ENVHARNESS: AWAKENING STATIC WORLDS FOR AGENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents learn by interacting with environments, yet these environments are hand-built and static: blind to an agent's weaknesses, and quickly left behind as it improves. While recent environment generation methods attempt to address this, they require domain-specific pipelines, rely on expensive or unreliable verifiers, and still produce static environments. Instead of rebuilding environments from scratch, we propose Environment Harness (ENVHARNESS), a programmable layer of plug-in components that wraps a static environment to reshape its behavior without modifying the underlying logic. Operating through standard interfaces, ENVHARNESS applies across diverse domains while ensuring every customized environment retains its original verifier. To automate this process, we introduce ENVRIGGER, which treats the target policy as a black box, observing its execution trajectories to synthesize ENVHARNESS components targeting diagnosed flaws, and validating them via fresh rollouts. Across five benchmarks in four domains, agents that learn skills from ENVHARNESS environments outperform those learning from original environments or domain-specific environment generation pipelines, achieving up to a 9.0-point improvement with 9.8% fewer execution steps. ENVHARNESS also provides a superior optimization signal for reinforcement learning, enabling continuous, targeted co-evolution of the policy and the environment.

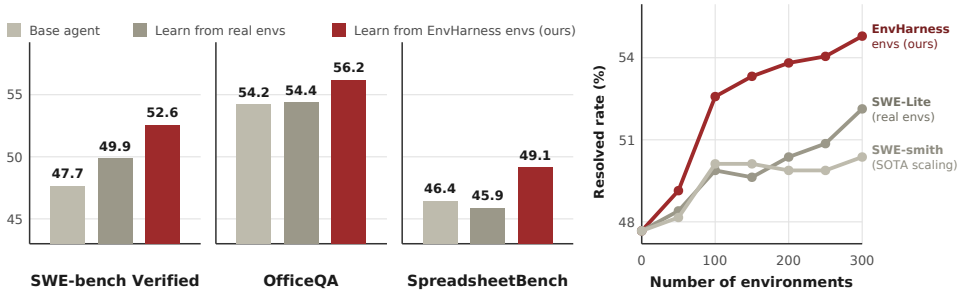

Figure 1: **Overall performance.** *Left:* Agents learning from ENVHARNESS environments consistently outperform those learning from the original environments across software engineering and office automation benchmarks, including SWE-bench Verified, OfficeQA, and SpreadsheetBench. *Right:* On SWE-bench Verified, under an identical environment budget, ENVHARNESS keeps improving as environments scale, while real and generated environments flatten out.

1 INTRODUCTION

As LLMs are deployed as autonomous agents, the source of learning shifts from curated text data to interactive environments. Whether navigating web pages (Gur et al., 2024), resolving issues in a codebase (Yang et al., 2024), or controlling an embodied platform (Wang et al., 2023), agents rely on their respective environments to acquire learning signals (Yao et al., 2022b). These environments act as interactive counterparts that present specific tasks, manage changing states, respond to actions, and evaluate success (Li et al., 2026). Unfortunately, building them requires substantial

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.

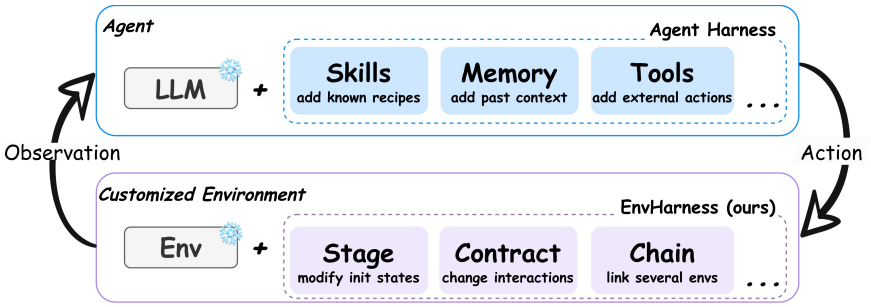

Figure 2: While an agent harness enhances a frozen LLM with plug-in components without modifying weights, ENVHARNESS mirrors this paradigm for environments, customizing a frozen environment while preserving its original logic.

human effort to hardcode the interaction logic and verifiers (Zhang et al., 2025; Merrill et al., 2026). Consequently, the resulting environments remain rigidly static, behaving identically regardless of which agent interacts with them or how much that agent has improved (Hu et al., 2026). This rigidity limits agent learning in two fundamental ways: it fails to provide a targeted signal that addresses a particular agent's weaknesses (Dennis et al., 2020; Jiang et al., 2021), and it has nothing more to teach once the agent learns to solve the existing tasks (Wang et al., 2019; Beukman et al., 2024).

Since manually building environments is expensive, emerging work has turned to automated environment generation (Zala et al., 2024; Guo et al., 2025; Song et al., 2026). Despite its scalability, this approach suffers from two limitations. First, generation pipelines are inherently domain-specific. Existing systems generate environments for web navigation (Trabucco et al., 2025), programming (Pan et al., 2024), or tool use (Lee et al., 2026a), but a pipeline built for one setting cannot transfer to the others. Second, ensuring correctness is both costly and unreliable. Because these environments and verifiers are generated by LLMs, practitioners must over-generate and heavily filter them (Wang et al., 2026; Yang et al., 2026a), which still cannot fully guarantee their correctness.

Rather than creating new environments to obtain learning signals, we propose Environment Harness **(ENVHARNESS)**, a programmable layer that transforms an existing, static environment into a dynamically customized one without modifying the environment itself. We make an analogy between ENVHARNESS and agent harness (Anthropic, 2025; 2026b; OpenAI, 2026) in Figure 2: An agent harness provides LLMs with external memory, tools, and skills to handle complex tasks beyond basic text generation. ENVHARNESS extends this concept to environments, equipping a static environment with modular, plug-in components. A `Stage` sets the starting point of an episode, a `Contract` controls the allowed actions and observations, and a `Chain` connects multiple base environments to form an extended episode. The agent continues to use the standard interface, but ENVHARNESS mediates this interaction. This enables a single environment to fulfill needs it was never built for: isolating a specific skill, extending a task's horizon, or calibrating difficulty so the agent struggles but ultimately succeeds. Crucially, operating strictly at the interface level makes ENVHARNESS domain-agnostic, while allowing every new environment to safely inherit the trusted, human-built verifier of its original environment.

While ENVHARNESS provides a universally applicable framework, its specific configuration must be tailored to each target policy and task. To automate this customization process, we introduce ENVRIGGER. By treating the policy as a black box, ENVRIGGER observes both successful and failed execution trajectories within the base environment to diagnose specific behavioral vulnerabilities. Guided by these findings, it synthesizes candidate ENVHARNESS components and wraps the current environment to evaluate them. ENVRIGGER then runs fresh policy rollouts, judging acceptance solely on whether the candidate environment cultivates the missing capabilities while remaining solvable. Components failing this evaluation undergo iterative revision until they succeed. Ultimately, this workflow realizes fully automated, task-policy-conditioned environment customization.

We conduct experiments on challenging benchmarks spanning embodied tasks (ALFWorld (Shridhar et al., 2020)), web browsing (WebArena (Zhou et al., 2024)), software engineering (SWE-bench Verified (Jimenez et al., 2024)), and office work (OfficeQA (Databricks, 2025), Spreadsheet-Bench (Ma et al., 2024)). Specifically, we evaluate ENVHARNESS across two representative learning paradigms: skill learning, where a frozen policy distills reusable skills from its trajectories

and retrieves them at inference time, and reinforcement learning (RL), where the policy's weights are updated. With skill learning, agents that learn from ENVHARNESS environments outperform those learning from original environments, achieving up to 9.0 points of improvement on held-out tasks (Table 2) while using 9.8% fewer interaction steps (Table 3). In RL settings, ENVHARNESS-customized environments similarly yield significantly stronger policies, with up to 6.5 points of improvement (Table 4). Finally, repeatedly executing the ENVRIGGER loop enables continuous co-evolution between the agent and its environment, unlocking compounding performance gains that scale effectively with the number of customized tasks.

2 ENVHARNESS

2.1 THE ENVHARNESS PARADIGM

An *agent harness* (Anthropic, 2025; 2026b; OpenAI, 2026) is the software layer (execution loops, tool registries, and context management) (Meng et al., 2026) that wraps an LLM to form an autonomous agent (Agent = Model + Harness). It adds new capabilities without changing the model weights. We apply the same idea to the other side of the agent-environment loop. We define **ENVHARNESS** as a programmable layer that wraps an existing, static environment and turns it into a customizable one (Customized Env = Static Env + ENVHARNESS). ENVHARNESS achieves this customization by modifying the information flow through the standard interface, leaving the underlying environment completely untouched. Analogous to the tools and memory of an agent harness, ENVHARNESS is assembled from modular plug-in components that tailor the environment to specific needs like isolating a specific skill, extending a task's horizon, or adjusting task difficulty.

Table 1: Analogy between Agent Harness and ENVHARNESS. Both scale capabilities through external layers rather than changing the core system.

	Agent Harness	**ENVHARNESS (Ours)**
Base System	Frozen LLM	Static environment
Designed to Solve	Lack of action, memory, or loops	Hardcoded interaction logic
Harness Layer	Capabilities (tools, memory)	Customization (states, rules, observations)
Unified Output	Autonomous agent	Customized environment

Formal Definition of ENVHARNESS. We model an environment as a tuple $E = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, s_0)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ the action space, $\mathcal{O}$ the observation space, $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ the transition function that maps a state and an action to the next state, R the reward induced by the verifier, and s_0 the initial state. An ENVHARNESS component is an environment-agnostic transformation w:

$$E' = w(E), \qquad E' = (\mathcal{S}', \mathcal{A}', \mathcal{O}', T', R', s_0'). \tag{1}$$

w reshapes the environment strictly at the interface level, without modifying its underlying simulator backend or implementation details. It customizes the initial state (s_0'), filters exposed spaces ($\mathcal{A}', \mathcal{O}'$), and updates transition mechanics (T'). Because all interventions remain external, the ground-truth evaluation logic is preserved, so every customized environment reuses the original verifier and no new verifier needs to be written or validated.

2.2 THREE ENVHARNESS COMPONENTS

The mapping w of equation 1 defines a general interface, and any transformation that follows this interface is a valid ENVHARNESS component. In this work, we introduce three concrete types of components, chosen to cover three fundamental modes of environment customization, and we expect more to follow. Each type is specified by its own parameters, overriding standard environment interface methods such as `reset` or `step`. Figure 3 compares the three component types with the original interface, highlighting the specific method each component overrides. We detail the software implementation of this interface protocol and class architecture in Appendix C. Throughout this subsection, we illustrate the components on one ALFWorld task, ***"put a clean mug on the desk"***, where the default instance leaves the mug in the open and ends immediately upon placement.

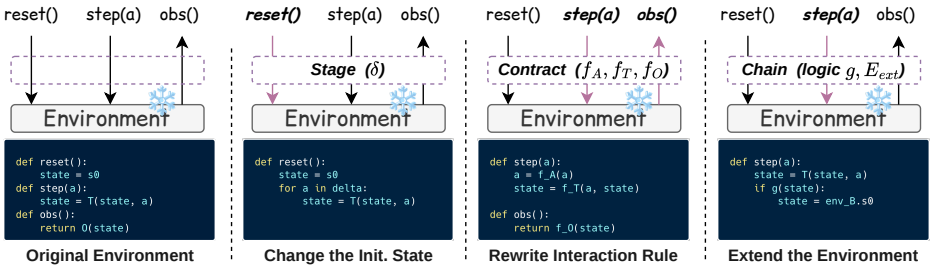

Figure 3: **Overview of ENVHARNESS components wrapping the standard environment interface.** The underlying base environment (native state transitions and original task verifier) remains completely frozen. From left to right: the base environment, followed by three ENVHARNESS components—`Stage`, `Contract`, and `Chain`. Highlighted arrows and headers indicate overridden interface methods, with code blocks showing how each wrapper modifies state initialization, transition dynamics, or observation handling without altering the base environment.

