HARNESS TRAINING: A POST-TRAINING VIEW OF AGENT HARNESS EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents depend on an agent harness that comprises persistent prompts, skills, tools, middleware, and workflows governing their behavior. Harness evolution updates this state after deployment without changing model weights. Existing work is commonly categorized by the component being edited or the search algorithm, rather than by whether an update is retained to reproduce target behavior or improve execution outcomes. We frame harness evolution as *Harness Training*, an analogue of parameter post-training at the level of learning objectives. This view distinguishes *Harness-SFT*, which learns from target trajectories, from *Harness-RL*, which learns from reward feedback. Experiments show that Harness-SFT makes agent executions more closely follow target trajectories, while Harness-RL changes retained behavior in the direction predicted by the reward objective. When combined, Harness-SFT increases the availability of task-solving edits, and Harness-RL uses reward feedback to identify and retain them, improving reward search. Based on this finding, we develop *Signal-Targeted Harness Training* (STHT), which applies Harness-SFT to tasks failed by the initial harness before initializing Harness-RL from the supervised state. Across diverse experimental settings, STHT reaches validation thresholds with fewer rollouts and outperforms competing methods on unseen tasks. These gains transfer across harnesses, reward optimizers, and teachers. Code is available anonymously at `https://anonymous.4open.science/r/harness-train-45C6/`.

1 INTRODUCTION

Large language models (LLMs) now support agentic systems that can plan, use tools, maintain memory, interact with environments, and coordinate multi-step actions (Yao et al., 2023; Wang et al., 2024; Yang et al., 2024b). Deployed behavior also depends on the surrounding *agent harness*: the persistent model-external state that organizes instructions and context, exposes tools, mediates observations, retains reusable knowledge, and governs execution (Pan et al., 2026a; Li et al., 2026). As model–harness systems are applied to open-ended, knowledge-intensive tasks, their ability to specialize and adapt after deployment becomes an important design requirement.

Agent systems already contain many of the ingredients needed to participate in this adaptation. They can generate task experience, critique unsuccessful executions, write and test reusable programs, and evaluate candidate system components. Viewed through the signal used to retain an update, existing approaches exhibit two recurring learning patterns. One line consolidates reusable knowledge from supplied or self-generated trajectories into persistent plans, skills, tools, or memories (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024; Yuan et al., 2024; Ouyang et al., 2026). These methods retain updates that better reproduce or preserve the demonstrated behavior. Another line uses execution outcomes as the selection signal. It proposes candidate prompts, workflows, skills, or complete harnesses and retains candidates that improve task reward or verifier outcomes (Yang et al., 2024a; Agrawal et al., 2026; Zhang et al., 2025; Wang et al., 2026; Lee et al., 2026; Park et al., 2026). Both turn task experience into persistent changes that shape future behavior, shifting improvement from a purely external engineering process toward a partially endogenous process of *harness evolution*.

However, this shared learning structure is not reflected in how the literature is organized. Existing work is typically grouped by the artifact it edits or the optimizer it uses, which conflates what is updated with what is learned. Edits to the same reusable skill can aim to reproduce target behavior

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



Figure 1: Parameter post-training and Harness Training share the distinction between target imitation and outcome optimization, but update different persistent state. The former updates model parameters, whereas the latter updates model-external harness components.

or improve reward, while either objective can update any harness component. Artifact-based categories therefore group together methods with different learning objectives and separate methods that optimize the same objective.

To develop such a view, we draw on parameter post-training. Parameter post-training adapts a pretrained model after pretraining by updating its parameters using target demonstrations or feedback on sampled behavior (Ouyang et al., 2022; Rafailov et al., 2023; Shao et al., 2024). It commonly combines two classes of techniques. Supervised fine-tuning (SFT) learns from target responses, while reinforcement learning (RL) and preference optimization learn from rewards, preferences, or verifiers applied to sampled behavior (Ouyang et al., 2022; Rafailov et al., 2023; Shao et al., 2024). These two classes parallel the trajectory-based consolidation and outcome-based selection patterns described above. Both update persistent model state that shapes future behavior.

Harness evolution exposes a similar learning structure. Its persistent state is model-external and structured, and its updates are usually discrete rather than gradient based. This correspondence motivates the central question of this paper:

Can harness evolution be understood through the lens of parameter post-training?

Under this view, we treat harness evolution as training persistent model-external state while keeping the task model frozen. We call this process *Harness Training*. The trajectory-based learning pattern described above corresponds to *Harness-SFT*. Harness-SFT uses target trajectories to guide edits that reproduce demonstrated behavior. The outcome-based learning pattern corresponds to *Harness-RL*. Harness-RL uses rewards, preferences, or verifier feedback to determine which edits are retained.

These terms describe the learning objective of a training stage, not the edited artifact or the optimizer. A method may alternate between Harness-SFT and Harness-RL, and either objective may update a prompt, Skill, memory, workflow, or full harness. Both regimes typically optimize discrete artifacts through LLM-guided search rather than gradients. Figure 1 illustrates their correspondence to parameter post-training. Whether regularities from parameter post-training extend to Harness Training remains an empirical question. We study this correspondence through four questions.

RQ1. Does Harness-SFT exhibit SFT-like behavior? We test whether trajectory-supervised harness updates make agent executions more closely reproduce target behavior. We also examine how

this behavior depends on task–trajectory pairing, trajectory quality and scale, and whether targets come from the agent itself or an external teacher.

RQ2. Does Harness-RL exhibit RL-like behavior? We test whether reward-driven harness updates make the agent learn behavior that obtains higher reward. We also examine whether changing the reward objective changes retained behavior in the predicted direction and compare discrete update rules under matched budgets.

RQ3. When do Harness-SFT and Harness-RL interact effectively? We study when composing Harness-SFT with Harness-RL improves reward search. We match starting reward to isolate the effect of supervised initialization and test whether Harness-SFT increases the availability of task-solving edits that Harness-RL can identify and retain through reward feedback.

RQ4. Can insights from Harness-SFT, Harness-RL, and their composition guide a more effective training framework? We combine the preceding findings into *Signal-Targeted Harness Training* (STHT), which applies Harness-SFT to tasks failed by the initial harness before starting Harness-RL from the supervised state.

This paper makes three contributions:

- We present Harness Training as an objective-level post-training view of agent harness evolution. This view treats persistent model-external state as trainable and distinguishes Harness-SFT from Harness-RL by target imitation versus outcome feedback.
- We experimentally show that Harness-SFT and Harness-RL exhibit the expected SFT-like and RL-like behavior. When combined, Harness-SFT increases the availability of task-solving Harness edits, and Harness-RL uses reward feedback to identify and retain them, improving reward-search efficiency.
- We develop *Signal-Targeted Harness Training* (STHT) from this insight and show that allocating a matched supervision-budget cap to tasks failed by the initial harness improves efficiency.

2 Related Work

2.1 Experience-Driven Harness Evolution

Agents can turn execution experience into persistent reflections, memories, and reusable procedures (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024; Yuan et al., 2024; Ouyang et al., 2026). Reflexion and ReasoningBank also learn from feedback on failed executions (Shinn et al., 2023; Ouyang et al., 2026). The experience used to propose an edit does not determine the objective used to retain it: a target trajectory can guide imitation, while an execution outcome can guide selection.

Prompt and workflow optimizers search candidate external state using execution feedback (Yang et al., 2024a; Agrawal et al., 2026; Zhang et al., 2025). Skill-based methods build and revise reusable procedures from practice or rollout feedback (Zheng et al., 2025; Wang et al., 2026; Yang et al., 2026; Zhang et al., 2026). SkillDAG evolves the graph used to select skills (Bai et al., 2026). Meta-Harness, Retrospective Harness Optimization, Agentic Harness Engineering, AutoSaddler, and Ecdysis optimize broader harness state (Lee et al., 2026; Pan et al., 2026b; Lin et al., 2026; Park et al., 2026; Yue et al., 2026). Natural-Language Agent Harnesses studies a harness representation, while AutoHarness synthesizes harnesses for games (Pan et al., 2026a; Lou et al., 2026). HarnessOpt-Bench and HarnessDev evaluate harness optimization (Ursekar et al., 2026; Wu et al., 2026).

These works develop specific editable representations, update procedures, and evaluation settings. We compare the effects of trajectory supervision and outcome feedback on persistent harness updates under a fixed task model. This objective-level view defines Harness-SFT and Harness-RL by what each update optimizes, even when one method uses both objectives across stages. We also test how the two regimes interact on the same harness.

2.2 From Continuous Parameter Training to Discrete Harness Training

Parameter post-training motivates our empirical questions. SFT learns from target demonstrations, while RL and preference optimization use feedback on sampled behavior (Ouyang et al., 2022; Rafailov et al., 2023; Shao et al., 2024). Studies of demonstration quality and the distinct effects of SFT and RL motivate our tests of trajectory supervision, reward sensitivity, and their composition (Zhou et al., 2023; Chu et al., 2025; Lin et al., 2025).

These findings do not carry over automatically. Parameter post-training typically uses established gradient-based methods to optimize continuous weights. The harness optimizers studied here search discrete, structured artifacts through stochastic LLM edit proposals and finite-rollout selection. A single edit can change many downstream decisions, and its effect can depend on the initial harness, modifier, editable scope, and budget. Different update rules can retain different edits from the same candidate group, while rollout noise can redirect later search. We therefore use parameter-training results to formulate hypotheses, test them experimentally, and identify harness-specific regularities.

