}$$

where c is a reconstructed context and $|y|$ counts the supervised target tokens. The loss averages token negative log-likelihood within each action example and then across examples. Prompt tokens, observations, and padding are masked from the loss; supervision is applied only to the recorded action output and its end-of-sequence token.

Each SFT stage starts from the current executor checkpoint and updates only its trainable parameters, with fixed generator. Co-evolution stages use their newly collected trajectories without mixing in trajectories from earlier stages. Reconstructing all student contexts under H_0 aligns the training interface with fixed-harness deployment, where the updated executor is evaluated through $J(\theta)$.

3.5 Harness–Executor Co-Evolution

HERO combines generator DPO and executor self-training into an alternating learning process. We denote the generator and executor at stage r by $G_r = G_{\phi_r}$ and $E_r = E_{\theta_r}$. Both start from the same pretrained model, with separate trainable parameters.

We first collect an initial bank of executions using the untrained generator G_0 and executor E_0. This bank supplies two datasets: execution preferences $\mathcal{P}_0$ for training G_1, and successful action examples $\mathcal{S}_0$ for training E_1. For initial executor training, we retain one clean successful trajectory per task, choosing the shortest eligible trajectory when several are available, and reconstruct its contexts under H_0. Thus, G_1 and E_1 learn from separate supervision derived from the same initial collection; the trajectories used to train E_1 were generated by G_0/E_0.

After bootstrap, we perform L co-evolution rounds indexed by $r = 1, \ldots, L$, starting from (G_1, E_1). Each round performs two updates. First, G_r proposes candidate specifications that are compiled into harnesses and evaluated by the frozen executor E_r. Their outcomes form a fresh preference dataset $\mathcal{P}_r$, and DPO produces G_{r+1} using G_r as the frozen reference. Second, G_{r+1} and E_r, both held fixed, collect new trajectories. Clean successes are reconstructed under H_0 to form $\mathcal{S}_r$, which trains E_{r+1} from E_r. This completes one round of co-evolution.

The two components exchange supervision through their interaction records. Updating the executor changes the behavior against which candidate harnesses are judged, motivating fresh preference collection in the next round. Updating the generator changes the experience available for executor learning. All learning data come from training tasks. After each executor update, we record success on a separate evaluation split under H_0, estimating $J(\theta_r)$ with the same deployment interface. The resulting deployment agent consists of H_0 and the learned executor; the generator is retained for subsequent training rounds.

In our pipeline, $\mathrm{Collect}(G, E, K; C)$ samples K specifications per training task and records executions under their compiled harnesses, with both models frozen. It also evaluates a fixed control C when provided. Pairs applies the preference construction in Section 3.3. $\mathrm{ActionData}$ selects at most one clean successful trajectory per task, choosing the shortest eligible trajectory if needed, and reconstructs its training contexts as in Section 3.4. DPO and SFT use fixed optimization budgets and start from the supplied checkpoints.

4 Experiments

We first describe the experimental setup and report overall performance. Then, we examine generator optimization, and evaluate the contribution of collected experience to executor learning.

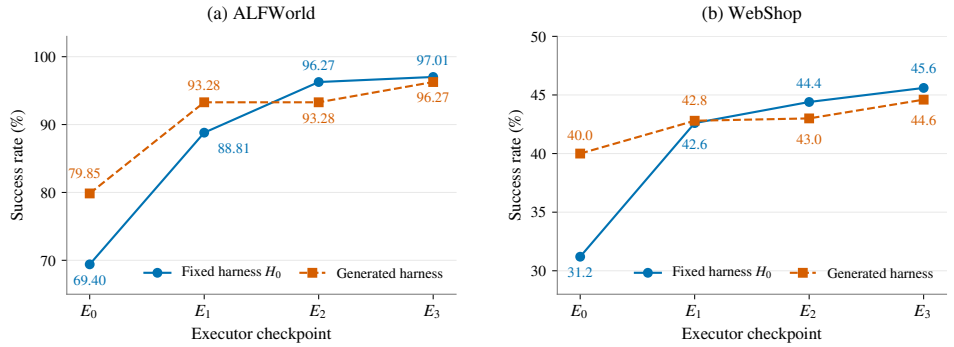

Figure 3: **Fixed-harness performance across executor training stages.** Qwen3.5-9B success on **(a)** ALFWorld official unseen tasks and **(b)** WebShop test tasks. Each point is one evaluation per task with seed 42 and uses the values in Table 1. The ALFWorld E_0 point retains the original evaluation configuration; E_1–E_3 use the later configuration. The panels use different vertical scales.