`Stage`: changing the initial state. A `Stage`, $w_{\text{stage},\delta}$, is specified by a sequence of state-modifying actions $\delta = (a_1, \ldots, a_k)$. These actions are applied to the initial state s_0 produced by `reset()`:

$$E' = w_{\text{stage},\delta}(E) = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, s_0'), \quad \text{where } s_0' = T\left(\cdots T\left(T(s_0, a_1), a_2\right)\cdots, a_k\right). \tag{2}$$

Under this transformation, only the initial state changes. A `Stage` customizes the agent's starting point, either introducing obstacles that require specific skills to tackle or completing early subgoals in advance to shorten the task horizon. For example, in our running task, one `Stage` executes the following sequence of actions: `take mug 1`, `open drawer 1`, `put mug 1 in drawer 1`, and `close drawer 1` on the state returned by `reset()`. This `Stage` intentionally hides the mug from the agent, forcing the agent to face a more challenging scenario: search for the object first instead of directly reaching for it in plain sight. Conversely, another `Stage` can simplify the setup by executing `cleaning the mug` in advance, leaving only the final placement steps.

`Contract`: rewriting the interaction. A `Contract`, $w_{\text{contract},r}$, is specified by a triplet of transformation maps $r = (f_A, f_T, f_O)$, each defaulting to the identity. These maps rewrite the action space, transition dynamics, and observation space, respectively:

$$E' = w_{\text{contract},r}(E) = (\mathcal{S},\ \mathcal{A}',\ \mathcal{O}',\ T',\ R,\ s_0), \quad \text{where } (\mathcal{A}', T', \mathcal{O}') = \big(f_A(\mathcal{A}), f_T(T), f_O(\mathcal{O})\big). \tag{3}$$

In practice, these transformations enforce action preconditions, augment or mask observations, and attach structured feedback to specific outcomes to steer agent learning. For example, on our running task, one `Contract` configures f_O to truncate the room description after the first two sentences, requiring the agent to build its spatial representation across several steps; another `Contract` uses f_T to make the `clean mug` action fail with an explanatory message if the agent is not holding the mug, forcing the agent to pick up the object first; and a third `Contract` configures f_A to remove high-level teleport navigation commands, forcing the agent to move and search step by step.

`Chain`: extending the environment. A `Chain`, $w_{\text{chain},\ell}$, is specified by a pair $\ell = (E_{\text{ext}}, g)$, where E_{ext} is an additional environment and g is a composition logic which combines the original environment E and E_{ext} into a composite environment E' exposed through the same interface:

$$E' = w_{\text{chain},\ell}(E) = (\mathcal{S}', \mathcal{A}', \mathcal{O}', T', R', s_0'), \quad \text{where } E' = g(E, E_{\text{ext}}). \tag{4}$$

To allow cross-environment combinations, the new spaces are simply the union of the base environments (e.g., $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_{\text{ext}}$), and R' acts as the new composite reward. The composition logic g is unrestricted, allowing environments to be concatenated, interleaved, or branched dynamically based on intermediate outcomes (see Appendix D for examples). The composition logic g can either combine both environments from the start, or use the transition function T' to dynamically transition from one environment to the next once a specific condition is satisfied. For our running environment, one `Chain` appends *"heat a potato and put it on the countertop"* in the same house, returning success under R' only when both environments are verified. This requires the agent to learn to carry its goal past the point where it would otherwise have stopped.

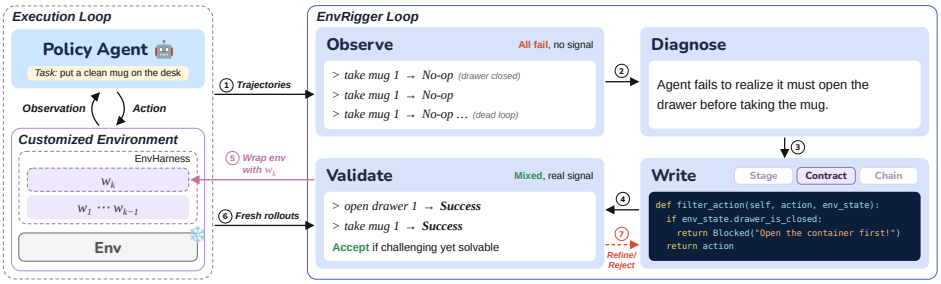

Figure 4: **Overview of ENVRIGGER.** The policy agent acts in a customized environment, a frozen base environment wrapped by ENVHARNESS components. On each task, ENVRIGGER observes the policy's trajectories, diagnoses its weaknesses, and writes a candidate component. Fresh rollouts in the customized environment validate it: the candidate is accepted if the environment is challenging yet solvable, yielding a mixed, real learning signal; otherwise, it is refined or rejected and rewritten.

Composition. Because all ENVHARNESS components share a standard interface, they compose freely. For example, stacking a `Stage`, a `Contract`, and a `Chain` on our base mug task yields a single composite environment:

$$E' = w_{\text{chain},\ell}\left(w_{\text{contract},r}\left(w_{\text{stage},\delta}(E)\right)\right). \tag{5}$$

In this setup, $w_{\text{stage},\delta}$ initializes the episode with the mug hidden in a drawer to enforce spatial search; $w_{\text{contract},r}$ truncates the observation to two sentences to impose partial observability; and $w_{\text{chain},\ell}$ appends a follow-up task to require goal persistence. Note that these transformations are not commutative in general ($w_1 \circ w_2 \neq w_2 \circ w_1$); the nesting order determines how the resulting environment is constructed and which constraints apply during initialization versus active interaction.

3 ENVHARNESS FOR AGENT LEARNING

3.1 PROBLEM SETUP

Given a base environment E supporting a set of base tasks, and a target policy agent π, our objective is to automatically generate a modified environment E' tailored to a specific task t, exposing the unique flaws of π to facilitate targeted policy improvement. A single ENVHARNESS component is itself policy-agnostic, as equation 1 defines w as a transformation of the environment alone, allowing the same component to be applied without modification to any policy. Conversely, the selection and parameterization of these components must be conditioned on both the base task t and the observed behavior of the policy π. We therefore introduce the task-policy-conditioned map $\mathcal{H}$:

$$E' = \mathcal{H}(E, t; \pi) = (w_k \circ w_{k-1} \circ \cdots \circ w_1)(E), \tag{6}$$

where each w_i is a customized ENVHARNESS component designed to wrap the base environment E and expose the critical weaknesses of π on task t. Rather than inspecting internal model weights, these components treat the agent as a black box and operate solely on its outputs to generate a steady, corrective training signal.

3.2 ENVRIGGER

We introduce ENVRIGGER to realize the task-policy-conditioned map $\mathcal{H}$ of equation 6. ENVRIGGER runs π in the environment on task t, analyzes the resulting trajectories, writes specific ENVHARNESS components to customize the environment for this task, and validates the customized environment using fresh policy rollouts. Candidates from customized environments that provide an appropriate learning signal are accepted, while the ENVHARNESS components of unsuccessful candidates are rejected or revised. Figure 4 illustrates this complete workflow, in which ENVRIGGER operates systematically through four distinct stages: Observe, Diagnose, Write, and Validate. To ensure that the initial state mutations introduced by `Stage` can be reliably reproduced, we assume the base environment supports deterministic resets during the validation phase.

Observe. The ENVRIGGER begins by running the policy π on the base task t in the current environment to collect and analyze a batch of rollout trajectories. While failures expose the specific weaknesses to be addressed within this task, successes help define the boundaries of these flaws, showing which capabilities are already intact and where they begin to fail.

Diagnose. ENVRIGGER analyzes the collected trajectories to identify the root causes of the observed behaviors, focusing on systemic issues such as repetitive action loops, failures in parsing long observations, or misread tool constraints. This diagnosis also determines the customization direction. For a struggling policy, the goal is to scaffold missing steps and simplify the task. Conversely, if the policy achieves a perfect success rate, it indicates the current environment is too forgiving to expose any remaining weaknesses. The ENVRIGGER diagnoses that the environment must be made harder, shifting the customization to inject more challenging scenarios that force the policy's potential flaws into the open. ENVRIGGER then outputs these findings as a textual diagnosis.

Write. Based on the diagnosis, ENVRIGGER synthesizes one or more ENVHARNESS components to target the identified flaws. A single flaw may require combining multiple components, such as a `Stage` that customizes the initial state and a `Contract` that filters subsequent interactions, emitted together as a candidate set. For example, if the diagnosis reveals that the policy relies on a fragile shortcut that bypasses learning, ENVRIGGER can write a `Contract` that blocks this action under specific conditions, forcing the policy to explore and master the intended skills.

Validate. To evaluate the candidate components from the Write stage, ENVRIGGER wraps the current environment with them to instantiate E' and runs fresh rollouts of π on the base task t. Based on some trajectory metrics, such as the success rate and failure distribution, ENVRIGGER analyzes these fresh rollouts to decide between three validation behaviors: accepting the candidate, rejecting unsolvable or non-challenging ones, or refining candidates with poorly scaled signals. If the candidate requires refinement, the trajectories and scaling feedback flow back into the Write stage, repeating this Write-and-Validate loop until a candidate is accepted or the revision budget is exhausted. All accepted components are ultimately added to the ENVHARNESS. The exact system prompt and decision-making criteria guiding these validation actions are detailed in Appendix A.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

In this section, we mainly focus on skill learning, where the policy's weights stay frozen and skills are extracted from its trajectories in each environment. We also evaluate the compatibility and performance of our framework under the online reinforcement learning paradigm, which is presented as part of our broader analysis in Section 5.

Benchmarks and Evaluation. We evaluate our framework on five benchmarks spanning four distinct domains: ALFWorld (Shridhar et al., 2020) for text-based embodied environments, WebArena (Zhou et al., 2024) for web interaction, SWE-bench Verified (Jimenez et al., 2024) for software engineering, and OfficeQA (Databricks, 2025) with SpreadsheetBench (Ma et al., 2024) for office automation. We report each benchmark's native metrics, additionally tracking the average steps on SWE-bench Verified as a measure of execution efficiency. Training and evaluation episodes are strictly disjoint on every benchmark, with detailed split configurations provided in Appendix E.1.

Importantly, ENVRIGGER and the policy agent utilize the same model backbone on each benchmark: Gemini-3.1-Flash-Lite for ALFWorld and WebArena, and Gemini-3.5-Flash elsewhere, ensuring that performance gains do not stem from distilling a stronger external model. On each training set, ENVRIGGER executes the optimization loop of Section 3 to generate ENVHARNESS-customized environments. We then extract skills from trajectories collected in these environments following ReasoningBank (Ouyang et al., 2025), and evaluate the skill-equipped policy agent on the held-out instances. We exclude the `Chain` component from this automated pipeline, since ENVRIGGER does not know what other environments contain and thus cannot choose which one to join. Instead, we analyze the effect of chaining separately in Appendix D and Appendix F.3.

Baseline Methods. We compare ENVHARNESS against three types of skill sources: **No Skills** (the frozen policy agent alone), **Original Envs** (skills from original environments to isolate reshaping effects), and domain-specific environment generators (**GenEnv** (Guo et al., 2025), **VeriEnv** (Chae et al., 2026), and **SWE-smith** (Yang et al., 2026a)). While these baseline generators are domain-specific, ENVHARNESS applies generically across all domains via a unified interface, including the office automation environments where no generation baseline exists. To ensure a fair comparison, all baselines share the same seed instances, environment count, extraction pipeline, and policy model. ENVRIGGER operates strictly on training episodes under the same oracle verification access, and each evaluation instance is attempted once. Baseline details are in Appendix E.2.

4.2 MAIN RESULTS

Table 2 and Table 3 summarize the primary results across five benchmarks. Based on these evaluations, we present the following key observations.

Table 2: Performance of skill-equipped agents across environment sources on ALFWorld and WebArena (mean and std over 3 runs). The last row reports the improvement of ENVHARNESS Envs over Original. "–" denotes benchmark-specific baselines that cannot be applied across domains.