3 Harness Evolution as Training

Harness evolution changes persistent model-external state using experience collected from agent execution (Pan et al., 2026a; Li et al., 2026; Lee et al., 2026). Let $H = (P, S, T, W, L)$ collect prompts and context policies, reusable skills, tools and interfaces, runtime middleware, and the agent loop (Pan et al., 2026a; Li et al., 2026). For task x and environment E, model parameters θ and harness H jointly induce a distribution over execution trajectories,

$$\tau \sim \pi_{\theta,H,E}(\cdot \mid x). \tag{1}$$

Parameter post-training fixes H_0 and updates θ using demonstrations or feedback on sampled behavior (Ouyang et al., 2022; Rafailov et al., 2023; Shao et al., 2024), whereas Harness Training fixes θ_0 and updates H. Their shared behavioral interface can be written as

$$\pi_z(\tau \mid x) = \begin{cases} \pi_{\theta,H_0,E}(\tau \mid x), & z = \theta, \\ \pi_{\theta_0,H,E}(\tau \mid x), & z = H. \end{cases} \tag{2}$$

Both optimize persistent state through the behavior it induces, although their feasible behavior classes and optimization procedures differ.

Harness-SFT retains edits that reproduce target behavior. Harness-SFT commonly writes a Skill, tool, or memory entry distilled from a target trajectory (Zhao et al., 2024; Wang et al., 2024; Yuan et al., 2024; Ouyang et al., 2026). The frozen task model then executes with the updated harness, and supervision favors edits whose induced actions more closely follow the demonstration. Let $\mathcal{D}_{\mathrm{sup}} = \{(x_i, \tau_i^\star)\}$ contain tasks paired with target trajectories. We express this behavioral imitation objective as

$$\mathcal{L}_{\mathrm{sup}}(H) = \mathbb{E}_{(x,\tau^\star)\sim\mathcal{D}_{\mathrm{sup}}} \mathbb{E}_{\hat{\tau}\sim\pi_{\theta_0,H,E}(\cdot|x)} \left[\ell_{\mathrm{beh}}(C(\hat{\tau}), C(\tau^\star))\right], \tag{3}$$

where C maps raw executions to comparable behavioral events and ℓ_{beh} measures deviation from the target behavior. The regime is defined by target imitation rather than by a particular metric. This corresponds to parameter SFT at the level of supplied target behavior, not at the level of token cross-entropy or gradient optimization.

The target source defines two supervision conditions. *Teacher Harness-SFT* uses trajectories supplied by an external teacher. *Self Harness-SFT* uses the agent's own previously successful executions as pseudo-demonstrations. Both retain edits for reproducing a selected target, while differing in supervision quality and coverage.

Harness-RL retains edits according to outcome feedback. Harness-RL evaluates sampled behavior using task reward, preferences, or verifier outcomes (Yang et al., 2024a; Agrawal et al., 2026; Zhang et al., 2025; Wang et al., 2026; Lee et al., 2026; Park et al., 2026). Its objective-level correspondence to parameter RL is

$$J_{\mathrm{rew}}(H) = \mathbb{E}_{x\sim\mathcal{D}_{\mathrm{rew}}} \mathbb{E}_{\tau\sim\pi_{\theta_0,H,E}(\cdot|x)}[R(x,\tau)]. \tag{4}$$

The reward may measure task success, utility, cost, preference, or verifier satisfaction. The regime is defined by optimizing sampled outcomes rather than reproducing a supplied target trajectory. This corresponds to parameter RL at the level of the outcome objective, not at the level of gradients or policy likelihoods.

Editable scope and learning signal are independent. Either regime may update a Skill or a broader harness (Wang et al., 2026; Yang et al., 2026; Lee et al., 2026; Park et al., 2026). The editable scope controls which states the optimizer can reach, while the pre-specified learning objective controls how proposed states are evaluated and updated.

Composing Harness-SFT and Harness-RL. Because both regimes update the same persistent Harness, they can be composed within one training schedule. Let $q_k \in \{\mathrm{SFT}, \mathrm{RL}\}$ denote the regime used at stage k:

$$H_{k+1} = \begin{cases} \mathrm{Optimize}_{\mathrm{SFT}}(\mathcal{L}_{\mathrm{sup}}; H_k), & q_k = \mathrm{SFT}, \\ \mathrm{Optimize}_{\mathrm{RL}}(J_{\mathrm{rew}}; H_k), & q_k = \mathrm{RL}. \end{cases} \tag{5}$$

This formulation covers supervised initialization followed by reward search as well as more general alternations between the two regimes. The correspondence to parameter post-training is limited to learning objectives and persistent-state updates; it does not imply shared loss functions, gradient dynamics, or policy classes. Appendix A provides the formal boundary conditions and maps representative methods to these definitions.

4 Empirical Analysis

Experimental setting. Unless otherwise stated, we freeze Qwen3.6-35B-A3B as the task model throughout training and evaluation (Qwen Team, 2026b). The STHT transfer settings additionally evaluate the frozen Qwen3.6-9B task model. Pi is the primary initial Harness, while Codex tests generalization across Harness implementations (Pi Maintainers, 2026; OpenAI, 2026a). The Harness under training is separate from the coding-agent modifier that proposes its edits. We instantiate the modifier with Pi or Codex, use Pi unless otherwise stated, and use GPT-5.5 as its underlying model (OpenAI, 2026b). GPT-5.5 also provides the primary verified teacher trajectories, while Qwen3.5-122B-A10B tests transfer across teachers (Qwen Team, 2026a). The four benchmark families cover spreadsheet editing with SpreadsheetBench, instrumented document QA with DocVQA, embodied interaction with ALFWorld, and executable code generation using the Code Implementation subset of EvoAgentBench. We select this EvoAgentBench subset because its executable tests provide reproducible task-level rewards under a fixed evaluation protocol. Appendix B details the benchmark adaptations, data splits, and subset selection.

Unless otherwise stated, a Meta-Harness-style coding agent proposes edits after inspecting the current Harness, previous patches, and training executions (Lee et al., 2026). The frozen task model evaluates each candidate, and the retained Harness initializes the next round. Harness-SFT retains edits that reduce canonical tool-call normalized edit distance to target trajectories, whereas Harness-RL retains edits according to task reward. For composed training, we divide the training split equally into the Harness-SFT set $\mathcal{D}_1$ and Harness-RL set $\mathcal{D}_2$. Training tasks generate and select candidate edits, validation tasks support checkpoint and threshold selection, and test tasks are used only for final evaluation. We measure successful-edit availability on training tasks and Reward AUC on the shared validation split. Results report means over three runs with shared seeds. Appendix B gives the complete protocols and metric definitions, and Appendices C–E report the full RQ1–RQ3 results.

RQ1. Does Harness-SFT exhibit SFT-like behavior? Repeated Harness-SFT edits lower validation tool-call NED and improve held-out success as training loss falls (Appendix Figure 3). The resulting loss curve resembles supervised learning over behavior, although the optimizer selects discrete Harness edits. Correct task–trajectory pairing produces stronger imitation and task performance than shuffled pairing, while shuffled trajectories still transfer generic tool-use procedures. Increasing the number of correctly paired trajectories further reduces validation NED, and different teacher sources produce different supervision quality (Appendix Figure 4). Together, these experiments show that Harness-SFT responds to pairing, data scale, and teacher quality in the expected supervised-learning directions.

RQ2. Does Harness-RL exhibit RL-like behavior? Changing the reward objective changes tool use, trajectory length, and verification in the targeted directions (Appendix Table 11). We separately compare Parent Acceptance, Group-Relative selection, DPO-inspired Pairwise editing, Population Search, and Bandit Allocation under the same reward and budget (Appendix Table 12) (Shao et al., 2024; Rafailov et al., 2023). Together, these experiments show that Harness-RL responds to reward design in the directions expected of reward-driven learning: the reward determines which behavior is preferred, while the update rule determines how the discrete Harness space is searched.

RQ3. How do Harness-SFT and Harness-RL interact? Harness-SFT→Harness-RL improves validation reward throughout subsequent reward search under both matched Skill scope and practical Full-Harness search (Appendix Figure 6). The two-stage pipeline begins from a stronger Harness, so we also compare it with a reward-only checkpoint matched on initial validation reward and task-level reward profile. The supervised initialization retains an AUC advantage after this matching (Appendix Figure 7). A better starting score therefore explains only part of the gain.

Each Harness-RL round evaluates several edits on one training task. If every candidate fails the task verifier, the persistent Harness cannot change in that round. We therefore measure the fraction of rounds containing at least one task-solving edit and the rollouts consumed before the first retained update. Starting from the reward-matched pair, Teacher SFT produces more candidate groups with a verifier-passing edit and reaches the first persistent update earlier across candidate-group sizes (Appendix Figure 8).

Insight. Harness-SFT makes task-solving edits more available, and Harness-RL uses reward feedback to find and retain them during subsequent search.

5 Signal-Targeted Harness Training

RQ4. Can the preceding insights guide a more effective training framework? Harness-SFT is most useful when it supplies target behavior for tasks failed by the initial harness H_0. We instantiate this principle as *Signal-Targeted Harness Training* (STHT).

5.1 Overview

STHT composes failure-targeted Harness-SFT with reward-driven Harness-RL. Figure 2 shows the complete procedure. The initial harness H_0 first executes the SFT task set $\mathcal{D}_1$ with the task model frozen. STHT collects the failed tasks, obtains teacher trajectories for them, and applies repeated Harness-SFT updates to the same persistent harness. The resulting state H_{sup} initializes Harness-RL on the RL task set $\mathcal{D}_2$. Harness-RL then proposes candidate harnesses, evaluates their induced outcomes with reward, and either retains a candidate or keeps the current state.