4.1 Experimental Setup

Benchmarks and data. We evaluate HERO on ALFWorld Shridhar et al. (2020) and WebShop Yao et al. (2022), using fixed training subsets of 500 and 2,000 tasks, respectively. The main ALFWorld experiment evaluates all 140 official seen tasks and 134 official unseen tasks. WebShop uses the full product catalogue, with development indices 500–1499 and test indices 0–499; training tasks are sampled from indices 1500 onward. Its development and test sets contain 1,000 and 500 tasks, respectively, with goal seed 233. Each benchmark is trained separately, and all preference and action supervision comes from its training subset.

Models and training. Our main generator and executor start from the released Qwen3.5-9B checkpoint Qwen Team (2026), with separate LoRA adapters Hu et al. (2021) of rank 16, scaling parameter 32, and zero dropout. Three executor training stages produce E_1, E_2, E_3. We use bfloat16, AdamW, and an effective batch size of 16. Initial generator DPO uses 30 epochs at a learning rate of 2×10^{-5}; later stages use 10 epochs at 2×10^{-6}, with $\beta = 0.1$. Executor SFT uses two epochs per stage, with learning rate 10^{-5} initially and 2×10^{-6} thereafter.

Harness and execution. Each experiment has a fixed deployment harness H_0. On ALFWorld, it supplies task instructions, public state tracking, observation rendering, and the environment action tool. On WebShop, we retain the default Harness-R1 agent loop Shao et al. (2026), including available actions and full interaction history. Generators use schema-constrained decoding with temperature 1.0 and top-p 0.95; executors use temperature zero. Thinking is disabled for both roles. ALFWorld permits at most 50 action attempts and 50 executor calls per episode; WebShop permits 20 decisions.

4.2 Overall Performance

Table 1 compares reported results from 7–9B executors with HERO using Qwen3.5-9B. It records the backbone, executor size, extra teacher supervision, and use of a runtime experience store. Our E_0–E_3 checkpoints are evaluated under the fixed deployment harness, without generator calls. HERO reaches 97.01% unseen success on ALFWorld and 45.60% test success on WebShop.

Comparison with reported results. On ALFWorld, HERO with Qwen3.5-9B reaches 97.01% unseen success, achieving the highest performance among compared methods. On the seen split, HERO reaches 93.57%, below EvoHarness-RL's 96.90%. EvoHarness-SFT and EvoHarness-RL use Claude Opus teacher trajectories and experience consolidation; **HERO learns from its own interactions**. On WebShop, our original executor reproduces the 31.20% default-harness result. With the same Qwen3.5-9B backbone, HERO reaches 45.60%, compared with 42.60% for Agent SFT and 43.00% for Agent SFT plus Harness-R1. Agent SFT uses successful target-agent trajectories; the Harness-R1 engineer additionally uses GPT-5.5-generated editing supervision. HERO exceeds these methods without additional teacher supervision.

Table 1: **Overall performance and supervision sources.** All scores are success rates (%); size denotes the executor's parameters. The upper and lower panels report ALFWorld and WebShop, respectively. External ALFWorld and WebShop results are reported by EvoHarness-RL Ning et al. (2026) and Harness-R1 Shao et al. (2026), respectively. Since exact protocol matching across published systems is difficult, we seek a fair comparison by using similarly sized executors and aligning evaluation settings as closely as possible, while explicitly reporting remaining differences.

(a) ALFWorld

Method	Backbone	Size	Extra teacher	Runtime memory	Seen (140)	Unseen (134)
Reported methods without an additional teacher						
ReAct	Qwen3	8B	No	No	47.90	50.00
Reported methods with additional teacher supervision						
EvoHarness-SFT	Qwen3	8B	Yes	Yes	68.60	69.40
EvoHarness-RL	Qwen3	8B	Yes	Yes	**96.90**	86.60
Reported methods with unspecified teacher provenance						
SkillRL‡	Qwen2.5	7B	NR	NR	89.90	—
ExpeL	Qwen3	8B	NR	Yes	49.30	—
ReasoningBank†	Qwen3	8B	NR	Yes	55.70	—
MemP†	Qwen3	8B	NR	Yes	49.70	—
Dynamic Cheatsheet	Qwen3	8B	NR	Yes	52.10	—
ACE	Qwen3	8B	NR	Yes	51.40	—
SkillOS-base†	Qwen3	8B	NR	Yes	53.10	—
GRPO	Qwen3	8B	NR	NR	65.60	—
SkillOS†	Qwen3	8B	NR	NR	80.20	—
EvoHarness-Base	Qwen3	8B	NR	Yes	56.40	77.60
HERO: self-generated supervision, no additional teacher supervision						
Original model E_0	Qwen3.5	9B	No	No	57.14	68.66
HERO, round $r = 1$, E_1	Qwen3.5	9B	No	No	91.43	88.81
HERO, round $r = 2$, E_2	Qwen3.5	9B	No	No	93.57	96.27
HERO, round $r = 3$, E_3	Qwen3.5	9B	No	No	93.57	**97.01**