Skill Source	ALFWorld			WebArena				
	In-Dist	OOD	Avg.	Reddit	Shopping	Shop Admin	GitLab	Avg.
No Skills	62.6 $_{1.7}$	60.7 $_{5.2}$	61.7 $_{3.4}$	39.6 $_{2.3}$	35.2 $_{3.3}$	44.1 $_{2.3}$	35.8 $_{8.4}$	38.7 $_{2.3}$
Original Envs	63.3 $_{2.8}$	61.4 $_{4.3}$	62.4 $_{3.4}$	38.7 $_{9.7}$	35.2 $_{1.3}$	44.6 $_{3.0}$	35.4 $_{4.0}$	38.5 $_{3.1}$
GenEnv	63.3 $_{1.2}$	61.9 $_{2.7}$	62.6 $_{1.9}$	–	–	–	–	–
VeriEnv	–	–	–	39.6 $_{4.2}$	30.2 $_{0.0}$	49.7 $_{2.4}$	**38.9** $_{5.6}$	39.6 $_{1.4}$
ENVHARNESS Envs	**66.2** $_{0.3}$	**70.4** $_{2.3}$	**68.3** $_{1.3}$	**40.6** $_{4.7}$	**37.4** $_{0.3}$	**50.8** $_{1.5}$	37.7 $_{3.1}$	**41.6** $_{1.8}$
Improvement	+2.9	+9.0	+5.9	+1.9	+2.2	+6.2	+2.3	+3.1

Table 3: Performance of agents equipped with skills extracted from different environment sources on SWE-bench Verified, OfficeQA, and SpreadsheetBench. Standard deviations are gray subscripts. "–" denotes that the method is benchmark-specific and cannot be applied to the other domain.

Skill Source	SWE-verified		OfficeQA		SpreadsheetBench	
	SR (↑)	Average Step (↓)	EM (↑)	F1 (↑)	Pass@1 (↑)	Mean Score (↑)
No Skills	47.67 $_{0.93}$	53.58 $_{2.93}$	54.23 $_{2.84}$	55.77 $_{2.98}$	46.44 $_{0.15}$	61.32 $_{0.37}$
Original Envs	49.88 $_{2.59}$	55.01 $_{1.69}$	54.40 $_{1.84}$	55.77 $_{1.59}$	45.88 $_{1.19}$	61.47 $_{0.59}$
SWE-smith	50.12 $_{1.74}$	54.72 $_{2.03}$	–	–	–	–
ENVHARNESS Envs	**52.58** $_{2.72}$	**49.61** $_{2.49}$	**56.20** $_{2.34}$	**57.73** $_{2.29}$	**49.15** $_{0.36}$	**62.48** $_{0.27}$
Improvement	+2.70	+5.40	+1.80	+1.96	+3.27	+1.01

ENVHARNESS delivers consistent performance and efficiency gains over static environments. Skills acquired in environments customized by ENVHARNESS consistently outperform those extracted from original environments on every benchmark, yielding up to a 9.0-point improvement on ALFWorld. Conversely, extracting skills from static base environments can actually degrade performance; for instance, on SpreadsheetBench, skills from unmodified environments fall below the no-skill baseline, while on SWE-bench Verified, they lengthen execution trajectories to 55.0 steps. In contrast, ENVHARNESS reduces the average steps per episode on SWE-bench Verified to 49.6. This efficiency gain directly correlates with specific diagnostics from ENVRIGGER, which generates targeted `Contracts` and `Stages` to disrupt repetitive action loops and filter verbose observations, successfully shortening execution trajectories.

ENVHARNESS generalizes across domains through a domain-agnostic interface. The underlying interface protocol, ENVRIGGER loop, and the skill extraction pipeline apply consistently across all five benchmarks, requiring only domain-specific prompt templates to adapt to different environments. By contrast, specialized generation baselines are constrained to their specific target benchmarks (indicated by dashes for inapplicable domains in Tables 2 and 3). Nevertheless, ENVHARNESS consistently outperforms these specialized baselines wherever they can be applied. On ALFWorld, ENVHARNESS skills surpass GenEnv by 5.7 points on average and by 8.5 points in

out-of-distribution settings, where generic instance generation merely increases repetitive practice without addressing policy weaknesses. On SWE-bench Verified, ENVHARNESS outperforms the purpose-built SWE-smith by 2.46 points in success rate while requiring 5.11 fewer execution steps per episode. Targeting diagnosed vulnerabilities through a unified interface thus proves superior to merely scaling up the quantity of training episodes through domain-specific generation.

5 ANALYSIS

We analyze ENVHARNESS along three dimensions: its generalization across policy models of different families and strengths, its effectiveness as a training signal for online reinforcement learning, and its scaling efficiency compared with unmodified and synthesized environments. Additional analyses, including held-out skill generalization, compute overhead, and targeted environment reshaping, are reported in Appendix G.

ENVHARNESS generalizes across different LLM backbones. To evaluate generalizability, we test four distinct models on SWE-bench Verified: Gemini 3.1 Flash-Lite, Qwen3.6 27B (Qwen Team, 2026), Gemini 3.5 Flash, and Claude Sonnet 4.6 (Anthropic, 2026a). These span open-weight and proprietary models across a wide capability range. In each setting, the target policy and ENVRIGGER share the same backbone, and the extraction pipeline and protocol are unchanged.

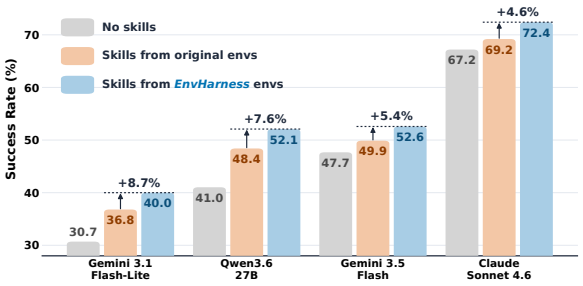

Figure 5: Cross-model results on SWE-bench Verified. Each group represents one policy model, ordered from weakest to strongest. Bars show success rates with no skills, with skills from unmodified environments or ENVHARNESS environments. Percentages represent relative gains over original environments.

Figure 5 shows the results. ENVHARNESS skills outperform real environment skills on all four policies, by 2.7 to 3.7 absolute points, even though the skill-free success rates span a broad range from 30.7 to 67.2. The size of this gain is largely independent of how strong the underlying policy is: the customization loop neither breaks down on the weakest model nor saturates on the strongest, and the same pipeline, prompts, and acceptance criteria are used throughout. What the policy's capability level appears to change is the content of the diagnoses rather than the applicability of the loop. We note that skills of either kind help the two weakest policies most relative to no skills at all.

ENVHARNESS enables better RL. Beyond skill learning, we investigate whether ENVHARNESS-customized environments can provide active training signals in online reinforcement learning. We perform this analysis on ALFWorld and WebShop (Yao et al., 2022a), a simulated online shopping environment that we use only for this RL study, using Qwen3-8B-base (Yang et al., 2025) as the policy, optimized via Group Relative Policy Optimization (GRPO) (Shao et al., 2024); training details are provided in Appendix F.1. We train two distinct policies: one trained solely on the original static environments, and one trained entirely on environments reshaped by ENVHARNESS, evaluating both on the same held-out instances. Table 4 reports the results. Training on customized environments improves ALFWorld in-distribution success by 6.5 points and both WebShop metrics, with a slight trade-off in ALFWorld OOD success (88.8 vs. 89.6).

	ALFWorld			WebShop	
Training Set	**In-Dist**	**OOD**	**Avg.**	**Score**	**SR**
Original Envs	81.4	**89.6**	85.5	75.6	66.0
ENVHARNESS Envs	**87.9**	88.8	**88.4**	**79.2**	**67.4**

Table 4: Reinforcement learning on ALFWorld and WebShop, comparing policies trained on the original environments and on ENVHARNESS environments. ALFWorld is scored by success rate on in-distribution and held-out instance types; WebShop reports environment score and success rate.

ENVHARNESS enables efficient environment scaling. Environment scaling evaluates performance as the available training environments expand, comparing three allocation strategies under an identical budget: ENVHARNESS environments, unmodified benchmark environments, and SWE-smith generated environments. Holding the policy model, environment budget, and skill retrieval protocol fixed, each batch of 50 environments yields one skill bank (alternating between 2 and 3 skills per bank, totaling 15 skills at 300 environments). Crucially, while both baselines draw environment batches independently of the learner, ENVHARNESS synthesizes each batch specifically targeting the policy equipped with previously accumulated skills, enabling the environments and the policy to co-evolve. As shown in Figure 6, ENVHARNESS climbs from 47.67 to 54.79 (a 7.12-point gain) and maintains an upward trajectory at 300 environments. In contrast, the same budget yields only 52.13 on original environments and 50.37 on generated ones. This performance gap confirms that targeting the learner's current capability boundary is fundamentally more effective than unconditioned environment scaling. Representative skills are provided in Appendix F.2.

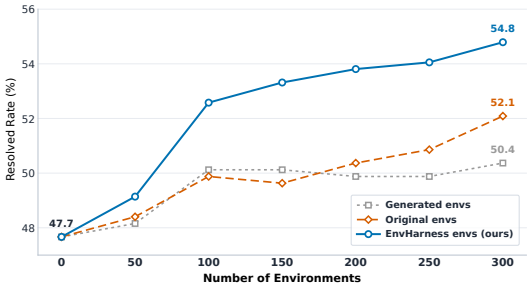

Figure 6: Environment scaling on SWE-bench Verified. All three sources supply the same number of environments and feed the same extraction and retrieval protocol.

6 RELATED WORK

6.1 ENVIRONMENT SCALING

Environment scaling supplies agents with more learning environments (Huang et al., 2025b; Xi et al., 2025). Existing methods include LLM-simulated feedback (Guo et al., 2025; Zala et al., 2024; Wang et al., 2025), generative world models (Zuo et al., 2026; Wang et al., 2026), programmatic synthesis (Chae et al., 2026; Sun et al., 2026; Dong et al., 2026), and intra-benchmark task generation (Yang et al., 2026a; Pan et al., 2024). Other approaches adapt environment presentation via RL curriculum generation (Jiang et al., 2021; Dennis et al., 2020; Wang et al., 2019; Liu et al., 2026) or hand-designed reward shaping (Lu et al., 2025). Unlike benchmark-specific pipelines or hand-designed curricula, ENVHARNESS reshapes existing environments through a unified interface to target the policy's diagnosed weaknesses, keeping original tasks and verifiers.

6.2 SELF-EVOLVING AGENT

Self-evolving agents improve autonomously from experience (Fang et al., 2025). Recent efforts evolve various agent components, including prompts (Shinn et al., 2023; Madaan et al., 2023), skill libraries (Wang et al., 2023; 2024; Xia et al., 2026; Huang et al., 2026b), experience memory (Zhao et al., 2024; Ouyang et al., 2025), model weights via self-generated tasks (Yuan et al., 2024; Huang et al., 2025a; Zhao et al., 2026; Xia et al., 2025; He et al., 2025; Huang et al., 2026a), and agent harnesses (Lee et al., 2026b). While these methods evolve the agent against a fixed world, ENVHARNESS inversely reshapes the environment itself against a frozen policy's diagnosed weaknesses.

7 CONCLUSION

We introduce ENVHARNESS, a programmable layer that turns a static environment into a controllable one. ENVHARNESS wraps a frozen benchmark with three plug-in components (`Stage`, `Contract`, and `Chain`) and reshapes it entirely through the standard `reset`/`step` interface to isolate skills, extend horizons, or calibrate difficulty. Since ENVHARNESS never touches internal code, a single implementation works seamlessly across domains while safely retaining the original, human-built verifiers. To fully automate this, ENVRIGGER diagnoses policy weaknesses from trajectories and synthesizes targeted components to yield precise learning signals. This reframes environment construction as a wrapping problem rather than an authoring one, providing a scalable pathway for agent learning.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods and assist with translation. We have not used generative AI tools to generate synthetic data sets, develop theoretical models or conceptual frameworks, design research methodology or experiments, or interpret results, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for editing the research paper to improve readability, creating or editing software code, sourcing information, and identifying relevant literature. We have reviewed all AI-assisted work. Specifically, for software development, the authors manually authored the design documents and provided effective unit tests; the core components of the AI-generated code implementations were subsequently manually inspected by the authors to guarantee correctness. All AI-edited text was manually reviewed word-by-word to ensure accuracy and preserve the intended scientific meaning. For literature retrieval, the authors manually read and verified all referenced papers to prevent hallucinations. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To ensure full reproducibility of our results, we have thoroughly documented all experimental configurations, model choices, and evaluation protocols throughout the paper and the supplementary appendix. The overarching experimental setup is described in Section 4.1. Comprehensive details regarding benchmark dataset splits, baseline method configurations, and ENVRIGGER hyperparameters are provided in Appendix E. The exact system prompt used for the designer agent is fully disclosed in Appendix A, while the software interface protocols and concrete code implementations for the environment components (`Stage`, `Contract`, and `Chain`) are documented in Appendices C and D. Furthermore, detailed hardware configurations and specific hyperparameters for the reinforcement learning experiments are listed in Appendix F.1.