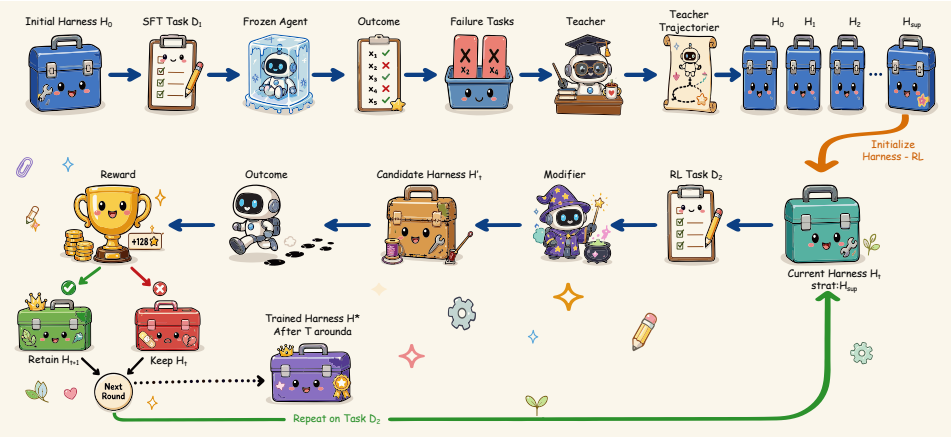

Figure 2: Overview of STHT. The initial harness identifies failed tasks in $\mathcal{D}_1$, repeated Harness-SFT updates produce H_{sup}, and Harness-RL continues from that state on $\mathcal{D}_2$.

5.2 FAILURE-TARGETED HARNESS-SFT

Failure identification. For every task $x \in \mathcal{D}_1$, the frozen task model executes once with the same initial Harness H_0:

$$\tau_x^{(0)} \sim \pi_{\theta_0, H_0, E}(\cdot \mid x), \qquad r_x^{(0)} = R(x, \tau_x^{(0)}). \tag{6}$$

For binary task success, STHT selects the failed-task set

$$\mathcal{D}_{\text{fail}}(H_0) = \left\{ x \in \mathcal{D}_1 : r_x^{(0)} = 0 \right\}. \tag{7}$$

The initial executions provide the targeting signal directly from H_0. A teacher supplies a verified target trajectory $\tau_x^\star$ for each selected task in $\mathcal{D}_{\text{fail}}(H_0)$. Under a supervision-budget cap K, STHT uses

$$K_H = \min\{K, |\mathcal{D}_{\text{fail}}(H_0)|\} \tag{8}$$

failed tasks. Setting $K \geq |\mathcal{D}_{\text{fail}}(H_0)|$ recovers the all-failure setting shown in Figure 2.

Repeated supervised updates. Let the selected target-task set be

$$\mathcal{D}_{\text{target}} = \{x_1, \ldots, x_{K_H}\} \subseteq \mathcal{D}_{\text{fail}}(H_0). \tag{9}$$

STHT applies the same Skill-scoped Harness-SFT procedure as RQ1, with $\Omega_{\text{SFT}} = \{S\}$, to the same persistent Harness in sequence:

$$H_i = \text{HarnessSFT}_{\Omega_{\text{SFT}}}\left(H_{i-1}, (x_i, \tau_{x_i}^\star)\right), \qquad H_{\text{sup}} = H_{K_H}. \tag{10}$$

Each update starts from the state produced by the previous task. The retained edit minimizes the training behavioral imitation loss in Equation 3.

5.3 HARNESS-RL FROM THE SUPERVISED STATE

Harness-RL initializes its current state with H_{sup} and optimizes over $\mathcal{D}_2$. At round t, the fixed modifier G proposes a candidate Harness, and the frozen task model executes it:

$$H_t' \sim G_{\Omega_{\text{RL}}}(\cdot \mid H_t, x_t, \mathcal{E}_t), \qquad \tau_t' \sim \pi_{\theta_0, H_t', E}(\cdot \mid x_t), \qquad r_t' = R(x_t, \tau_t'), \quad x_t \in \mathcal{D}_2. \tag{11}$$

Here, $\mathcal{E}_t$ is the accumulated execution evidence produced by the RL optimizer on $\mathcal{D}_2$. The original Harness-RL update rule then either retains the candidate or keeps the current Harness:

$$H_{t+1} = U_{\text{RL}}(H_t, H_t', r_t'), \qquad H_0^{\text{RL}} = H_{\text{sup}}, \qquad H^\star = H_T. \tag{12}$$

Here, U_{RL} denotes the configured reward-based update rule. It returns H_t' when the candidate satisfies that rule and returns H_t otherwise. Figure 2 depicts one candidate for clarity; optimizers that evaluate candidate groups retain their original selection rules.

6 EVALUATION OF STHT

We evaluate STHT in eight dataset–Harness–actor settings. Five primary settings use the frozen Qwen3.6-35B-A3B actor and GPT-5.5 teacher. They comprise SpreadsheetBench with Pi and Codex, DocVQA and ALFWorld with Pi, and the EvoAgentBench Code Implementation subset with Pi. We use only this EvoAgentBench subset because its executable tests provide objective and reproducible task-level rewards under a fixed evaluation protocol, making it suitable for controlled comparisons of reward-driven Harness updates. Three additional transfer settings use Qwen3.6-9B as the actor and Qwen3.5-122B-A10B as the trajectory teacher on SpreadsheetBench with Pi and Codex and on DocVQA with Pi. Appendix B details the benchmark adaptations, data splits, evaluation protocols, and the rationale for selecting the EvoAgentBench subset. The two SpreadsheetBench Harnesses share the same data split but are trained separately, while the transfer settings retain the same training and evaluation protocol and change only the actor and teacher models. Every reported result is the mean over three independent runs that differ only in their random seeds, and all conditions compared within a row use the same three seeds.

6.1 END-TO-END PERFORMANCE

To distinguish the benefit of failure-targeted supervision from the general benefit of initializing RL with SFT, we compare STHT with three SFT–RL variants. Self SFT→RL learns from successful trajectories generated by the frozen actor itself, whereas Uniform SFT→RL uses teacher trajectories from uniformly sampled training tasks under the same supervision budget as STHT. SFT→RL instead continues RL from the Harness produced by the Teacher SFT baseline.

Table 1: Test success across eight settings, grouped by actor and teacher. Initial Harness denotes performance before training.

Dataset / Harness	Initial Harness	RL only	Teacher SFT	Self SFT →RL	Uniform SFT →RL	SFT→RL	STHT
Actor: Qwen3.6-35B-A3B; teacher: GPT-5.5							
SpreadsheetBench / Pi	71.78%	75.36%	72.86%	75.71%	76.43%	77.14%	**78.21%**
DocVQA / Pi	77.81%	81.28%	80.75%	82.35%	83.69%	84.76%	**86.63%**
EvoAgentBench Code Implementation / Pi	50.00%	60.47%	53.49%	55.81%	61.63%	62.79%	**65.12%**
ALFWorld / Pi	78.83%	86.13%	81.02%	82.85%	84.67%	85.40%	**87.23%**
SpreadsheetBench / Codex	76.43%	77.14%	76.79%	78.57%	80.00%	81.79%	**84.64%**
Five-setting macro	70.97%	76.08%	72.98%	75.06%	77.28%	78.38%	**80.37%**
Actor: Qwen3.6-9B; teacher: Qwen3.5-122B-A10B							
SpreadsheetBench / Pi	63.21%	66.43%	63.93%	66.79%	67.86%	68.93%	**70.71%**
SpreadsheetBench / Codex	67.86%	70.00%	68.57%	71.43%	72.86%	74.64%	**77.14%**
DocVQA / Pi	68.18%	70.32%	72.46%	73.53%	75.40%	77.27%	**79.95%**
Three-setting transfer macro	66.42%	68.92%	68.32%	70.58%	72.04%	73.61%	**75.93%**

STHT achieves the highest test success in all eight settings. Across the five primary settings, it improves macro success from 77.28% under Uniform SFT→RL and 78.38% under All-Task SFT→RL to 80.37%. The gains over Uniform range from 1.78 to 4.64 percentage points. On EvoAgentBench Code Implementation, the 3.49-point gain corresponds to three additional solved tasks. On ALFWorld, STHT solves 114/140 seen and 125/134 unseen tasks, or 239/274 overall.

The same ordering persists when both model roles change. With Qwen3.6-9B and Qwen3.5-122B-A10B, STHT reaches 75.93% macro success, versus 72.04% for Uniform SFT→RL and 73.61% for SFT→RL. Its gains over Uniform range from 2.85 to 4.55 points, indicating that failure targeting transfers across actor and teacher models.

Table 2: Training dynamics and normalized end-to-end cost across the five primary settings. Successful-round rate, validation reward AUC, and normalized cost are macro averages. T_q is the median number of task-model rollouts required to reach the validation threshold. Cost is normalized within each setting, with STHT set to 1.000.

Metric	RL only	Teacher SFT	Self SFT→RL	Uniform SFT→RL	STHT
Successful-round rate ↑	0.412	n/a	0.536	0.618	**0.711**
Validation reward AUC ↑	0.727	n/a	0.755	0.776	**0.804**
Median T_q ↓	192	n/a	144	112	**80**
Normalized end-to-end cost ↓	**0.620**	0.460[†]	0.870	0.920	1.000

[†]Teacher SFT terminates after supervised training and omits the Harness-RL stage.

Relative to Uniform SFT→RL, STHT increases the fraction of Harness-RL rounds containing a task-solving edit from 0.618 to 0.711. This increase is accompanied by a higher validation reward AUC, from 0.776 to 0.804, and a 28.6% reduction in median time to threshold, from 112 to 80 rollouts. STHT therefore converts failure-targeted supervision into more productive reward-search rounds and faster validation improvement, at an 8.7% increase in normalized end-to-end cost.

6.2 DOES TARGETING FAILED TASKS MATTER?

We isolate task selection on ALFWorld/Pi under the primary model configuration. All conditions use the same supervision count, Skill-scoped Harness-SFT procedure, Full-Harness reward optimizer,

rollout budget, task order, and checkpoint-selection rule. Only the selection rule changes. Uniform samples tasks randomly; Success-Targeted selects initial successes; High-NED selects the largest normalized tool-call distances to teacher trajectories; and STHT selects initial failures.

Table 3: Teacher-task selection on ALFWorld/Pi. AUC uses validation data; test success uses all 274 test tasks.