(b) WebShop

Method	Backbone	Size	Extra teacher	Runtime memory	Dev (1,000)	Test (500)
Reported methods without an additional teacher						
Default harness	Qwen3.5	9B	No	No	—	31.20
ReAct	Qwen3.5	9B	No	No	—	37.40
Self-Refine	Qwen3.5	9B	No	No	—	29.00
Agent SFT	Qwen3.5	9B	No	No	—	42.60
Reported methods with additional teacher supervision						
Supervised-only engineer	Qwen3.5	9B	Yes	No	—	38.60
Harness-R1	Qwen3.5	9B	Yes	No	—	42.20
Agent SFT + Harness-R1	Qwen3.5	9B	Yes	No	—	43.00
HERO: self-generated supervision, no additional teacher supervision						
Original model E_0	Qwen3.5	9B	No	No	28.90	31.20
HERO, round $r = 1$, E_1	Qwen3.5	9B	No	No	40.90	42.60
HERO, round $r = 2$, E_2	Qwen3.5	9B	No	No	42.00	44.40
HERO, round $r = 3$, E_3	Qwen3.5	9B	No	No	**43.30**	**45.60**

Extra teacher means additional-model supervision for post-training or experience-store consolidation; pretraining and fixed prompts are excluded. Runtime memory means a reusable cross-task experience store, excluding episode history and state tracking. NR means the reviewed source does not establish the field; — denotes an unreported score. †/‡ mark scores quoted by EvoHarness-RL.

Performance across rounds. Figure 3 shows how fixed-harness performance changes across executor updates. On ALFWorld, $E_1 \rightarrow E_2$ raises unseen success from 88.81% to 96.27%, followed by a smaller increase to 97.01% with E_3; seen success remains 93.57% in the final two rounds. On WebShop, the first update raises test success from 31.20% to 42.60%. The later stages add 1.80 and 1.20 points, yielding a total improvement of 14.40%.

Table 2: **Generator optimization with a frozen executor.** All rows use Qwen3.5-9B. Success is in %; Δ is the before-to-after DPO change in percentage points.

Executor	G update	Fixed H_0	Before	After	Δ
ALFWorld: Unseen					
E_1	$G_1 \to G_2$	88.81	87.31	93.28	+5.97
E_2	$G_2 \to G_3$	96.27	93.28	93.28	0.00
WebShop: Test					
E_1	$G_1 \to G_2$	42.60	42.80	42.80	0.00
E_2	$G_2 \to G_3$	44.40	43.00	43.40	+0.40

Table 3: **Teacher data and student execution.** ALFWorld; columns denote teacher sources.

Metric	H_0	G_0	G_2
Collection			
Success (%)	89.40	83.00	89.20
Clean trajectories	388	340	386
Mean steps	19.87	23.44	18.88
Student: Unseen			
Success (%)	91.04	93.28	93.28
Mean steps	17.08	17.06	15.67

4.3 Analysis

Role of generator optimization. HERO aims to improve executor success under H_0. DPO supports this objective by changing how training experience is generated: with the executor frozen, it increases the relative probability of harness specifications whose executions succeed over those that fail. SFT then updates the executor using recorded successful actions. The two updates have distinct learning signals: DPO optimizes execution preferences, while its contribution to subsequent executor learning is assessed after SFT. We therefore examine both immediate execution performance and downstream learning under H_0.

Effect on task execution. Table 2 compares DPO updates at the E_1 and E_2 stages, with each executor held fixed. On ALFWorld, $G_1 \to G_2$ improves E_1's unseen success by 5.97 percentage points to 93.28%, exceeding its 88.81% baseline under H_0. The next update, $G_2 \to G_3$, leaves E_2's success unchanged at 93.28%, below its 96.27% under H_0. On WebShop, the corresponding updates yield no change for E_1 and a 0.40-point increase for E_2. The resulting generated harnesses achieve 42.80% and 43.40%, compared with 42.60% and 44.40% under H_0, respectively. Thus, their execution performance remains close to the fixed harness in these stages. These results show that DPO can improve execution with a frozen executor, although the benefit varies across stages and an updated generator need not outperform H_0.