REFERENCES

Anthropic. Effective harnesses for long-running agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents, 2025. Accessed: 2026-07-07.

Anthropic. Claude 4.6 sonnet. https://www.anthropic.com/news/claude-sonnet-4-6, 2026a. Accessed: 2026-07-28.

Anthropic. Harness design for long-running application development. https://www.anthropic.com/engineering/harness-design-long-running-apps, 2026b. Published: March 24, 2026. Accessed: 2026-07-07.

Michael Beukman, Samuel Coward, Michael Matthews, Mattie Fellows, Minqi Jiang, Michael Dennis, and Jakob Foerster. Refining minimax regret for unsupervised environment design. *arXiv preprint arXiv:2402.12284*, 2024.

Hyungjoo Chae, Jungsoo Park, and Alan Ritter. Safe and scalable web agent learning via recreated websites. *arXiv preprint arXiv:2603.10505*, 2026.

Databricks. Officeqa: A grounded reasoning benchmark, 2025.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in neural information processing systems*, 33:13049–13061, 2020.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. Genenv: Difficulty-aligned co-evolution between llm agents and environment simulators. *arXiv preprint arXiv:2512.19682*, 2025.

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. In *International Conference on Learning Representations*, volume 2024, pp. 52690–52717, 2024.

Yicheng He, Chengsong Huang, Zongxia Li, Jiaxin Huang, and Yonghui Yang. Visplay: Self-evolving vision-language models from images. *arXiv preprint arXiv:2511.15661*, 2025.

Yihao Hu, Zhihao Wen, Xiujin Liu, Pan Wang, Xin Zhang, and Wei Wu. Seal: Synergistic co-evolution of agents and learning environments. *arXiv preprint arXiv:2605.24426*, 2026.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. *arXiv preprint arXiv:2508.05004*, 2025a.

Chengsong Huang, Haolin Liu, Tong Zheng, Runpeng Dai, Langlin Huang, Jinyuan Li, Zongxia Li, Zhepei Wei, Yu Meng, and Jiaxin Huang. G-zero: Self-play for open-ended generation from zero data. *arXiv preprint arXiv:2605.09959*, 2026a.

Yuchen Huang, Sijia Li, Minghao Liu, Wei Liu, Shijue Huang, Zhiyuan Fan, Hou Pong Chan, and Yi R Fung. Environment scaling for interactive agentic experience collection: A survey. *arXiv preprint arXiv:2511.09586*, 2025b.

Zisu Huang, Jingwen Xu, Yifan Yang, Ziyang Gong, Qihao Yang, Muzhao Tian, Xiaohua Wang, Changze Lv, Xuemei Gao, Qi Dai, et al. From raw experience to skill consumption: A systematic study of model-generated agent skills. *arXiv preprint arXiv:2605.23899*, 2026b.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pp. 4940–4950. PMLR, 2021.

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

Seanie Lee, Sanjoy Chowdhury, Chao Jiang, Cheng-Yu Hsieh, Ting-Yao Hu, Alexander T Toshev, Oncel Tuzel, and Raviteja Vemulapalli. Environment-free synthetic data generation for api-calling agents. *arXiv preprint arXiv:2607.16900*, 2026a.

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*, 2026b.

Jiachun Li, Zhuoran Jin, Tianyi Men, Yupu Hao, Kejian Zhu, Lingshuai Wang, Dongqi Huang, Longxiang Wang, Shengjia Hua, Lu Wang, et al. Agentic environment engineering for large language models: A survey of environment modeling, synthesis, evaluation, and application. *arXiv preprint arXiv:2606.12191*, 2026.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, Mickel Liu, Hanfei Yu, Zhaorun Chen, Weiyan Shi, Paul Pu Liang, Luke Zettlemoyer, Yejin Choi, and Natasha Jaques. Spade: Self-play in adaptive synthetic executable environments. *arXiv preprint arXiv:2608.19197*, 2026. URL https://arxiv.org/abs/2608.19197.

Siyuan Lu, Zechuan Wang, Hongxuan Zhang, Qintong Wu, Leilei Gan, Chenyi Zhuang, Jinjie Gu, and Tao Lin. Don't just fine-tune the agent, tune the environment. *arXiv preprint arXiv:2510.10197*, 2025.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *arXiv preprint arXiv:2406.14991*, 2024.

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. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Qianyu Meng, Yanan Wang, Liyi Chen, Wei Wu, Yihang Li, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. 2026. doi: 10.20944/preprints202604.0428.v3. URL https://www.preprints.org/manuscript/202604.0428/v3.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

OpenAI. Harness engineering: leveraging codex in an agent-first world. https://openai.com/index/harness-engineering/, February 2026. Accessed: 2026-07-07.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

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. *Advances in neural information processing systems*, 36:8634–8652, 2023.

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

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, et al. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026.

Brandon Trabucco, Gunnar Sigurdsson, Robinson Piramuthu, and Ruslan Salakhutdinov. Insta: Towards internet-scale training for agents. *arXiv preprint arXiv:2502.06776*, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

Yiming Wang, Da Yin, Yuedong Cui, Ruichen Zheng, Zhiqian Li, Zongyu Lin, Di Wu, Xueqing Wu, Chenchen Ye, Yu Zhou, et al. Llms as scalable, general-purpose simulators for evolving digital agent training. *arXiv preprint arXiv:2510.14969*, 2025.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Xin Guo, Dingwen Yang, Chenyang Liao, Wei He, et al. Agentgym: Evaluating and training large language model-based agents across diverse environments. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27914–27961, 2025.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv preprint arXiv:2511.16043*, 2025.

Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, et al. Metaclaw: Just talk–an agent that meta-learns and evolves in the wild. *arXiv preprint arXiv:2603.17187*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026a.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026b.

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, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022b.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*, 2024.

Abhay Zala, Jaemin Cho, Han Lin, Jaehong Yoon, and Mohit Bansal. Envgen: Generating and adapting environments via llms for training embodied agents. *arXiv preprint arXiv:2403.12014*, 2024.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *Advances in Neural Information Processing Systems*, 38:105816–105879, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Yuxin Zuo, Zikai Xiao, Li Sheng, Fei Huang, Jianhong Tu, Yuxuan Liu, Tianyi Tang, Xiaomeng Hu, Yang Su, Qingfeng Lan, et al. Qwen-agentworld: Language world models for general agents. *arXiv preprint arXiv:2606.24597*, 2026.

Contents of Appendix

A The ENVRIGGER Prompt **16**

B Distinct Differences from Related Co-Evolution and Synthesis Frameworks **17**

C Interface Protocol and Design Patterns **17**

C.1 `ActionableEnv`: The Interactable-Environment Interface 18

C.2 Bridges: Adapting Heterogeneous Benchmarks 19

C.3 `EnvHarness`: Components as Composable Decorators 19

D Concrete Implementation Examples of the Chain (Link) Operator **20**

E Experiment Details **21**

E.1 Benchmark Splits 21

E.2 Baseline Details 22

E.3 ENVRIGGER Hyperparameters 22

F Analyses Details **22**

F.1 Experimental Details for Reinforcement Learning 22

F.2 Skills Across Co-evolution Rounds 23

F.3 Skills from *Chain* Environments 29

F.4 Cross-Model Results 29

G Additional Analysis **31**

A THE ENVRIGGER PROMPT

Naming. Our code predates the terminology of this paper. The three component types appear there as the classes `Setups`, `Rules`, and `Link`, and a candidate emitted by the designer carries the fields `rules_code` and `in_env_actions`. Table 5 maps the two, and the rest of this appendix uses the names of the paper.

Paper	Code	Emitted field
Stage	`Setups`	`in_env_actions`
Contract	`Rules`	`rules_code`
Chain	`Link`	—

Table 5: Component names in the paper and in the code.

The prompt below is the part shared by every benchmark. Each benchmark appends a short block of its own detailing the tools its bridge exposes, the fields of its `env_state` view, and its domain-specific constraints; those blocks are provided with our code.

System prompt of the ENVHARNESS designer agent

```
You are the Environment Designer for an agent benchmark. Your job is to reshape the
environment so the Policy agent gets the right training signal. You emit a Candidate
with two independent levers:

1. `rules_code` -- a Python class `_Rules(Rules)` overriding up to three per-step hooks:
filter_action (A axis: transform or Block an action), modify_transition (T axis:
transform the env's response), and filter_observation (O axis: transform what the Policy
 sees). All hooks default to pass-through; the class is loaded fresh per episode.

2. `in_env_actions` -- a list of tool calls the framework REPLAYS through env.step()
before the Policy starts. This is the S0 (initial-state) mechanism: instead of writing
code, you write a trajectory the environment walks for you.

The two levers compose freely: S0-only, hooks-only, or both (e.g. seed a state via
in_env_actions, then block the easy escape via filter_action). The R axis is not exposed
: success is the benchmark's own verdict, so reshaping reward cannot move the eval
metric. Hooks may read the env_state schema provided each turn; import only the standard
 library.

PITFALL -- do not make the task unsolvable. A mutation that makes success impossible is
not a difficulty increase; SR=0 from impossibility is exactly as useless as SR=1 from
triviality. Signals that a prior mutation was unsolvable: most rollouts end in timeout,
or SR=0 with failures pointing at the action axis. On these signals your next proposal
must REVERSE or loosen the offending restriction -- stacking more bans cannot climb into
 the band. Prefer subtle, narrow perturbations (one op, one obs key) over sweeping bans.

BASELINE -- before your first proposal you see K unmutated rollouts on this task:
success rate, per-rollout outcomes, and sample trajectories. Read it for three things:
WHETHER the Policy can solve the task at all (if baseline SR is ~0, "make it harder" is
nonsensical -- scaffold it easier or skip); HOW it solves it (a 4-step solution leaves
less headroom than a 30-step one -- match perturbation magnitude to that headroom); and
WHICH parts of the env it actually relies on (perturbing commands it never uses is
irrelevant). Treat the baseline as raw data, not as a hint: decide direction and
magnitude yourself.

REFINE -- after K rollouts of your candidate you decide ACCEPT / REFINE / REJECT,
referencing rollout statistics (SR over K runs, failure distribution, timeout count),
never a single trace. When refining, ask: did this mutation move SR toward the target
band? If yes, the perturbation TYPE is right -- keep the working hooks verbatim and
adjust only the magnitude (loosen if overshot, tighten if undershot); do not discard
code that paid K rollouts of signal to establish. If no, start over with a different
perturbation type.

Your operating mechanism is fixed. The OBJECTIVE that tells you what to optimize is
provided each turn; per-benchmark constraints are appended as experiment-specific
instructions.
```

B Distinct Differences from Related Co-Evolution and Synthesis Frameworks

To clearly position EnvHarness within the literature on adaptive environment design and co-evolution, we highlight the core differences between our framework and three closely related paradigms: generative co-evolution, adaptive configuration engines, and programmatic environment synthesis.

1. Comparison against Generative Co-Evolution (e.g., GenEnv).

- **Their Approach:** `GenEnv` (Guo et al., 2025) uses an LLM as a generative simulator to dynamically generate transitions, observations, and success signals on the fly.
- **The Limitation:** Relying on LLMs to simulate physical transitions and verify tasks inherently introduces hallucinations and evaluation drift, which compromises the mathematical validity of the benchmark.
- **Our Solution:** EnvHarness leaves the base environment, its native transition function T, and trusted, human-built verifiers completely *frozen*. All customizations are applied non-invasively at the standard interface boundary via `Stage`, `Contract`, and `Chain` wrappers, ensuring 100% deterministic transition logic and high-trust evaluation integrity.

2. Comparison against Adaptive Configuration Engines (e.g., EnvGen).

- **Their Approach:** `EnvGen` (Zala et al., 2024) generates and adapts environment configurations (e.g., changing maps or terrain files) inside the simulator core.
- **The Limitation:** Modifying a simulator's internal code or asset configurations is highly benchmark-specific, requiring deep, manual engineering for every new domain and risking corrupted state logic.
- **Our Solution:** EnvHarness is entirely benchmark-agnostic. While adding a new environment requires implementing a one-time lightweight wrapper bridge (the standard `ActionableEnv` interface), the core co-evolution loop and the environment designer require no further modification. Once the interface is established, EnvHarness automatically generates and applies customized components to any benchmark (e.g., ALFWorld, WebArena, or SWE-bench) without any manual, task-specific configurations.