Selector	Rounds with success↑	Val. AUC↑	Test success↑
Uniform	0.625	0.790	84.67%
Success-Targeted	0.497	0.755	82.12%
High-NED	0.646	0.798	85.04%
Failure-Targeted (STHT)	**0.716**	**0.817**	**87.23%**

High-NED improves test success from 84.67% under Uniform selection to 85.04%, whereas STHT reaches 87.23%, improving over Uniform and High-NED by 2.56 and 2.19 percentage points, respectively. Success-Targeted performs worse than Uniform on all three metrics. These results show that the benefit does not come simply from supervising tasks already solved by the initial Harness or tasks with large trajectory differences. Instead, initial failures identify supervision targets that better support subsequent reward search.

6.3 Does STHT Remain Effective across Harnesses, Optimizers, and Teachers?

On SpreadsheetBench with Qwen3.6-35B-A3B, we vary the initial Harness, reward optimizer, or trajectory teacher while holding the modifier and budgets fixed. Parent Acceptance is the default optimizer; the Group-Relative and AutoSaddler rows change only this component. The final row replaces GPT-5.5 teacher trajectories with Qwen3.5-122B-A10B. Each row compares Uniform selection with STHT under matched budgets.

Table 4: STHT transfer across initial Harnesses, reward optimizers, and trajectory teachers on SpreadsheetBench.

Initial Harness	Reward optimizer	Teacher	Uniform AUC	STHT AUC	△AUC (pp)	△Final (pp)
Codex	Parent Acceptance	GPT-5.5	0.769	0.799	+3.0	+4.64
Pi	Parent Acceptance	GPT-5.5	0.714	0.741	+2.7	+1.78
Pi	Group-Relative	GPT-5.5	0.731	0.758	+2.7	+2.50
Pi	AutoSaddler	GPT-5.5	0.724	0.750	+2.6	+2.14
Codex	Parent Acceptance	Qwen3.5-122B-A10B	0.748	0.773	+2.5	+2.86

With Parent Acceptance and GPT-5.5 fixed, STHT improves over Uniform for both the Pi and Codex Harnesses. With the Pi Harness and GPT-5.5 fixed, the gain remains positive under Parent Acceptance, Group-Relative selection, and AutoSaddler. With the Codex Harness and Parent Acceptance fixed, STHT also improves performance with trajectories from either GPT-5.5 or Qwen3.5-122B-A10B. These one-factor comparisons show that the benefit of failure targeting is not tied to a particular initial Harness, reward optimizer, or trajectory teacher.

7 Conclusion

This paper studies agent harness evolution as training persistent model-external state. Harness-SFT retains edits that reproduce target behavior, whereas Harness-RL retains edits that improve rewarded outcomes. Experiments show SFT- and RL-like regularities and find that supervised initialization makes task-solving edits easier to discover during reward search. STHT applies this insight by targeting initial failures for teacher supervision before Harness-RL. This correspondence concerns learning objectives, not gradient dynamics. Our results cover a frozen task model and a finite set of harnesses, tasks, teachers, and update rules. Failure-targeted supervision may help less when rewards already distinguish candidates or the model cannot use teacher trajectories. Joint model–harness training and adaptation to changing environments remain future work.

ETHICS STATEMENT

This study evaluates agent-harness training on benchmark tasks and does not involve human or animal subjects. We use third-party benchmarks under their respective terms and will redistribute only data and artifacts for which we have permission. We are not aware of additional ethical concerns specific to this study.

REPRODUCIBILITY STATEMENT

The main text defines the Harness-SFT, Harness-RL, and STHT procedures and reports their evaluation. The appendix specifies the models, initial harnesses, benchmark adapters, data splits, update rules, budgets, and metrics. At submission, we will release an anonymous repository with the implementation, configurations, evaluation scripts, split manifests, and author-generated data; we will document how to obtain third-party benchmarks from their original sources.

AI USE STATEMENT

Generative AI tools assisted with drafting and editing manuscript text, improving clarity and organization, providing feedback on the framing and design of experimental comparisons, and creating and revising schematic figures. The authors reviewed the AI-assisted text and figures, checked the scientific claims and references, made the final research decisions, and take responsibility for the entire submission.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026.

Tong Bai, Zhenglin Wan, Pengfei Zhou, Xingrui Yu, Wangbo Zhao, Yang You, and Ivor W. Tsang. SkillDAG: Self-evolving typed skill graphs for LLM skill selection at scale. *arXiv preprint arXiv:2606.03056*, 2026.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V. Le, Sergey Levine, and Yi Ma. SFT memorizes, RL generalizes: A comparative study of foundation model post-training. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 10818–10838, 2025. URL https://proceedings.mlr.press/v267/chu25c.html.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, Yafeng Deng, and Teng Li. EvoAgentBench: Benchmarking agent self-evolution via ability transfer. *arXiv preprint arXiv:2607.05202*, 2026.

Yoonho Lee, Roshen S. Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. In *Conference on Language Modeling*, 2026. URL https://colm.eventhosts.cc/Conferences/2026/AcceptedPapers. To appear.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

Junjie Li, Xi Xiao, Yunbei Zhang, Chen Liu, Lin Zhao, Xiaoying Liao, Yingrui Ji, Janet Wang, Jianyang Gu, Yingqiang Ge, Weijie Xu, Xi Fang, Xiang Xu, Tianchen Zhao, Youngeun Kim, Tianyang Wang, Jihun Hamm, Smita Krishnaswamy, Jun Huan, and Chandan K. Reddy. Agent harness engineering: A survey. OpenReview preprint, 2026. URL https://openreview.net/forum?id=eONq7FdiHa.

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

Xiaofeng Lin, Hejian Sang, Zhipeng Wang, and Xuezhou Zhang. Debunk the myth of SFT generalization. *arXiv preprint arXiv:2510.00237*, 2025.

Xinghua Lou, Miguel Lazaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: Improving LLM agents by automatically synthesizing a code harness. In *ICLR 2026 Workshop on AI with Recursive Self-Improvement*, 2026.

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. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2024.

Minesh Mathew, Dimosthenis Karatzas, and C. V. Jawahar. DocVQA: A dataset for VQA on document images. In *IEEE Winter Conference on Applications of Computer Vision*, 2021.

OpenAI. Codex cli. GitHub repository, 2026a. URL https://github.com/openai/codex.

OpenAI. Introducing GPT-5.5. OpenAI product release, 2026b. URL https://openai.com/index/introducing-gpt-5-5/.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, 2022.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, et al. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026a.

Wenbo Pan, Shujie Liu, Chin-Yew Lin, Jingying Zeng, Xianfeng Tang, Xiangyang Zhou, Yan Lu, and Xiaohua Jia. Evolving agents in the dark: Retrospective harness optimization via self-preference. *arXiv preprint arXiv:2606.05922*, 2026b.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Pi Maintainers. Pi coding agent. GitHub repository, 2026. URL `https://github.com/earendil-works/pi`.

Qwen Team. Qwen3.5-122B-A10B model card. Hugging Face model repository, 2026a. URL `https://huggingface.co/Qwen/Qwen3.5-122B-A10B`.

Qwen Team. Qwen3.6-35B-A3B model card. Hugging Face model repository, 2026b. URL `https://huggingface.co/Qwen/Qwen3.6-35B-A3B`.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, 2023.

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

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.

Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S. Kalmath, Veronica Chatrath, and Yuan Xue. HarnessOpt-Bench: Evaluating LLMs at harness optimization. *arXiv preprint arXiv:2608.06301*, 2026.

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. *Transactions on Machine Learning Research*, 2024.

Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. SkillGrad: Optimizing agent skills like gradient descent. *arXiv preprint arXiv:2605.27760*, 2026.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. HarnessDev: Can LLMs create and evolve their own agent harness? *arXiv preprint arXiv:2609.01437*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024a.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024b.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

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

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations*, 2024.

Ruiqing Yue, Yu Cui, Zhuoyu Sun, Sicheng Pan, Xianhong Xue, Tingyu Li, Ting Li, Wenzhuo Zhu, Yi Chen, Yifei Liu, Baohan Huang, Zhe Cui, Haibin Zhang, and Cong Zuo. Ecdysis: Efficient and effective training of runtime harnesses for LLM agents. *arXiv preprint arXiv:2609.11677*, 2026.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. CoEvoSkills: Self-evolving agent skills via co-evolutionary verification. In *Conference on Language Modeling*, 2026. URL https://colm.eventhosts.cc/Conferences/2026/AcceptedPapers. To appear.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI Conference on Artificial Intelligence*, 2024.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, et al. SkillWeaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. LIMA: Less is more for alignment. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/ac662d74829e4407ce1d126477f4a03a-Abstract-Conference.html.

This appendix provides supplementary material that complements the main paper:

- Appendix A formalizes the Harness Training view, including the persistent-state representation, the Harness-SFT and Harness-RL objectives, and the scope of the correspondence to parameter post-training.
- Appendix B details the experimental setting, including the models, initial harnesses, training roles, benchmark adapters, data splits, budgets, and metrics.
- Appendix C reports the trajectory-supervision study as Harness-SFT, including supervision conditions, training dynamics, task–trajectory pairing, trajectory scale, and teacher source.
- Appendix D reports the reward-driven study, including reward-objective interventions, the comparison of reward-search update rules, and the comparison of editable scopes.
- Appendix E reports how the two regimes compose, including supervised initialization for reward search, the reward-matched control, and successful-edit availability.
- Appendix F reports task-order sensitivity and cross-Harness transfer.
- Appendix G gives the complete STHT pseudocode.
- Appendix H traces one completed STHT run from Harness-SFT to full-Harness Harness-RL.
- Appendix I gives the complete versioned prompt pack and the benchmark-local actor, teacher, and modifier prompt construction.

A Formal Details of the Harness Training View

Table 5: Representative methods organized by persistent state, available evidence, update rule, and training objective.