Controlled teacher-source comparison. We collect once with H_0, naive G_0, and optimized G_2 on the same 500 ALFWorld training tasks, using E_1 for collection and student initialization with matched supervision and SFT budgets (Table 3). G_2 and H_0 yield similar numbers of clean trajectories, but G_2 has shorter average rollouts, including failures. Their supervision differs: after matching and H_0 reconstruction, 59.1% of examples have identical input contexts and action targets, ignoring order. Under H_0, the G_2-source student averages 8.80% fewer steps than the H_0-source student on 119 jointly solved official unseen tasks (12.54 versus 13.75; retrospective comparison). Its 93.28% unseen success matches the naive-source result; the 2.24-point advantage over the H_0-source result remains inconclusive.

5 Conclusion

In this paper, we presented HERO, a framework for harness-executor co-evolution through verifiable environment feedback. HERO alternates DPO for a task-conditioned harness generator with executor self-training on successful action trajectories. Both components are initialized from the same pretrained model and learn from their own interactions, without additional teacher-provided solutions or critiques. With Qwen3.5-9B, the final executors achieve 97.01% success on ALFWorld's official unseen split and 45.60% on the WebShop test set, each using its fixed deployment harness H_0 without generator calls at evaluation. These results highlight harness generation as a way to shape the experience used for executor learning, with the resulting performance gains retained under a fixed deployment harness.

AI USE STATEMENT

In this work, we used generative AI tools solely to improve the clarity and readability of the manuscript and to assist with figure preparation. We did not use generative AI tools for any tasks requiring disclosure under the conference policy or for any other aspects of the research. All AI-assisted text and figures were reviewed and finalized by the authors. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the assistance of generative AI.

REFERENCES

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*, 2026.

Zi-Yi Dou, Cheng-Fu Yang, Xueqing Wu, Kai-Wei Chang, and Nanyun Peng. Re-rest: Reflection-reinforced self-training for language agents. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15394–15411, 2024.

Wenbo Gao, Zhaomou Song, Zhiyuan Ji, Renxi Liu, Xing Li, Xianzhi Yu, Xiaoguang Li, James Chung-wai Cheung, Weizhe Lin, and Yaoyuan Wang. Experience funnel: A state-policy alternating loop for self-evolving agents. *arXiv preprint arXiv:2609.08919*, 2026.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Haechan Kim, Yoonho Lee, Gisang Lee, Chelsea Finn, and Kangwook Lee. Whale: A simple recipe for joint harness-weight optimization. *arXiv preprint arXiv:2609.00196*, 2026.

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.

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.

Xuying Ning, Dongqi Fu, Tianxin Wei, Hanqing Zeng, Yuanchen Bei, Bingxuan Li, Zihao Li, Qifan Wang, Xiang Shen, Yifan Wu, et al. Evoharness-rl: Learning self-evolving runtime harness for long-horizon llm agents. *arXiv preprint arXiv:2608.05446*, 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*, volume 2025, pp. 79791–79821, 2025.

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

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

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.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

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.

Dilara Soylu, Christopher Potts, and Omar Khattab. Fine-tuning and prompt optimization: Two great steps that work better together. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 10696–10710, 2024.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

Z Xi, Y Ding, W Chen, B Hong, H Guo, J Wang, D Yang, C Liao, X Guo, W He, et al. Agentgym: Evolving large language model-based agents across diverse environments. arxiv 2024. *arXiv preprint arXiv:2406.04151*.

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

Zhou Yu, Bin Bi, Shiva Kumar Pentyala, Shubham Mehrotra, Sougata Chaudhuri, Shilpa Bhagavath, Zeyuan Chen, Ran Xu, Phil Mui, James Zhu, et al. Co-evolving harnesses and models: On-policy correction helps weaker models catch up where imitation fails. *arXiv preprint arXiv:2609.09134*, 2026.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. *Advances in Neural Information Processing Systems*, 38:113959–113991, 2026.

Adam Zweiger, Jyo Pari, Han Guo, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. *Advances in Neural Information Processing Systems*, 38:74084–74115, 2026.