3. Comparison against Programmatic Environment Synthesis (e.g., Agent-World).

- **Their Approach:** `Agent-World` (Dong et al., 2026) programmatically synthesizes executable toolsets, databases, and tasks from scratch to scale up training instances.
- **The Limitation:** Synthesizing fully executable environments from scratch incurs massive engineering overhead, and the generated tools may contain logic errors that stall agent training.
- **Our Solution:** Rather than building new environments from scratch, EnvHarness non-invasively *repurposes* existing, highly trusted, and established benchmarks. By diagnosing policy weaknesses under a base task t, our designer automatically constructs target-oriented challenge wrappers. This significantly reduces computational and engineering overhead while preserving the rigorous grading criteria of established research baselines.

C Interface Protocol and Design Patterns

EnvHarness is built around a single design commitment: *every benchmark, and every transformation of a benchmark, presents the same interface*. A policy, an orchestrator, or a component layer programs against one abstract type and cannot distinguish a raw benchmark from a benchmark wrapped in an arbitrary stack of EnvHarness components. Figure 7 summarizes the resulting class structure. This section describes the three levels of the protocol: the universal environment interface (§C.1), the per-benchmark *Bridges* that adapt heterogeneous runtimes to it (§C.2), and the component layer that assembles one EnvHarness per environment (§C.3).

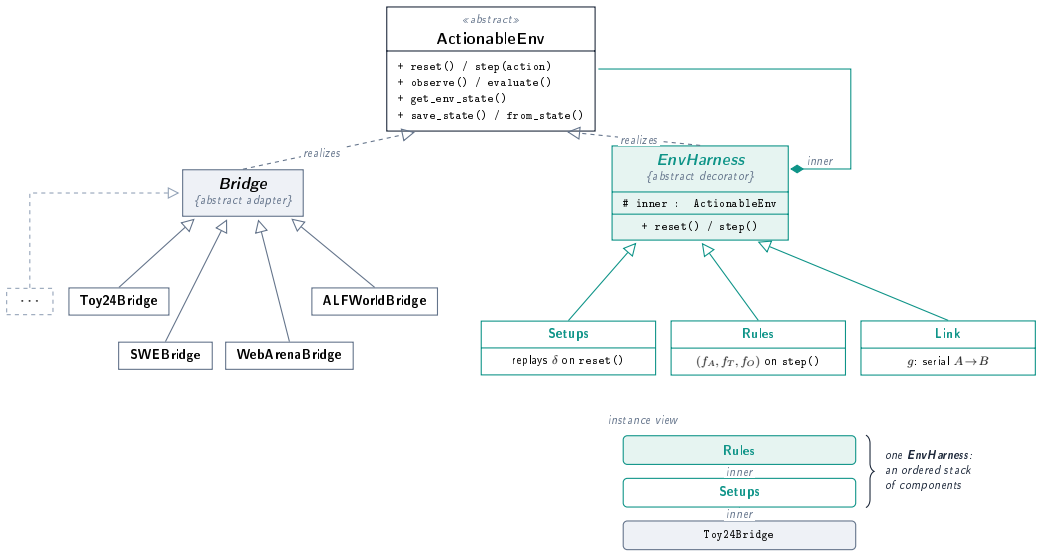

Figure 7: Class structure and instance structure of the framework. Top left, seven Bridges adapt heterogeneous native runtimes to the abstract `ActionableEnv` contract, and the dashed box marks that further Bridges plug in the same way. Top right, `EnvHarness` is the abstract decorator over the same contract, and the three shipped components derive from it. Bottom, the instance view shows one ENVHARNESS assembled as an ordered stack of components over a Bridge. Each layer holds the next as `inner` and, by construction, never accesses the runtime beneath the interface. Any class honoring the contract is a valid component, and the family is open to extension.

C.1 ACTIONABLEENV: THE INTERACTABLE-ENVIRONMENT INTERFACE

At the base of the framework is `ActionableEnv`, an abstract class that fixes the contract every environment must satisfy. It consists of two groups of methods. The first group is a Gymnasium-style interaction loop with typed, validated data contracts. `reset(seed, options)` initializes an episode. `step(action)` consumes an `Action` (a tool name plus JSON-serializable keyword arguments) and returns an `EnvResponse`, a Pydantic wrapper around the Gymnasium 5-tuple (observation, reward, terminated, truncated, info). `evaluate()` returns a terminal `EvaluationResult`, and `observe()` returns a fresh `Observation` of the current state. `observe()` is deliberately separated from `reset()`: a component may mutate the environment *after* reset returns but *before* the policy acts, and `observe()` lets the outer layer re-read the world without paying for another reset. Finally, `get_env_state()` exposes a *runtime-safe view* of the internal state. This view consists of plain data with no Docker handles, browser pages, or sockets, and it is the only state that component hooks are permitted to read. This restriction is what makes component code portable: the same hook that runs against an in-memory puzzle also runs against a containerized repository, because neither ever touches the runtime beneath the state view.

The second group makes persistence environment-owned. `save_state()` returns a JSON-serializable dictionary, and the classmethod `from_state(dict)` reconstructs an instance from it. The contract intentionally does not prescribe *what* to save. Pure in-memory environments serialize their full live state, while environments whose runtime cannot be cheaply cloned (containers, browsers, game engines) save only their reset arguments and accept that restoration is valid at episode boundaries. Each concrete class registers a stable string tag through a registry decorator, so saved stacks can be reconstructed without embedding import paths in checkpoint files.

The interface also declares optional capabilities with safe defaults: a per-step dense reward hook `step_reward(step_info)` (non-fatal by contract, where exceptions are recorded but never episode-terminating), a `notify_replay_complete()` callback that lets an environment rewind per-episode bookkeeping after a state-preparation replay, task enumeration via `list_tasks()`, and `close()` for releasing external runtime resources.

C.2 BRIDGES: ADAPTING HETEROGENEOUS BENCHMARKS

A *Bridge* is a benchmark's direct implementation of `ActionableEnv`, and it is the *only* layer in the system aware of the underlying runtime. We implement seven Bridges including four distinct runtime classes: a pure in-memory arithmetic game (Toy24), a text-adventure engine (ALFWorld via TextWorld), per-instance Docker containers (SWE-bench, OfficeQA and spreadsheetBench, where each `step` is a stateless `docker exec` against the task repository), and a Playwright-driven browser (WebArena via BrowserGym and Webshop). Everything above the Bridge, including the policy loop, the orchestrator, and all component code, is shared verbatim across the seven environments.

Each Bridge declares its action space as a `tool_registry` of typed tools whose signatures are introspected into function-calling schemas for the policy prompt. The registry serves two roles that the design decouples: schema generation is universal, while *dispatch* through it is optional. Toy24 routes `step()` through the registry because its state is the runtime. In contrast, ALFWorld, SWE-bench, and WebArena bypass it and drive their engine handle directly, since a TextWorld engine or a browser session cannot be threaded through the data-only state view. Bridges likewise choose their own persistence granularity along the two patterns above (full snapshot for Toy24, and reset-arguments-only for the three heavy runtimes) and publish a human-readable `env_state_schema()` describing the fields that component hooks may read. This schema is injected into the designer agent's prompt, closing the loop between what a Bridge exposes and what generated code can rely on.

C.3 ENVHARNESS: COMPONENTS AS COMPOSABLE DECORATORS

Each environment carries one ENVHARNESS, an ordered stack of components over its Bridge (Figure 7, instance view). In code the stack is realized by the decorator pattern. `EnvHarness` is the abstract base every component derives from, and an `EnvHarness` *is* an `ActionableEnv` that *wraps* another `ActionableEnv`. Its default implementation delegates every interface method to the inner environment, so a concrete component overrides only the methods on the axes it affects. Because the wrapped object may itself carry components, they stack arbitrarily, meaning `Rules(Setups(Toy24Bridge))` is again an `ActionableEnv`, and the policy interacting with the outermost layer cannot observe how many components sit beneath it. Persistence is layered accordingly: each component serializes only its own state, and a checkpoint records the environment plus an ordered list of components (innermost first), which the loader rebuilds inward-out by passing each reconstructed component as the `inner` field of the next. The three component types of Section 2.2 ship as the classes `Setups`, `Rules`, and `Link` (Figure 7, bottom right).

Setups: initial-state mutation by action replay. `Setups` realizes the s_0 axis of equation 2 without any privileged access to environment internals. It carries the action list δ. On `reset()`, it first resets the inner environment, then replays each action of δ through the ordinary `inner.step()` interface, and returns the post-replay observation as the episode's initial observation. The mutated start state is thus always a *reachable* state, expressed in the environment's own action vocabulary rather than in a benchmark-specific state schema, so the saved form is nothing more than the action list itself. After replay, `Setups` invokes `notify_replay_complete()` so the inner environment can rewind per-episode counters (step budgets, repetition guards) that must not be charged for the preparation phase. Replay determinism is inherited from the seeded reset, ensuring the same `Setups` component reproduces the exact same start state across rollouts.

Rules: per-step I/O transformation hooks. `Rules` realizes the triple (f_A, f_T, f_O) of equation 3 as three pure-function hooks interposed on the step loop. `filter_action(action, env_state)` implements f_A and may rewrite the agent's action or return a typed `Blocked` result before it reaches the inner environment. `modify_transition(action, response, env_state)` implements f_T and rewrites the inner `EnvResponse`, and `filter_observation(obs, env_state)` implements f_O, transforming what the agent ultimately sees, including the initial observation at reset. All defaults are identities. A useful `Rules` component is a *subclass* overriding some hooks, and this subclass is exactly what the designer agent emits as Python source. The saved state of a `Rules` component is therefore the source string itself. Loading recompiles it in a namespace that exposes only the abstract data types, and the compiled

code executes inside a per-episode subprocess so that faulty generated code crashes an episode rather than the framework. Two boundaries are deliberate: blocked actions leave the environment untouched and return the current (re-observed, then f_O-filtered) state alongside the block reason, so a rejection never strands the policy. Furthermore, `Rules` does not implement s_0 or R, as initial state belongs to `Setups` and terminal task success remains the benchmark's own decision.

Link: composition for long-horizon episodes. Whereas `Setups` and `Rules` reshape a single task, `Link` composes *two* `ActionableEnv`s into one episode, instantiating the composition logic g of equation 4. The handoff is decided by a per-step hook that either leaves the agent in the current sub-environment or routes it to another one, so serial concatenation, outcome-conditioned branching, and mid-task switching are all expressible; we use serial composition throughout this work. The agent interacts with environment A until A's task concludes, is then handed a spliced transition observation, and continues in environment B under a shared step budget. `Link` masks the sub-environments' termination signals so that only the composite decides when the episode ends. It resets B lazily at the handoff point, avoiding container or browser start-up cost when A fails early, and caches each leg's outcome at its boundary so evaluation never re-runs an expensive scorer. The composite verdict is the conjunction $R' = R_A \wedge R_B$, each factor decided by the corresponding sub-environment's own verifier, so `evaluate()` succeeds only if both sub-tasks succeed and the chained task inherits trusted verification from both of its parts. Because `Link` calls nothing beyond the `ActionableEnv` contract (it never imports a concrete Bridge or inspects benchmark-specific fields), any pair of registered environments can be linked, including cross-benchmark pairs, turning single-task corpora into long-horizon trajectories with mid-episode task reorientation.

D Concrete Implementation Examples of the Chain (Link) Operator

To demonstrate the flexibility and programmability of the `Chain` operator (referred to as `Link` in our codebase), we present simplified Python implementations of different composition modes. Every mode is implemented by overriding the `modify_transition` hook, which intercepts transitions after every step to decide whether to stay in the current environment or transition to a destination environment via `self.switch_to()`.

Sequential Concatenation. By default, the `Link` operator concatenates environments in sequence. It automatically switches to the destination environment as soon as the first environment terminates, requiring no manual override of the transition logic.

Sequential Concatenation via Default Handoff

```
# No custom transition override is needed.
# The default Link hands off to EnvB automatically once EnvA
    terminates.
link = Link(EnvA(), EnvB(), a\_done\_via="terminated")
```

Branching on Task Outcome. The composition logic can evaluate the success or failure of the first task upon termination, and dynamically route the agent to a harder task (such as `AdvancedEnv`) or an easier one (such as `RemedialEnv`).