Method	State	Evidence	Update rule	Objective
Trajectory-supervised consolidation				
ExpeL	Rules	Success/failure traces	Consolidate reusable rules	Harness-SFT
Voyager	Skill library	Validated self-traces	Retain reusable programs	Harness-SFT
CRAFT	Tool library	Demonstrations	Synthesize reusable tools	Harness-SFT
ReasoningBank	Reasoning memory	Self-generated traces	Retain reusable strategies	Harness-SFT
Outcome-evaluated search				
OPRO	Prompt	Task score	Select higher-scoring prompts	Harness-RL
PromptAgent	Prompt	Error feedback and score	Select error-guided edits	Harness-RL
GEPA	Program modules	Feedback and task metric	Select feedback-guided edits	Harness-RL
AFlow	Agent graph	Validation reward	Search workflow candidates	Harness-RL
SkillGrad	Skill	Trajectory feedback	Diagnose and validate revisions	Harness-RL
SkillOpt	Skill	Rollout reward	Accept or reject revisions	Harness-RL
Meta-Harness	Full harness	Execution reward	Search harness candidates	Harness-RL
AutoSaddler	Runtime harness	Failure signals and score	Diagnose and select patches	Harness-RL

A.1 Persistent State Induces Behavior

Let x be a task, E the environment, θ the model parameters, and H the harness. We represent the harness as

$$H = (P, S, T, W, L), \tag{13}$$

where P denotes prompts and context construction, S reusable skills, T tools and their interfaces, W runtime middleware, and L the agent loop or workflow. Together, the model and harness induce a distribution over execution trajectories:

$$\tau \sim \pi_{\theta,H,E}(\cdot \mid x). \tag{14}$$

Parameter training fixes a harness H_0 and varies θ, yielding the behavior class

$$\Pi_\Theta(H_0, E) = \{\pi_{\theta,H_0,E} : \theta \in \Theta\}. \tag{15}$$

Harness Training fixes model parameters θ_0 and varies H, yielding

$$\Pi_\mathcal{H}(\theta_0, E) = \{\pi_{\theta_0,H,E} : H \in \mathcal{H}\}. \tag{16}$$

Both procedures therefore optimize persistent state that determines behavior on future tasks. We write their shared interface as

$$\pi_z(\tau \mid x), \qquad z \in \mathcal{Z}, \tag{17}$$

where

$$(z, \mathcal{Z}) = \begin{cases} (\theta, \Theta), & \text{parameter training,} \\ (H, \mathcal{H}), & \text{Harness Training.} \end{cases} \tag{18}$$

The two feasible behavior classes need not be equal, and neither class is assumed to contain the other. The correspondence is that both θ and H provide a trainable interface to the induced trajectory distribution.

A.2 Mathematical Formulation of Harness-SFT

Operational loop. Harness-SFT typically follows an iterative inspect, edit, execute, and retain procedure. At round t, let $\mathcal{B}_t \subseteq \mathcal{D}_{\text{sup}}$ be a minibatch of tasks paired with target trajectories. An LLM modifier reads the current harness and the supplied trajectories, then stochastically proposes discrete edits intended to reproduce their behavior. Each candidate is executed by the frozen task model and compared with the corresponding targets:

$$\begin{aligned} H_{t,j} &\sim G(\cdot \mid H_t, \mathcal{B}_t, \Omega_{\text{SFT}}), \\ \hat{\tau}_{t,j}(x) &\sim \pi_{\theta_0, H_{t,j}, E}(\cdot \mid x), \\ \ell_{t,j} &= \frac{1}{|\mathcal{B}_t|} \sum_{(x,\tau^\star)\in\mathcal{B}_t} \ell_{\text{beh}}(C(\hat{\tau}_{t,j}(x)), C(\tau^\star)), \\ H_{t+1} &= U_{\text{SFT}}\left(H_t, \{(H_{t,j}, \ell_{t,j})\}_{j=1}^m\right). \end{aligned} \tag{19}$$

The update may retain a candidate that better reproduces the supplied trajectories even when its task reward remains zero. The targets directly specify the behavior learned, and trajectory imitation selects the persistent update.

Skill and memory updates. When the editable state is a Skill library or memory, the modifier converts the target trajectories into persistent entries,

$$e^A_{t,j} \sim G_A(\cdot \mid H_t, \mathcal{B}_t), \qquad H_{t,j} = \text{Write}_A(H_t, e^A_{t,j}), \qquad A \in \{\text{Skill}, \text{Memory}\}. \tag{20}$$

For a Skill library, $e^A_{t,j}$ is a reusable procedure or program distilled from the demonstrated action sequence. For memory, it is a stored demonstration or a compact strategy extracted from the trajectory and retrieved on later tasks. Supervision is applied to the rollout induced by the written entry. The entry may compress or generalize the demonstration, but the candidate is retained only when its execution more closely follows the target trajectory under ℓ_{beh}.

Objective correspondence. Let $\mathcal{D}_{\text{sup}} = \{(x_i, \tau_i^\star)\}_{i=1}^N$ contain teacher trajectories. Parameter SFT decomposes each trajectory into teacher-forced histories $h_k^\star$ and target actions $a_k^\star$, then minimizes token- or action-level cross-entropy (Ouyang et al., 2022):

$$\mathcal{L}^{\text{param}}_{\text{sup}}(\theta) = \mathbb{E}_{(x,\tau^\star)\sim\mathcal{D}_{\text{sup}}}\left[-\sum_{k=1}^{|\tau^\star|} \log p_\theta(a_k^\star \mid h_k^\star, H_0, E)\right]. \tag{21}$$

After dividing Equation 21 by the empirical mean trajectory length, let $\mu_{\mathcal{D}}(h)$ be the distribution obtained by sampling uniformly from all teacher-forced history occurrences, and let $q_{\mathcal{D}}(a \mid h)$ be the empirical conditional distribution of target actions at history h. The resulting per-action cross-entropy decomposes as

$$\begin{aligned} \overline{\mathcal{L}}^{\text{param}}_{\text{sup}}(\theta) = \mathbb{E}_{h\sim\mu_{\mathcal{D}}}\left[\mathsf{H}(q_{\mathcal{D}}(\cdot \mid h))\right] \\ + \mathbb{E}_{h\sim\mu_{\mathcal{D}}}\left[D_{\text{KL}}\left(q_{\mathcal{D}}(\cdot \mid h) \| p_\theta(\cdot \mid h, H_0, E)\right)\right]. \end{aligned} \tag{22}$$

The first term is independent of θ. If a history appears with only one target action, then $q_{\mathcal{D}}(\cdot \mid h) = \delta_{a^\star(h)}$, its entropy is zero, and the corresponding KL term reduces to $-\log p_\theta(a^\star(h) \mid h, H_0, E)$.

Equation 22 is only the standard cross-entropy identity for parameter SFT; it is not a KL regularizer imposed on Harness Training.

Although a harness induces a trajectory distribution, that distribution is generally neither tractable nor differentiable with respect to the discrete state H. We therefore instantiate ℓ_{sup} with a rollout-level behavioral objective. Let $C(\tau)$ remove nondeterministic metadata and retain observable decisions, and let

$$d_{\text{NED}}(u, v) = \frac{D_{\text{edit}}(u, v)}{\max\{1, |u|, |v|\}} \tag{23}$$

be normalized edit distance. Supervision-driven Harness Training minimizes

$$\mathcal{L}_{\text{sup}}^{\text{harness}}(H) = \mathbb{E}_{(x,\tau^\star)\sim\mathcal{D}_{\text{sup}}} \mathbb{E}_{\hat{\tau}\sim\pi_{\theta_0,H,E}(\cdot|x)} \left[d_{\text{NED}}\left(C(\hat{\tau}), C(\tau^\star)\right) \right]. \tag{24}$$

For the exact-mismatch loss $d_{0/1}(u, v) = \mathbf{1}\{u \neq v\}$ and one target trajectory, this objective becomes

$$\mathcal{L}_{0/1}^{\text{harness}}(H \mid x) = 1 - \Pr_{\hat{\tau}\sim\pi_{\theta_0,H,E}} \left[C(\hat{\tau}) = C(\tau^\star) \mid x \right]. \tag{25}$$

Thus, minimizing the behavioral loss increases the probability of reproducing the target behavior. Normalized edit distance supplies graded credit when complete trajectory matching is rare. Equations 21 and 24 use different estimators and update mechanisms. Parameter SFT updates model weights through teacher-forced token or action losses, whereas Harness-SFT keeps θ_0 frozen and searches over discrete model-external state using stochastic LLM proposals, free-running executions, and an explicit retention rule. Both are supervision-driven because each training example specifies the behavior to imitate.

At a fixed teacher-forced history h with target action $a^\star$, the exact-match reward $R(a) = \mathbf{1}\{a = a^\star\}$ has expectation $p_\theta(a^\star \mid h)$, whereas SFT maximizes $\log p_\theta(a^\star \mid h)$. The two expressions order positive target probabilities identically, but their gradients differ by the factor $p_\theta(a^\star \mid h)$:

$$\nabla_\theta p_\theta(a^\star \mid h) = p_\theta(a^\star \mid h) \nabla_\theta \log p_\theta(a^\star \mid h). \tag{26}$$

Across examples, this factor also changes their relative weighting. We therefore use an objective-level analogy between target imitation and reward optimization without claiming identical gradients or algorithms.

A.3 Mathematical Formulation of Harness-RL

Operational loop. Harness-RL typically follows an iterative diagnose, edit, evaluate, and retain procedure. At round t, the frozen task model executes with the current harness H_t. Let $\mathcal{E}_t$ contain the parent execution, failed candidate trajectories, tool errors, and verifier diagnostics. An LLM modifier reads this evidence, identifies failure patterns, and stochastically proposes discrete harness edits. The task model executes each candidate, and the configured reward or preference rule determines the persistent update:

$$\begin{aligned} &H_{t,j} \sim G(\cdot \mid H_t, x, \mathcal{E}_t, \Omega_{\text{RL}}), \qquad \tau_{t,j} \sim \pi_{\theta_0,H_{t,j},E}(\cdot \mid x), \quad r_{t,j} = R(x, \tau_{t,j}), \\ &H_{t+1} = U_{\text{RL}}\left(H_t, \{(H_{t,j}, r_{t,j})\}_{j=1}^m\right). \end{aligned} \tag{27}$$

The update rule may retain the parent when no candidate satisfies its criterion. Failed trajectories can still guide later diagnosis and proposals even when no candidate is retained. Thus, failure evidence helps determine where the modifier searches, whereas reward or preference feedback determines which edit becomes persistent. Reading a failed trajectory and proposing a repair does not make the stage Harness-SFT because no target behavior is supplied for imitation.

Objective correspondence. Parameter RL and Harness-RL share an expected-reward objective. With a fixed harness, parameter RL optimizes

$$J_{\text{rew}}^{\text{param}}(\theta) = \mathbb{E}_{x\sim\mathcal{D}_{\text{rew}}} \mathbb{E}_{\tau\sim\pi_{\theta,H_0,E}(\cdot|x)} \left[R(x, \tau) \right]. \tag{28}$$

Many LLM post-training methods add a reference-policy penalty (Ouyang et al., 2022),

$$J_{\text{RL}}^{\text{param}}(\theta) = J_{\text{rew}}^{\text{param}}(\theta) - \beta\, \mathbb{E}_{x\sim\mathcal{D}_{\text{rew}}} \left[D_{\text{KL}}\left(\pi_{\theta,H_0,E}(\cdot \mid x) \| \pi_{\theta_{\text{ref}},H_0,E}(\cdot \mid x)\right) \right], \tag{29}$$

or a token-level estimator of this trajectory-distribution divergence. This penalty constrains deviation from a reference policy; it does not by itself eliminate reward misspecification or reward hacking. With a frozen model, reward-driven Harness Training optimizes

$$J_{\mathrm{rew}}^{\mathrm{harness}}(H) = \mathbb{E}_{x\sim\mathcal{D}_{\mathrm{rew}}}\mathbb{E}_{\tau\sim\pi_{\theta_0,H,E}(\cdot|x)}\left[R(x,\tau)\right]. \tag{30}$$

Both are instances of

$$\max_{z\in\mathcal{Z}} \mathbb{E}_x\mathbb{E}_{\tau\sim\pi_z(\cdot|x)}\left[R(x,\tau)\right]. \tag{31}$$

Equations 28 and 30 have the same unregularized expected-reward form, but their optimization processes differ. Parameter RL changes model weights, typically through gradient-based or likelihood-based updates derived from sampled outcomes. Harness-RL keeps θ_0 frozen and searches over discrete model-external state using stochastic LLM proposals, finite executions, and an explicit retention rule. Its full objective may therefore use discrete gates, scope restrictions, or regression tests rather than the reference-policy penalty in Equation 29. Pairwise preferences can likewise evaluate sampled harness behaviors. Exact DPO, however, requires policy likelihood ratios (Rafailov et al., 2023); methods without those probabilities are more accurately described as DPO-inspired preference search.

A.4 Scope of the Correspondence

The correspondence holds at the levels of persistent training state, induced behavior, and the source of the learning objective: target imitation or outcome feedback. It does not imply identical loss functions, equal feasible behavior classes, equal optimization dynamics, or access to the same gradients and probabilities.

The classification follows the learning objective specified for each training stage. A stage that optimizes reproduction of supplied target behavior is Harness-SFT, whereas a stage that optimizes sampled outcomes is Harness-RL. A procedure with no editable persistent external state is not Harness Training. A procedure that updates both model weights and the harness is joint model–harness training and lies outside our frozen-model setting.

Because both objectives update persistent state, supervised and reward-driven stages can be composed. Let $q_k \in \{\mathrm{SFT}, \mathrm{RL}\}$ specify the objective used at stage k:

$$z_{k+1} = \begin{cases} \mathrm{Optimize}_{\mathrm{SFT}}(\mathcal{L}_{\mathrm{sup}}; z_k), & q_k = \mathrm{SFT}, \\ \mathrm{Optimize}_{\mathrm{RL}}(J_{\mathrm{rew}}; z_k), & q_k = \mathrm{RL}. \end{cases} \tag{32}$$

For parameter post-training, $z = \theta$; for Harness Training, $z = H$. Equation 32 represents updates to shared persistent state. Our empirical composition study examines Harness-SFT followed by Harness-RL; it makes no claim that this schedule is universally optimal.

Feedback type and editable scope remain separate choices. A Skill can be optimized from rewards, and a full harness can be distilled from demonstrations. We vary these axes independently and specify the initialization, data source, modifier, selection rule, and budget for each comparison.

B Experimental Setting

B.1 Roles Held Fixed Across Experiments

Four roles are kept separate throughout. The task model is frozen; the trained harness is the persistent state that Harness-SFT and Harness-RL update; the modifier is an independent coding agent that proposes edits to that state; and the teacher supplies verified target trajectories. The trained harness is never the modifier that edits it, and the teacher never supplies reward. Table 6 lists the instantiations used in this paper.

Table 6: Model and Harness roles used across experiments.

Role	Instantiation	Status
Task model	Qwen3.6-35B-A3B (primary); Qwen3.6-9B (STHT transfer) (Qwen Team, 2026b)	Frozen throughout each training and evaluation run.
Trained harness	Pi (primary); Codex (transfer)	The persistent state updated by Harness-SFT and Harness-RL.
Modifier	A coding agent instantiated with Pi or Codex, with GPT-5.5 behind it (OpenAI, 2026b)	Frozen; a separate process from the trained harness.
Teacher	GPT-5.5 (primary); Qwen3.5-122B-A10B (transfer) (Qwen Team, 2026a)	Frozen; supplies verified target trajectories only.

B.2 Initial Harnesses

We instantiate the frozen task model with two initial harnesses, Pi and Codex (Pi Maintainers, 2026; OpenAI, 2026a). Pi is the primary initial harness used in RQ1–RQ4. Codex provides a second initial-harness setting that tests whether the findings generalize across harness implementations. Each harness retains its native prompt structure, context policy, tool-call serialization, and control loop, while benchmark adapters expose the same semantic actions and verifiers.

B.3 Benchmarks

We evaluate Harness Training on four benchmarks covering structured artifact manipulation, visual document understanding, embodied interaction, and executable code generation. These benchmarks differ in action space, trajectory length, observability, verifier structure, and reward density. Together, they allow us to examine whether the same Harness-SFT and Harness-RL principles hold across distinct forms of agent behavior.

SpreadsheetBench. SpreadsheetBench evaluates agents on real-world spreadsheet manipulation from a natural-language instruction and an input workbook (Ma et al., 2024). Tasks require coordinated changes to cell values, formulas, and formatting while preserving unrelated workbook content. We place each workbook in an isolated task workspace and provide semantic operations for reading and writing cells, inspecting formulas and styles, saving the output workbook, and verifying the result. The verifier evaluates the final saved workbook, so success requires the agent to select the correct operations, maintain artifact state across multiple steps, and produce a valid output file. Our split contains 60 training, 40 validation, and 280 test tasks. We use SpreadsheetBench to study trajectory pairing, supervision scale and source, editable scope, and transfer between Pi and Codex Harnesses.

DocVQA. DocVQA evaluates question answering over document images, where answers must be recovered from textual content and document layout (Mathew et al., 2021). Because the original benchmark evaluates final answers rather than interactive agent behavior, we adapt each instance into an instrumented multi-step pilot. The resulting interface provides crop, OCR, evidence-selection, and answer actions. The agent must locate relevant image regions, extract their text, identify supporting evidence, and submit a final answer. We evaluate each execution using answer and evidence checks. This setting tests whether Harness Training can improve perception and tool orchestration when the information required to solve a task is not directly exposed to the agent. Our split contains 107 training, 53 validation, and 374 test tasks.

ALFWorld. ALFWorld connects text-based interaction with embodied household environments (Shridhar et al., 2021). An agent receives textual observations and completes household goals through sequences of navigation and object-manipulation actions. Tasks require the agent to track environment state and plan over long action sequences. They also provide a sparse binary success signal, making useful Harness edits difficult to identify from individual rollouts. We use ALFWorld to study reward-driven Harness updates, the interaction between Harness-SFT and

Harness-RL, successful-edit availability, and sensitivity to training-task selection and order. The end-to-end STHT evaluation uses 60 training and 40 validation tasks and reports performance on 140 seen and 134 unseen test tasks. Mechanism studies use the experiment-specific subsets described in their corresponding protocols.

EvoAgentBench Algorithmic Reasoning. EvoAgentBench evaluates whether agents can transfer reusable procedures across tasks and domains (Gao et al., 2026). We use its Code Implementation adapter for the Algorithmic Reasoning domain, which contains LiveCodeBench-based programming tasks evaluated by an official executable test runner. We select this setting because our Harness-RL experiments require an objective and reproducible task-level reward for every candidate Harness. Executable tests provide this signal under a fixed interface and verifier, allowing differences between training methods to be compared without simultaneously changing the environment or evaluation protocol. The setting also tests whether implementation, debugging, and verification procedures learned from training executions transfer to unseen programming tasks. We evaluate only this domain and do not claim coverage of the complete EvoAgentBench benchmark.

Table 7 summarizes the interfaces, data splits, and experimental roles of the four benchmarks. We preserve an official split when it provides the labels required by the experiment. Otherwise, we construct group-disjoint training and held-out test splits by workbook, document, environment layout, or task template using `split_seed=42`. We reserve a development subset for selecting shared budgets, thresholds, and training diagnostics. Teacher trajectories and self-generated traces are collected only after the data partition is fixed. Modifiers receive only training evidence, retained Harness updates are selected using training tasks, and the held-out test split is used only for final reporting.