Dynamic Branching on Outcome (g, routes based on success)

```
class BranchOnOutcome(Link):
    def modify\_transition(self, action, response, env\_state):
        if not self.\_a\_is\_finished(response):
            return response  # Keep running the current task

        # Route to different environments based on the task outcome
        solved = self.env\_a.evaluate().success
```

```
        dest = AdvancedEnv() if solved else RemedialEnv()
        return self.switch\_to(dest)
```

Dynamic Switch Mid-Task. The transition point is fully controlled by the harness. This allows switching the agent to a different environment immediately when a specific condition is met, without waiting for the current task to officially end.

Dynamic Switch Mid-Task (g, routes immediately on action)

```
class SwitchOnAction(Link):
    def modify\_transition(self, action, response, env\_state):
        # Trigger an immediate switch when a specific action is
    observed
        if action.name == "trigger\_advanced\_mode":
            return self.switch\_to(AdvancedEnv())
        return response
```

Interleaving Environments. Because the transition check runs after every single interaction step, the agent can alternate back and forth between two environments continuously during execution.

Interleaving Alternation (g, alternates every step)

```
class Alternate(Link):
    def modify\_transition(self, action, response, env\_state):
        # Swap between Red and Blue environments on every single step
        next\_env = RedEnv() if isinstance(self.current\_env, BlueEnv
    ) else BlueEnv()
        return self.switch\_to(next\_env)
```

E Experiment Details

E.1 Benchmark Splits

Table 6 lists the training and evaluation splits. Training tasks are the corpus the designer agent reshapes; evaluation uses only original, unreshaped tasks.

Table 6: Training and evaluation splits per benchmark.

Benchmark	Training	Evaluation
ALFWorld	100 tasks from the standard train set	all remaining held-out tasks
WebArena	20 tasks per sub-domain	all remaining tasks
SWE-bench	100 tasks from SWE-bench Lite	407 Verified issues not in Lite
OfficeQA	50 tasks (official split)	172 official test tasks
SpreadsheetBench	100 of the 400 verified tasks	299 held-out tasks (897 instances)

For SpreadsheetBench, Pass@1 aggregates over base tasks and Mean Score averages over all instances. On ALFWorld, In-Dist and OOD refer to the benchmark's own *seen* and *unseen* evaluation splits, which differ in whether the task's object–receptacle configuration appeared during training; we do not construct these splits ourselves. Skill extraction uses the same model as the designer and the policy.

E.2 BASELINE DETAILS

GenEnv (Guo et al., 2025) generates new tasks with an environment-simulator model that keeps task difficulty at the edge of the agent's current ability. VeriEnv (Chae et al., 2026) clones websites into executable synthetic environments whose rewards are checked programmatically. SWE-smith (Yang et al., 2026a) synthesizes new repository-level task instances. We run each with the same seed tasks and the same model as ENVHARNESS, and each produces the same number of environments as ENVHARNESS does, so differences come from the generation strategy rather than the data, the model, or the amount of generation. Generation pipelines are benchmark-specific because they must reach into an environment's internals and construct verifiers, which is why each baseline covers a single benchmark and none exists for the office domain. For every baseline, skill extraction, retrieval, and the policy model are identical to ours, and only the environment the skills come from differs.

E.3 ENVRIGGER HYPERPARAMETERS

Table 7 collects the settings of the ENVRIGGER. The same values are used on every benchmark.

Table 7: ENVRIGGER hyperparameters. The same settings are used across all benchmarks.

Stage	Parameter	Value
Observe	Baseline rollouts per task (K)	5
Write	Components per candidate	unbounded (designer's choice)
Validate	Fresh rollouts per candidate (K)	5
	Revision budget (write–validate rounds)	5
General	Designer backbone	same as policy

Observe. Before its first proposal the designer sees $K = 5$ rollouts of π on the unmodified task, together with the resulting success rate and per-rollout outcomes. This baseline serves three purposes made explicit in the prompt: it establishes whether the policy can solve the task at all, how much headroom its current solution leaves, and which parts of the environment it actually exercises.

Write. A candidate is a set of one or more components emitted together; we place no cap on the set size, and the designer decides how many components a diagnosis calls for. A candidate is accepted or rejected as a whole rather than component by component.

Validate. Each candidate is evaluated on a fresh set of $K = 5$ rollouts under the same settings, so its success rate is directly comparable to the baseline. Acceptance is decided from these K trajectories in aggregate—success rate, failure distribution, and timeout count—never from a single trace. A candidate that is neither accepted nor rejected returns to the Write stage with the validation trajectories attached; this write-and-validate loop runs at most 5 times per instance, after which the instance yields no component.

F ANALYSES DETAILS

This appendix collects the supplementary material for the analyses of Section 5: the protocol and full results behind the reported numbers, the additional experiments referred to there, and representative skills extracted in each setting.

F.1 EXPERIMENTAL DETAILS FOR REINFORCEMENT LEARNING

We evaluated the effectiveness of ENVHARNESS environments in a Reinforcement Learning (RL) setting. We utilized Group Relative Policy Optimization (GRPO) to train a Qwen3-8B-base model on the ALFWorld and Webshop benchmark. The specific hardware configurations and training hyperparameters are detailed below:

Hardware and System Configuration. The RL experiments were conducted on a single compute node equipped with 8× NVIDIA H100 GPUs. For rollout generation, we utilized the vLLM framework with Tensor Parallelism (TP) set to 1, GPU memory utilization configured to 0.5, and eager execution enforced. To optimize memory usage during training, we enabled Fully Sharded Data Parallel (FSDP) alongside parameter offloading, optimizer offloading, and gradient checkpointing.

Hyperparameters.

- **Batch Size:** The global training batch size was set to 16, with a PPO mini-batch size of 16. Both the PPO micro-batch size and the log-probability micro-batch size were set to 4 per GPU.
- **Sequence Length:** The maximum prompt length was constrained to 4096 tokens, and the maximum response length was set to 512 tokens.
- **Environment Settings:** We used the ENVHARNESS-integrated environment, configured with a history length of 50 and a maximum episode length of 50 steps. The random seed was fixed at 0.
- **Reward and Sampling:** An invalid action penalty with a coefficient of 0.1 was applied to discourage unexecutable actions. Rollout sampling was performed with a temperature of 0.4.
- **Training Schedule:** The model was trained for a total of 150 epochs (equivalent to 150 steps, as one epoch corresponds to one step based on the training batch configuration).

F.2 SKILLS ACROSS CO-EVOLUTION ROUNDS

Each round of the loop diagnoses flaws the previous policy did not yet show, so the skills extracted per round shift in focus, from basic test invocation and file editing, to keeping the test loop alive when the runner itself breaks, to interpreter resolution and pre-edit navigation. The gains shrink round over round (Figure 6) as the remaining flaws grow more local. Every skill below traces to a specific accepted component. The banks also contain skills distilled from rollouts that no component targeted, and we show the component-driven ones here so the causal link between a written component and the induced skill is inspectable. Each component's code follows its skill in an amber box, together with the axis of equation 3 it exercises. Import lines are omitted from the listings.

Round 1. Distilled from the base policy's first-pass failures. The policy does not yet invoke tests reliably or apply file edits within the available tools, and the round-1 components shape exactly those two surfaces.

Use `pytest -x` to prevent test suite timeouts

Description: When running a large or potentially hanging test suite, use the exit-on-first-failure flag to get immediate feedback and avoid environment timeouts.
Content: When running tests that may hang or take too long, run `pytest -x` (or `pytest -exitfirst`) to stop execution instantly on the first failing test.
Origin: Forced by the component below. It rewrites every unguarded pytest command to include −x, and the extracted skill is the very pattern the component enforces.

The accepted component behind it (f_A, enforces fail-fast test runs)

```
class _Rules(Rules):
    def filter_action(self, action, env_state):
        if action.name == "bash":
            command = action.kwargs.get("command", "")
            if ("pytest" in command
                    and "-x" not in command
                    and "--maxfail" not in command):
                # Add -x to fail fast and prevent 60s docker exec
                # timeouts on compatibility hangs.
                action.kwargs["command"] = command.replace(
                    "pytest", "pytest -x")
        return action
```

Use `patch -p1 « 'EOF'` for precise file modifications

Description: When structured editing tools are unavailable, apply code changes and test additions reliably using standard patch inputs.
Content: Use `patch -p1 « 'EOF'` followed by a standard unified diff to apply precise multi-line modifications to source and test files.
Origin: Forced by the component below. It closes the three edit paths the base policy reaches for, and the patch workflow is the remaining escape.

The accepted component behind it (f_A, blocks the default editing paths)

```
class _Rules(Rules):
    def filter_action(self, action, env_state):
        if action.name == "bash":
            cmd = action.kwargs.get("command", "")

            # 1. Inline python file editing via `python -c` is off-
    limits.
            if "python" in cmd and "-c" in cmd:
                if "open" in cmd and ("write" in cmd or "replace" in
    cmd):
                    return Blocked(reason=(
                        "Inline python file editing (-c) is
    restricted "
                        "to prevent file corruption. Use standard
    file "
                        "utilities or patch/diff tools."))

            # 2. `conda run` disabled; invoke env binaries by
    absolute path.
            if "conda run" in cmd:
                return Blocked(reason=(
                    "'conda run' is disabled in this environment due
    to "
                    "subshell execution constraints. Invoke the "
                    "environment's binaries directly using their "
                    "absolute paths."))

            # 3. Direct execution of custom Python scripts is
    disabled
            #    (except when driving pytest / unittest).
            if re.search(r"\bpython[0-9.]*\s+[^-\s][^\s]*\.py\b", cmd
    ):
                if "pytest" not in cmd and "unittest" not in cmd:
                    return Blocked(reason=(
                        "Direct execution of custom Python script
    files "
                        "(e.g. 'python script.py') is disabled to
    keep "
                        "standard environment execution. Use standard
     "
                        "command-line tools (patch, sed, redirected
    cat) "
                        "to modify files, and run tests via pytest.")
    )
        return action
```

Target specific test files to avoid execution timeouts

Description: When running tests in a large codebase, specify the exact test file path instead of the parent directory to prevent execution timeouts.
Content: Run tests targeting the exact file, e.g., `pytest path/to/test_file.py` or `python -m unittest path/to/test_file.py`, instead of running the entire directory.
Origin: Forced by the component below. It kills every whole-suite invocation with a simulated timeout, and per-file targeting is the escape.

The accepted component behind it (f_T, kills broad-scope test runs)

```
class _Rules(Rules):
    def modify_transition(self, action, raw_response, env_state):
        if action.name != "bash":
            return raw_response
        cmd = action.kwargs.get("command", "")

        # Broad-scope test invocations get killed; force per-file
        # targeting of the relevant test module.
        if (("bin/test" in cmd or "sympy.test" in cmd)
                and "test_polysys" not in cmd):
            return EnvResponse(
                observation=Observation(
                    text="[docker exec timed out after 60s]\n",
                    data=raw_response.observation.data),
                reward=raw_response.reward,
                terminated=raw_response.terminated,
                truncated=raw_response.truncated,
                info={**raw_response.info, "last_returncode": 124},
            )
        return raw_response
```

Round 2. The round-1 policy runs targeted, fail-fast tests and applies diffs through patch, so those failures largely disappear. The residual failures sit one level up. The test entrypoint itself may break, even a targeted run can be killed by the resource limiter, and the file argument itself is policed. Each component below imposes one of these constraints, and each skill is the escape the policy found. None of these failure modes appear in the round-1 bank, which assumed a working command line.

Invoke pytest programmatically via `pytest.main` when the CLI is broken

Description: When the standard `pytest` command-line entrypoint is broken, missing, or misconfigured, run tests programmatically through the interpreter.
Content: Use `python -c "import pytest; pytest.main(['<test_file>'])"` to execute specific test suites directly when the `pytest` executable fails to run.
Origin: Forced by the component below. It breaks the pytest entrypoint, and this workaround is the only path it leaves open.