Table 7: Benchmark interfaces, data splits, and experimental roles. Split sizes correspond to the end-to-end STHT evaluation. Mechanism studies use the subsets specified in their respective protocols.

Benchmark	Train/Val/Test	Agent interface and verifier	Primary experimental role
SpreadsheetBench	60/40/280	Cell, formula, and formatting operations with final-workbook verification	Supervision factors, editable scope, and cross-Harness transfer
DocVQA	107/53/374	Crop, OCR, evidence-selection, and answer actions with answer and evidence checks	Instrumented perception and tool-orchestration pilot
ALFWorld	60/40/274	Text navigation and object manipulation with sparse binary success	Harness-SFT and Harness-RL composition and reward-search mechanisms
EvoAgentBench Code Implementation	60/40/86	Program implementation with executable test-based verification	Transfer of persistent coding procedures to unseen tasks

B.4 Data Partitions

Every training split is divided 1:1 into the SFT task set $\mathcal{D}_1$ and the RL task set $\mathcal{D}_2$. When a training split has odd size, $\mathcal{D}_1$ receives the extra task. STHT identifies failed tasks and applies Harness-SFT on $\mathcal{D}_1$, and Harness-RL searches on $\mathcal{D}_2$. The development subset that fixes shared budgets and thresholds is drawn from $\mathcal{D}_1$, before any Harness update. Table 8 lists the partitions.

We preserve an official split when it exposes the labels required by the experiment. Otherwise, we form group-disjoint train and held-out test splits by workbook, document, environment layout, or task template. All splits are pre-defined and published as manifests; no split seed is used at experiment time. Teacher trajectories and self-generated traces are collected only after this partition. Modifiers receive training evidence, retained updates are chosen using training tasks, and the held-out split is used only for reporting.

Table 8: Data partitions. When a procedure composes Harness-SFT with Harness-RL, the training split is divided 1:1 into the SFT task set $\mathcal{D}_1$ and the RL task set $\mathcal{D}_2$; when the training split has odd size, $\mathcal{D}_1$ receives the extra task. Harness-SFT alone uses the full training split.

Benchmark	**Train**	$\mathcal{D}_1$ **(SFT)**	$\mathcal{D}_2$ **(RL)**	**Validation**	**Test**
SpreadsheetBench	60	30	30	40	280
DocVQA	107	54	53	53	374
EvoAgentBench (Code)	60	30	30	40	86
ALFWorld	60	30	30	40	140 seen + 134 unseen

B.5 Search Configuration

Table 9 lists the budgets and constants shared by the experiments in this paper. An experiment that changes one of them states the change in place and keeps the remainder fixed.

Table 9: Search configuration.

Setting	**Value**	**Used in**
Candidates per group m	8	Default for all runs.
Candidate-group scan	2, 4, 8, 16	Successful-edit availability study.
Task-model rollouts per run	256	Reward search and availability study.
Supervision-budget cap K	64	STHT failure-targeted supervision.
Nested trajectory budgets N	8, 16, 32, 48, All	Pairing and scale study.

B.6 Checkpoint and Threshold Selection

Each experiment declares exactly one selection rule in Table 10. Test tasks are evaluated only after the checkpoint has been chosen.

Table 10: Checkpoint and threshold selection rules.

Stage	**Primary criterion**	**Tie-break and constraints**
Harness-SFT	Lowest validation tool-call NED	Validation success must not fall below the initial harness; ties by higher validation success.
Harness-RL	Highest validation reward AUC	Ties by final validation reward.
STHT	Highest validation reward AUC	The reward threshold is fixed on the development subset before test.

B.7 Metrics

Tool-call normalized edit distance measures behavioral imitation, held-out success measures task completion, and reward AUC measures reward-search efficiency over the rollout budget. Successful-edit availability is the fraction of Harness-RL rounds whose candidate group contains at least one verifier-passing edit. T_q is the number of task-model rollouts required to reach the validation threshold fixed for a setting. Cost includes modifier calls and task-model rollouts. Cost is normalized within each setting, with one reference condition set to 1.000: Bandit Allocation in the update-rule comparison and STHT in the five-setting comparison.

B.8 Reproducibility

Every reported success rate, reward, NED, and AUC value is the mean over three independent runs that differ only in their random seeds. All conditions compared within a row or a figure use the same three seeds, and no result is reported from a single run. The seed set and the split manifests are released with the code.

C TRAJECTORY SUPERVISION AS HARNESS SFT

Harness-SFT names the training objective: retained harness edits are selected for better imitation of target behavior. In this study, we instantiate that objective by restricting the editable scope to the reusable Skill component S while holding the remaining harness components fixed. A Skill is therefore one form of editable harness state, not a separate training regime; the same Harness-SFT objective could instead edit a prompt or a broader harness.

C.1 SUPERVISION CONDITIONS

We run every condition independently from Codex and Pi. The task model, task split, semantic tool contract, trace count, modifier, candidate count, and update rule are held fixed within each comparison. Figures in this section report results for the Pi harness; per-harness results, including Codex, are retained for the cross-harness analysis in Section F.

We compare two sources of target behavior. *Teacher Harness-SFT* uses verified trajectories generated by Qwen3.5-122B-A10B or GPT-5.5. *Self Harness-SFT* uses the frozen task model's own previously successful executions as pseudo-demonstrations.

C.2 TRAINING DYNAMICS

Trajectory comparison. Pi and Codex expose different tool names and argument schemas, so this experiment compares behavior after mapping each raw tool call c to a canonical semantic event,

$$\psi(c) = (\text{operation}, \text{target}, \text{arguments}, \text{outcome}), \tag{33}$$

where benchmark adapters normalize paths, identifiers, cell ranges, equivalent operation names, and unordered arguments. For a trajectory $\tau = (c_1, \ldots, c_m)$, let

$$C_{\text{tool}}(\tau) = (\psi(c_1), \ldots, \psi(c_m)) \,. \tag{34}$$

For an executed trajectory $\hat{\tau}$ and paired target $\tau^\star$, tool-call normalized edit distance is

$$d^{\text{tool}}_{\text{NED}}(\hat{\tau}, \tau^\star) = \frac{D_{\text{edit}}(C_{\text{tool}}(\hat{\tau}), C_{\text{tool}}(\tau^\star))}{\max\{1, |C_{\text{tool}}(\hat{\tau})|\,, |C_{\text{tool}}(\tau^\star)|\}}, \tag{35}$$

where D_{edit} is unit-cost Levenshtein distance (Levenshtein, 1966). Lower NED indicates closer behavioral imitation, and zero denotes an exact canonical event match. We compute it on free-running executions of the frozen task model rather than teacher-forced tokens.

Does execution solve the task? Each benchmark also supplies a task verifier $V_x(\tau) \in \{0, 1\}$. For K_{eval} free-running rollouts per held-out task,

$$\text{Success}(H; \mathcal{D}) = \frac{1}{|\mathcal{D}| K_{\text{eval}}} \sum_{x \in \mathcal{D}} \sum_{k=1}^{K_{\text{eval}}} V_x(\tau_{x,k}), \tag{36}$$

where $\tau_{x,k} \sim \pi_{\theta_0, H, E}(\cdot \mid x)$ and compared harnesses share evaluation seeds. The two metrics therefore answer different questions: NED asks whether execution imitates the target trajectory, and held-out success asks whether it actually completes the task.

An SFT-like interpretation requires a measurable loss curve alongside the final task score. At update round r, we evaluate the current Skill without teacher forcing and compute train and validation NED against the paired target trajectories. Validation NED uses a development subset within training, separate from the examples used to propose candidate edits and from held-out test tasks. Within this diagnostic, the Correct Trace condition uses one fixed verified external-teacher source; Section C.3 later varies the source separately. The No-Trace baseline receives the same tasks and search budget but no target trace. Figure 3 jointly tracks behavioral loss and execution quality. A useful supervised signal should reduce validation NED before or together with held-out task improvement; a widening train–validation gap would expose overfitting to the demonstrations.

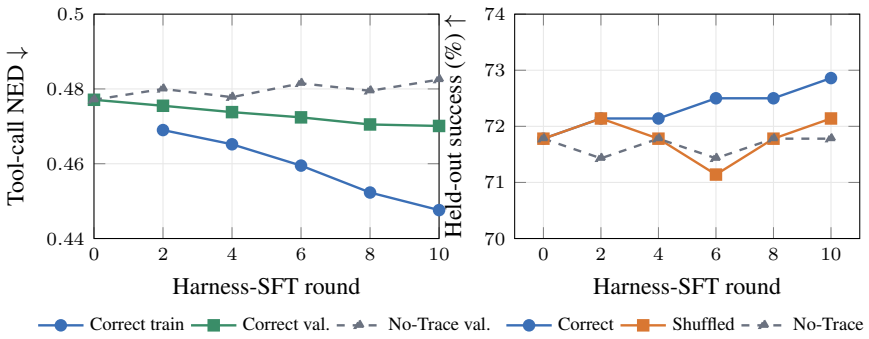

Figure 3: Harness-SFT training dynamics in the Skill-scoped instantiation, using SpreadsheetBench and Pi. NED measures target-trajectory imitation, held-out success measures task completion. The No-Trace baseline tests the effect of task-only editing.

The two panels test successive links in the Harness-SFT mechanism. The left panel asks whether repeated target-supervised updates reduce behavioral imitation loss on both training and validation tasks; the No-Trace curve tests whether task exposure and editing compute alone explain the change. The right panel asks whether lower imitation loss transfers to held-out task completion and compares correctly paired trajectories with an equally sized shuffled set. The intended joint signature is therefore lower validation NED and higher held-out success.