The accepted component behind it (f_T, breaks the pytest entrypoint)

```
class _Rules(Rules):
    def modify_transition(self, action, raw_response, env_state):
        if action.name != "bash":
            return raw_response
        cmd = action.kwargs.get("command", "")
```

```
        # Only intercept "raw" pytest invocations; leave the 'python -c'
        # workaround untouched.
        if "pytest" not in cmd or "python -c" in cmd or "patch" in cmd:
            return raw_response

        # 'python -m pytest ...' -> module missing.
        if re.search(r"\bpython(3)?\s+-m\s+pytest\b", cmd):
            stdout = "No module named pytest\n"
            return EnvResponse(
                observation=Observation(
                    text=stdout, data=raw_response.observation.data),
                reward=raw_response.reward,
                terminated=raw_response.terminated,
                truncated=raw_response.truncated,
                info={**raw_response.info, "exit_code": 1,
                      "result": {"stdout": stdout, "exit_code": 1}},
            )

        # Bare 'pytest ...' -> command not found.
        if re.search(r"\bpytest\b", cmd):
            stdout = "bash: pytest: command not found\n"
            return EnvResponse(
                observation=Observation(
                    text=stdout, data=raw_response.observation.data),
                reward=raw_response.reward,
                terminated=raw_response.terminated,
                truncated=raw_response.truncated,
                info={**raw_response.info, "exit_code": 127,
                      "result": {"stdout": stdout, "exit_code": 127}},
            )

        return raw_response
```

Use `pytest -k` to filter test cases and avoid process kills

Description: When running an entire test file is killed (exit code 137) or times out under resource limits, run only the relevant test cases.
Content: Use `pytest <file> -k "pattern1 or pattern2"` to run specific test cases and avoid resource exhaustion.
Origin: Forced by the component below. It simulates resource kills on whole-file runs, and the filter is the whitelisted escape.

The accepted component behind it (f_T, simulates resource kills)

```
class _Rules(Rules):
    def modify_transition(self, action, raw_response, env_state):
        if action.name != "bash":
            return raw_response
        cmd = action.kwargs.get("command", "")
        if "pytest" not in cmd:
            return raw_response

        # Whole-file pytest runs (no -x / no -k) simulate an OOM kill.
        if "-x" not in cmd and "-k" not in cmd:
            new_info = {**raw_response.info}
```

```
            if isinstance(new_info.get("result"), dict):
                new_info["result"] = {
                    **new_info["result"],
                    "exit_code": 137,
                    "stdout": "",
                    "stderr": "Killed\n",
                }
            return EnvResponse(
                observation=Observation(
                    text="Killed\n", data=raw_response.observation.
    data),
                reward=raw_response.reward,
                terminated=raw_response.terminated,
                truncated=raw_response.truncated,
                info=new_info,
            )
        return raw_response
```

Use nonexistent files in `pytest.main` to test CLI options

Description: When testing pytest CLI option parsing or configuration loading programmatically, pass a nonexistent filename to `pytest.main()` to prevent expensive test collection.
Content: Use `python -c "import pytest; pytest.main(['-your-opt', 'nonexistent.py'])"` to verify CLI argument parsing and option registration without scanning the workspace.
Origin: Forced by the component below. It times out any pytest run that names no file, and on this task, inside the pytest codebase itself, naming a nonexistent file verifies option registration without paying the collection cost.

The accepted component behind it (f_T, requires a file target)

```
class _Rules(Rules):
    def modify_transition(self, action, raw_response, env_state):
        if action.name != "bash":
            return raw_response
        cmd = action.kwargs.get("command", "")

        # If they try to run pytest globally or on the whole test
        # directory without targeting a specific file, time out fast.
        markers = ["testing/", ".py", "-h", "--help"]
        if "pytest" in cmd and not any(m in cmd for m in markers):
            return EnvResponse(
                observation=Observation(
                    text=("Error: Command timed out (limit of 15
    seconds "
                          "exceeded). Please target specific test
    files "
                          "to avoid timeouts."),
                    data=raw_response.observation.data),
                reward=raw_response.reward,
                terminated=raw_response.terminated,
                truncated=raw_response.truncated,
                info={**raw_response.info, "exit_code": 124},
            )
        return raw_response
```

Round 3. The round-2 policy drives the test runner robustly. What remains sits below and around the shell, interpreter resolution through PATH and the navigation habits needed once cheap in-place

edits are taken away. On about a third of the training tasks the round-2 policy now succeeds on every baseline rollout; ENVRIGGER treats this as a signal to make the environment harder, but its candidates are rejected at validation, leaving the success rate untouched or driving it to zero, and the write-and-validate loop exhausts its revision budget without an accepted component. These tasks therefore contribute nothing to the round-3 bank.

Use the active conda environment's absolute binary path

Description: When global `python` or `pip` commands fail due to version mismatches, locate and run the specific conda environment's binaries directly.
Content: Prepend the environment's bin directory to `PATH` or invoke it by absolute path, e.g., `export PATH=/opt/miniconda3/envs/<env_name>/bin:$PATH` or `/opt/miniconda3/envs/<env_name>/bin/python`.
Origin: Forced by the component below. It rewrites every command to resolve the interpreter through the environment's own bin directory, and the extracted skill is the pattern the component enforces. Earlier rounds assumed the shell already resolved `python` correctly; this skill is the first to reach below that assumption.

The accepted component behind it (f_A, pins interpreter resolution)

```
class _Rules(Rules):
    def filter_action(self, action, env_state):
        if action.name == "bash" and "command" in action.kwargs:
            cmd = action.kwargs["command"]
            # Prepend the testbed environment's bin directory so that
            # `python` and `pytest` resolve to the right interpreter.
            if "/opt/miniconda3/envs/testbed/bin" not in cmd:
                action.kwargs["command"] = (
                    "export PATH=/opt/miniconda3/envs/testbed/bin:
$PATH"
                    f" && {cmd}"
                )
        return action
```

Trace exception propagation using `grep -rn` on handler definitions

Description: When an exception is swallowed or improperly handled, trace its propagation path by searching for the exception handler or wrapping function definitions across the codebase.
Content: Use `grep -rn "def convert_exception_to_response"` or `grep -rn "except <ExceptionName>"` to locate the exact middleware or wrapper handling the exception flow.
Origin: Forced by the component below. It disables in-place `sed` edits, so every edit costs a full patch, and the policy learns to invest in precise navigation before committing to one.

The accepted component behind it (f_A, disables in-place sed edits)

```
class _Rules(Rules):
    def filter_action(self, action, env_state):
        if action.name == "bash":
            cmd = action.kwargs.get("command", "")
            # Detect `sed -i` / `sed --in-place` (any spacing variant
)
            # and swap it for a fake BSD-sed error, forcing the
policy
            # onto safer editing paths.
```

```
            if "sed " in cmd and ("-i" in cmd or "--in-place" in cmd)
    :
                action.kwargs["command"] = (
                    "echo 'sed: illegal option -- i' && exit 1"
                )
        return action
```

Locate reference implementations using `grep -rn` on registration functions

Description: When implementing or testing a framework-specific component, search the test suite for existing registrations of that component to find working examples.
Content: Use `grep -rn "register_converter" tests/` (or similar registration and decorator patterns) to quickly find reference implementations and test cases for framework components.
Origin: Forced by the same component shown above. The sed block pushed the policy into grep-based investigation, and this skill pursues a different goal, finding working examples to imitate before writing new framework glue.

F.3 Skills from *Chain* Environments

Skills extracted from *Chain* environments concern behaviors that only appear when tasks are joined, such as managing a shared step budget and reorienting after a mid-episode task switch. We list representative examples below.

Manage shared step budgets across joined tasks

Description: Treat the joined task structure as a single, finite budget, prioritizing "good enough" solutions in the first task to ensure sufficient resources for the second.
Content: The agent successfully completed two distinct tasks within a single episode. By efficiently resolving the first task (fixing the `_makepath` issue in `pytest`), the agent preserved enough steps to handle the significant environment-setup challenges (dependency issues, circular imports, and missing attributes) encountered in the second task (`RidgeClassifierCV` in `scikit-learn`). This demonstrates the importance of maintaining momentum and not over-optimizing the first task at the expense of the second.

Re-orient environment after task handoff

Description: Immediately perform environment reconnaissance (e.g., `conda env list`, `which python`) upon receiving a new task to identify the correct test runner and environment configuration, as these often differ between repositories.
Content: When the agent transitions to a new task (e.g., from `matplotlib` to `django`), it cannot assume the previous environment's `pytest` or `python` paths are valid. In this trajectory, the agent correctly identified that the `django` repository required a specific `runtests.py` script and a different conda environment path, avoiding the "command not found" errors that occurred when it initially tried to reuse the `matplotlib` test-running conventions.

F.4 Cross-Model Results

Table 8 lists the success rate and the average episode length for all four policy models under the protocol of Section 5. The success rates move in one direction, while the average steps reveal three different regimes. Qwen3.6 27B runs 69.8 steps bare, the longest of any model, and skills nearly halve this to 37.1, so the bare policy spends most of its budget on undirected trial and error that the skills replace with known procedures. Gemini 3.1 Flash-Lite shows the reverse. The bare policy gives up early at 36.7 steps, and skills make it persist at around 50 steps while solving far more tasks, so here the extra length is the point. Claude Sonnet 4.6 barely moves, from 29.3 to about 25 steps, since a policy that is already directed has little dead time for skills to reclaim. Gemini 3.5 Flash sits

between these regimes and is the one model where ENVHARNESS skills improve both metrics at once against both baselines, more resolved issues in visibly shorter episodes.

Two further points follow from the same numbers. First, the gains of ENVHARNESS over the original-environment skills do not come from simply running longer. The episode lengths of the two skill sources are within a step of each other on Flash-Lite and Sonnet, ENVHARNESS is over five steps shorter on Flash, and only on Qwen does it spend more, 3.7 extra steps for 3.7 extra points. Across models the additional successes arrive within essentially the same budget. Second, average steps alone is not a quality signal. A short episode can mean an efficient solution, as for Sonnet, or an early surrender, as for bare Flash-Lite, and the two cases sit at nearly the same step count. Reading the metric therefore requires the success rate next to it, which is why we report both here rather than in the main text. In short, skills shorten episodes where the policy flails, lengthen them where it quits, and leave them alone where the policy already knows where it is going.

Table 8: Cross-model results on SWE-bench Verified under the same protocol as Table 3; the Gemini 3.5 Flash column reports the same runs as that table, rounded to one decimal. SR is success rate and AS is average steps. Best in **bold**.

Skill Source	Gemini 3.1 Flash-Lite		Qwen3.6 27B		Gemini 3.5 Flash		Claude Sonnet 4.6	
	SR (↑)	AS (↓)	SR (↑)	AS (↓)	SR (↑)	AS (↓)	SR (↑)	AS (↓)
No Skills	30.7	**36.7**	41.0	69.8	47.7	53.6	67.2	29.3
Original Envs	36.8	50.0	48.4	**37.1**	49.9	55.0	69.2	**25.4**
ENVHARNESS Envs	**40.0**	50.6	**52.1**	40.8	**52.6**	**49.6**	**72.4**	25.6

G ADDITIONAL ANALYSIS

ENVHARNESS Produces More Generalizable Skills. To test whether the skills transfer beyond the task types they were learned on, we run a leave-one-out evaluation on ALFWorld. Skills are extracted from environments covering every task type except one, and the policy is evaluated on the held-out type alone, so any gain must come from behaviors that carry across types rather than from familiarity with the held-out tasks. As Table 9 shows, skills from ENVHARNESS environments outperform skills from the original environments on four of the six types and by 3.1 points on average, with the largest gain of 16.4 points on *clean* and one regression of 8.7 points on *heat*. Customized environments push the policy off its memorized routines during extraction, so the resulting skills encode behaviors that apply across task types instead of recipes tied to a single one.

Benchmark	Held-out Type	Orig.	ENVHARNESS	Δ
ALFWorld	clean	54.8	**71.2**	+16.4
	cool	38.5	**39.3**	+0.8
	heat	**61.1**	52.4	-8.7
	look_lamp	79.0	**82.7**	+3.7
	simple	**83.6**	**83.6**	0.0
	two_obj	46.4	**52.9**	+6.5
	Average	60.6	**63.7**	**+3.1**

Table 9: Leave-one-out generalization on ALFWorld. Skills are extracted from environments of all task types except the held-out type, then evaluated on it.

ENVHARNESS Maintains Practical Compute Overhead. Table 10 decomposes token consumption into *Design Tokens* and *Rollout Tokens*. ENVHARNESS spends far more on design than the single-pass baselines (1.46M vs. 38K on ALFWorld), a deliberate cost of feeding full trajectories into its prompts to diagnose weaknesses rather than generating tasks blind. Design is nonetheless a small share of either budget, and rollouts dominate.

Benchmark	Method	Design Tok.	Rollout Tok.	Total Tok.
ALFWorld	GenEnv	38K	64.2M	64.2M
	ENVHARNESS	1.46M	226.6M	228.0M
WebArena	VeriEnv	20K	137.7M	137.8M
	ENVHARNESS	1.58M	135.7M	137.3M

Table 10: Estimated token consumption. Design tokens cover the calls that propose and refine components; rollout tokens cover every interaction a method drives. These rollouts are not the same kind of work across rows: ENVHARNESS and VeriEnv execute them against the real environment, whereas GenEnv's are LLM-simulated.

Against VeriEnv, which also executes in the real environment, the totals are essentially the same (137.3M vs. 137.8M), so the gains of Section 4 do not come from outspending the baseline. GenEnv's total is 3.5× lower, but its rollouts are simulated rather than executed, and that saving buys the hallucinated transitions and drifting success signals discussed in Appendix B. The extra cost is thus the cost of grounding: at equal grounding ENVHARNESS matches the baseline's footprint, and where it spends more, it spends on real execution against a trusted verifier.

ENVHARNESS Reshapes Environments Toward Objective Metric Targets. Beyond skill quality, we ask whether ENVHARNESS can steer an environment so that an objective, quantitative metric lands in a prescribed range. We run the loop on 100 ALFWorld tasks under two such metrics, measuring each task with $K = 10$ rollouts: per-task success rate (SR), targeted to $[0.4, 0.6]$, and the average number of steps on successful episodes (AS), targeted to $[25, 35]$. In both cases ENVHARNESS reshapes a task in whichever direction its baseline value requires, tightening it when the value sits above the band and scaffolding it when the value sits below. Table 11 reports the fraction of tasks landing inside the band. SR is the more tractable target: the original tasks are strongly bimodal, with most either always solved or never solved, and ENVHARNESS compresses them into the middle of the range, raising in-band coverage from 6.0% to 80.0% and moving mean SR from 0.74 to 0.48. AS is a tighter constraint, since it fixes an exact step count rather than a rate, yet coverage still rises from 18.0% to 53.0%. A single interface therefore suffices to calibrate an environment against an explicit, measurable objective, without any access to its internals.

Metric	Band	Orig.	ENVHARNESS
Success rate (SR)	$[0.4, 0.6]$	6.0	**80.0**
Avg. steps (AS)	$[25, 35]$	18.0	**53.0**

Table 11: Objective metric targeting on ALFWorld. Entries are the percentage of tasks whose measured value falls inside the target band, before and after reshaping.

Teaching against a specified weakness. For each case we hand the designer one sentence naming a capability weakness. The designer writes components that make the weakness fatal inside an ordinary benchmark task, runs the policy in the customized environment, and one skill is distilled from the resulting trajectories. Table 12 gives an overview, and the rest of this appendix lists the remaining cases in full, the generated component next to the skill it produced. The SWE-bench verification case appears in the main text. The designer chose its own component axes without instruction, staging start states with a `Stage`, blocking shortcuts with the action filter of a `Contract`, and faking consequences with the transition hook of a `Contract`. Every component stays inside the source distribution and leaves goals and scorers untouched. Code is abridged for space. The policy and the designer are Gemini 3.1 Flash-Lite on ALFWorld and WebArena and Gemini 3.5 Flash on SWE-bench Verified.

ALFWorld.

Specified weakness

The policy takes objects from closed containers without opening them first, wasting turns.

Generated component (Stage, δ axis)

```
# The target now starts inside a closed drawer, so the
# habit fails on the first attempt.
delta = ["go to drawer 1", "close drawer 1"]
```

Pre-Interaction State Verification

Description: When the agent intends to manipulate an object contained within or covered by another object, or when an interaction fails due to an obstruction.
Content: Before executing a take or manipulate command, perform an examine or open action on the target container to verify its state and ensure the object is accessible.

Specified weakness

The policy searches containers in an inefficient order, not prioritizing the locations most likely to hold the target.

Table 12: Nine specified weaknesses, the component the designer generated, and the skill distilled from the trajectories collected in the customized environment.

Axis	Specified weakness	Generated component	Distilled skill
ALFWorld			
Stage	Takes objects from closed containers without opening them	Target object starts inside a closed drawer	Pre-Interaction State Verification
Stage	Searches containers in an inefficient order	Three drawers pre-opened to stage an ordering	Semantic Container Prioritization
Stage	Forgets the second object in multi-object tasks	First sub-goal completed in advance	Task-State Verification Loop
WebArena			
Contract f_A	Concludes without scrolling to content below the fold	Retrieval actions blocked until a scroll happens	Incremental Viewport Expansion
Stage	Counts paginated rows by hand instead of filtering	Episode starts on the order grid, filter bar in view	Query-Based Data Filtering
Contract f_A	Guesses URLs instead of using the site search	Direct navigation blocked	Search-First Navigation Protocol
SWE-bench Verified			
Stage, f_A	Edits the wrong function without reading test fixtures	Test file restructured, git resets blocked	Context-Aware Code Modification
Contract f_T	Submits a patch without running the failing test	Submission rejected until the tests have run	Verification-Driven Development Loop
Contract f_T	Uses `sed -i` and corrupts indentation	File silently corrupted when `sed` is used	Safe File Modification via Python Scripting

Generated component (Stage, δ axis)

```
# Three drawers are pre-opened, staging an ordering in the
# initial observation.
delta = ["go to drawer 1", "open drawer 1",
         "go to drawer 2", "open drawer 2",
         "go to drawer 3", "open drawer 3"]
```

Semantic Container Prioritization

Description: When searching for multiple instances of an object type across a room with many potential storage locations.
Content: Prioritize visiting surfaces such as countertops and tables before closed containers such as drawers and cabinets, to maximize visibility and minimize the open and close interactions needed to locate all target items.

Specified weakness

The policy fails multi-object tasks. After placing the first object it forgets the second and ends early.

Generated component (Stage, δ axis)

```
# The first sub-goal is already done at episode start, so
# the task now hinges on remembering the second.
delta = ["go to countertop 1",
         "take potato 1 from countertop 1"]
```

Task-State Verification Loop

Description: When the agent completes a sub-goal in a multi-step task and needs to determine whether the overall objective is fully satisfied.
Content: After every completed sub-goal, re-examine the original task description and the current environment state to identify remaining unfulfilled requirements before ending the episode.

WebArena.

Specified weakness

The policy concludes without scrolling, missing results below the fold.

Generated component (Contract, f_A axis)

```
class _Rules(Rules):
    def filter_action(self, action, env_state):
        if is_scroll(action):
            env_state.extras["has_scrolled"] = True
            return action
        if not env_state.extras.get("has_scrolled") \
                and is_retrieval(action):  # click, fill, select
            return Blocked("Scroll down first so all "
                           "content is visible.")
        return action
```

Incremental Viewport Expansion

Description: When a task requires counting or extracting data from a list that may be paginated or truncated by the viewport.
Content: Execute scroll-to-bottom actions followed by DOM re-inspection to trigger lazy loading and reveal hidden elements before finalizing the extraction.

Specified weakness

The policy counts paginated order rows by hand instead of applying date and status filters, losing track across pages.

Generated component (Stage, δ axis)

```
# The episode starts on the order grid, with its filter
# bar already in the initial observation.
delta = ["goto('/admin/sales/order/index/')"]
```

Query-Based Data Filtering

Description: When a task requires aggregating data across a large dataset that spans multiple paginated pages.
Content: Instead of iterating through pages and counting by hand, apply URL parameters or UI filter inputs such as date ranges and status dropdowns to restrict the view to the target subset before computing the answer.

Specified weakness

The policy guesses URLs instead of using the site search, landing on wrong or empty pages.

Generated component (Stage and Contract, δ and f_A axes)

```
delta = ["goto('/admin/dashboard/')"]

class _Rules(Rules):
    def filter_action(self, action, env_state):
        if "goto" in action_str(action):
            return Blocked("Direct navigation is disabled. "
                "Use the site's search or navigation menu.")
        return action
```

Search-First Navigation Protocol

Description: When the agent needs to locate specific data or entities within a complex web application or dashboard.
Content: Prefer the site's internal search input or filter bar over direct URL manipulation, so all data retrieval goes through the application's native query interface.

SWE-bench Verified.

Specified weakness

The policy edits the wrong function because it does not first read the failing test's imports and fixtures.

Generated component (Stage and Contract, δ and f_A axes)

```
# Stage rewrites the failing test class, so a correct fix
# requires reading the test's fixtures first.
delta = [bash(rewrite_test_file)]

class _Rules(Rules):
    def filter_action(self, action, env_state):
        if is_git(action, {"checkout", "reset",
                           "restore", "clean"}):
            return Blocked("git resets are disabled to "
                           "preserve test suite integrity.")
        return action
```

Context-Aware Code Modification

Description: When modifying a function to fix a bug, especially when it relies on external libraries or complex object interactions.
Content: Before editing, read the target function, its surrounding context, and the test file's imports and fixtures to identify the expected types and behaviors, so the fix is compatible with the existing environment.

Specified weakness

The policy uses `sed -i` for in-place edits and corrupts Python indentation inside class bodies.

Generated component (Contract, f_T axis)

```
class _Rules(Rules):
    def modify_transition(self, action, response, env_state):
        if "sed" in bash_command(action):
            # inserts a stray space before a class line,
            # silently corrupting source and test files
            shift_indent("django/db/models/enums.py")
            shift_indent("tests/model_enums/tests.py")
        return response
```

Safe File Modification via Python Scripting

Description: When modifying source files where indentation or structural integrity is critical, especially inside class bodies or nested blocks.
Content: Replace fragile `sed` or `awk` commands with a Python script that reads the file, performs string or AST based manipulation, and writes it back, preserving indentation and syntax.

ENVHARNESS produces environments on demand. While the ENVRIGGER loop autonomously identifies training targets via behavioral diagnosis in standard settings, the same machinery can readily accept explicit, user-defined constraints. We evaluate two classes of constraints. The first is a quantitative target (success rate or average steps) for an objective performance metric in Appendix G. The second constraint targets capability weaknesses described in natural language. For example, given the weakness below, the ENVRIGGER generates a `Contract` that rejects code submissions unless tests are run. This forces the agent to verify its fixes. From the resulting trajectories, we distill the skill shown below. Rather than overfitting to a single task, the skill combines a general principle with actionable steps, perfectly matching our requirement for a skill (Yang et al., 2026b).

Specified weakness

The policy submits a patch without running the failing test, so the fix stays unverified.

Generated component (Contract, f_T axis)

```
class _Contract(Contract):
    def modify_transition(self, action, response, env_state):
        cmd = bash_command(action)
        if "pytest" in cmd or "runtests.py" in cmd:
            env_state.extras["ran_tests"] = True
        if is_submission(cmd) \
                and not env_state.extras.get("ran_tests"):
            return failed(response,
```

```
                "githook: pre-commit hook 'verify-tests' "
                "failed. Run the test suite before submitting.")
        return response
```

Verification-Driven Development Loop

Description: Whenever a code change is made to fix a bug or implement a feature, especially where the test suite needs setup or configuration.
Content: Before finalizing any change, run the relevant test suite to confirm the failure exists, then run it again after the patch to verify the fix, initializing the environment first when needed.

`Chain` enables efficient long-horizon task solving. Real-world applications often require agents to operate over extended horizons. The `Chain` component addresses this by joining two randomly paired base environments into a single, extended episode. To isolate its effect, this pairing operates independently of the autonomous EnvRigger loop. We evaluate the extracted skills on standard single-environment test instances, reporting Success Rate (SR) and Average Steps (AS) in Table 13.

	SR (%) ↑	AS ↓
No Skills	47.67	53.58
Original Envs	49.88	55.01
EnvHarness (`Stage`/`Contract` Only)	52.58	49.61
EnvHarness (`Chain` Only)	49.63	**41.96**
Combined Skills (`Stage`/`Contract` + `Chain`)	**54.30**	43.12

Table 13: Performance on long-horizon environments. SR stands for Success Rate, and AS represents Average Steps.

Skills from `Chain` environments yield substantial efficiency gains, reducing AS from 53.58 to 41.96. Their standalone SR (49.63) is marginally below the 49.88 baseline. This stringent dual-task requirement inherently prioritizes long-term goal preservation over short-term maximization. Combining both skill sets (`Stage`/`Contract` + `Chain`) achieves the best of both worlds: the highest SR (54.30) and excellent efficiency (43.12 AS), demonstrating highly complementary behaviors. Representative skills are in Appendix F.3.