C.3 Pairing, Scale, and Teacher Source

We conduct this study on SpreadsheetBench using the Pi harness and edit only its persistent Skill state. We partition the data into 60 training tasks, 40 validation tasks, and 280 test tasks. Harness updates use only the 60-task training split. We evaluate retained checkpoints on the validation split and select the checkpoint with the lowest validation tool-call NED among checkpoints whose validation success does not fall below the initial Pi harness; ties are resolved by higher validation success. The selected checkpoint is evaluated on the 280-task test split with the shared evaluation seeds, and the split provides no feedback for training or checkpoint selection.

GPT-5.5 and Qwen3.5-122B-A10B provide verified trajectories for the same set of training tasks. We use nested trajectory budgets $N \in \{8, 16, 32, 48, \text{All}\}$, where every smaller subset is contained in the next larger subset and *All* denotes the complete available supervision set.

The *Correct* condition pairs each SpreadsheetBench task with its own teacher trajectory. The *Shuffled* condition applies a derangement within the same spreadsheet-operation family and difficulty stratum, so every task receives a valid trajectory from another task while preserving trace count and approximate trajectory length. Shuffled trajectories can still teach generic spreadsheet procedures, including reading source cells before writing, saving the workbook, and verifying outputs.

We compute canonical tool-call NED on all 40 validation tasks using fixed reference trajectories and report benchmark-verifier success on all 280 test tasks. Lower validation NED means that the selected Pi harness more closely reproduces the reference spreadsheet operations. Test success measures whether the final workbook state is correct and does not require exact trajectory imitation.

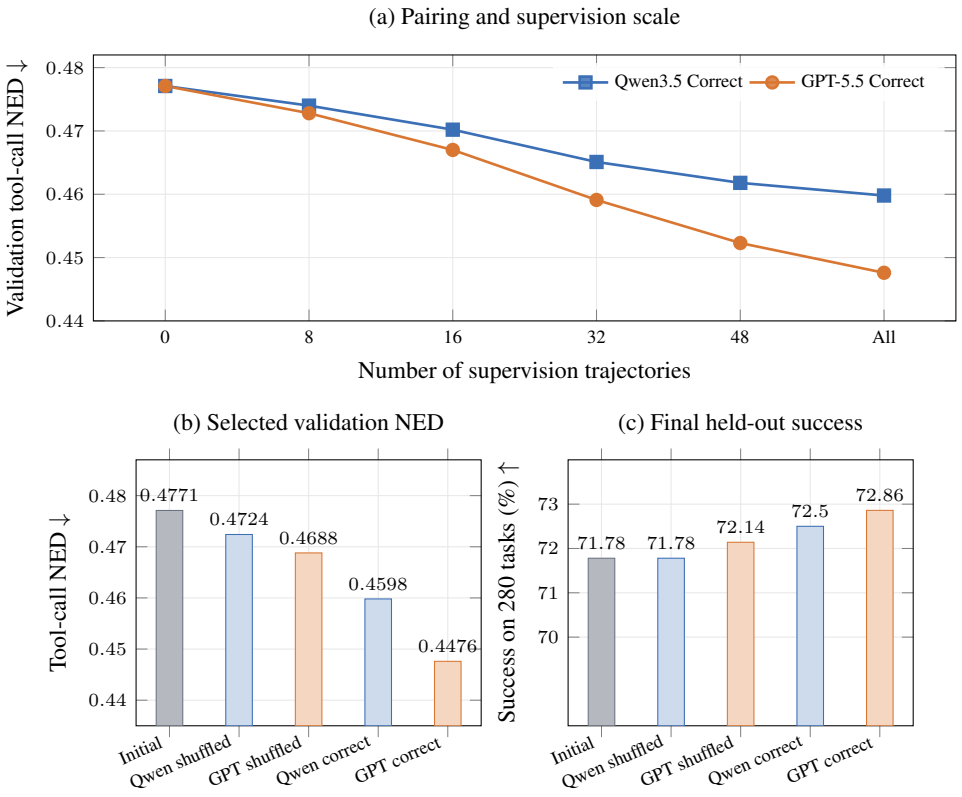

Figure 4: Pairing, scale, and teacher source on SpreadsheetBench with the Pi harness. Training uses 60 tasks, checkpoint selection uses 40 validation tasks, and final evaluation uses 280 test tasks. Panel (a) shows validation NED for Correct supervision across nested trajectory budgets. Panels (b) and (c) compare the checkpoints selected after using all available training trajectories. Shuffled supervision transfers generic spreadsheet procedures but remains weaker than task-aligned supervision.

Correctly paired supervision produces a clear scale effect. With Qwen3.5 supervision, validation NED decreases from 0.4771 for the initial Pi harness to 0.4598 after using all available training trajectories. GPT-5.5 produces a larger reduction to 0.4476. The gains become smaller near the largest supervision budget, indicating diminishing returns as additional demonstrations contribute fewer new spreadsheet-operation patterns.

Shuffled supervision also improves over the initial harness, but its effect is smaller. Qwen3.5 Shuffled reaches 0.4724 validation NED and 71.78% test success, while GPT-5.5 Shuffled reaches 0.4688 NED and 72.14% success. These gains are consistent with learning reusable SpreadsheetBench conventions from valid but mismatched trajectories. Correct task–trajectory pairing supplies the missing task-specific correspondence: Qwen3.5 Correct reaches 0.4598 NED and 72.50% success, while GPT-5.5 Correct reaches 0.4476 NED and 72.86% success.

The NED and success metrics answer different questions. Validation NED measures whether the harness reproduces the reference sequence of semantic spreadsheet actions and is used for checkpoint selection. Test success measures whether the selected harness produces the correct workbook state on unseen tasks. Correct supervision outperforming Shuffled on both metrics shows that the improvement does not come only from exposure to teacher-formatted trajectories; the correspondence between each task and its demonstrated behavior carries additional training signal.

D Reward-Driven Harness Training

We examine whether reward-driven harness updates exhibit RL-like behavior. An RL-like objective should produce predictable changes in retained behavior when the reward changes while the search

settings remain fixed. This test establishes an objective-level correspondence to RL, not equivalence to policy-gradient optimization. The tasks, search procedure, and metrics differ from Section C. We use benchmark-native success checks where available: workbook validation for SpreadsheetBench, embodied goal completion for ALFWorld, answer and evidence checks for the DocVQA pilot, and executable tests for EvoAgentBench Code Implementation.

Each verifier produces a normalized reward $R(x, \tau) \in [0, 1]$. For K_{eval} free-running rollouts per held-out task, we report

$$\overline{R}(H; \mathcal{D}) = \frac{1}{|\mathcal{D}| K_{\text{eval}}} \sum_{x \in \mathcal{D}} \sum_{k=1}^{K_{\text{eval}}} R(x, \tau_{x,k}), \tag{37}$$

where $\tau_{x,k} \sim \pi_{\theta_0, H, E}(\cdot \mid x)$ and compared harnesses share evaluation seeds.

To measure learning speed, let $0 = b_0 < \cdots < b_L = B$ be cumulative task-model rollout checkpoints and let $\bar{R}_\ell$ be mean held-out reward at b_ℓ. We compute normalized reward area under the curve by

$$\text{AUC}_R = \frac{1}{B} \sum_{\ell=0}^{L-1} \frac{\bar{R}_\ell + \bar{R}_{\ell+1}}{2} (b_{\ell+1} - b_\ell). \tag{38}$$

D.1 Reward Search and Editable Scope

The primary reward-search run starts from Codex and uses the default Pi coding-agent modifier backed by GPT-5.5. It samples eight candidates per round and selects them on training tasks. The reward-objective intervention restricts the editable scope to the Skill component, and the scope study compares

$$\Omega_{\text{skill}} = \{S\}, \qquad \Omega_{\text{full}} = \{P, S, T, W, L\}. \tag{39}$$

Both scope conditions follow the configuration in Appendix B; their difference isolates the effect of the searchable representation. The reward-objective intervention below uses the matched Skill scope Ω_{skill}, so its comparison reflects the reward objective alone.

D.2 Does Harness-RL Respond to the Reward Objective?

We now vary the reward objective and ask whether the retained Harness changes in the corresponding behavioral dimension. For every task–seed pair, the compared conditions differ only in the reward objective; all other settings follow Appendix B. For the first update, we generate one shared candidate group and score the same cached executions under every reward. We then run separate multi-round searches from the same initial state using matched random seeds.

Let $s(x, \tau) \in \{0, 1\}$ denote benchmark success, $c(\tau) \in [0, 1]$ normalized tool-call count, $\ell(\tau) \in [0, 1]$ normalized canonical trajectory length, and $v(x, \tau) \in [0, 1]$ valid verification coverage. We clip and normalize c and ℓ using benchmark-specific caps fixed on the development split before training. The coverage v is the fraction of benchmark-defined verification targets checked by valid tool outputs that are consumed before the final answer. We compare four selection objectives:

$$\begin{aligned} R_{\text{success}}(x, \tau) &= s(x, \tau), \\ R_{\text{cost}}(x, \tau) &= s(x, \tau)\big(1 - \lambda c(\tau)\big), \\ R_{\text{step}}(x, \tau) &= s(x, \tau)\big(1 - \lambda \ell(\tau)\big), \\ R_{\text{verify}}(x, \tau) &= s(x, \tau)\big((1 - \lambda) + \lambda v(x, \tau)\big), \end{aligned} \qquad 0 < \lambda < 1. \tag{40}$$

The success gate ensures that a cheap, short, or heavily checked failure cannot outrank a successful execution. Tool-call count and trajectory length are reported separately because the latter includes all canonical observable actions, not only external calls.

Table 11: Behavioral response to reward-objective changes on ALFWorld with the Pi harness.

Selection reward	Seen ↑	Unseen ↑	Tool calls ↓	Steps ↓	Verification ↑
Initial Harness	74.29%	83.58%	14.62	13.29	10.95%
Success-only	80.71%	91.79%	14.90	13.45	13.14%
Cost-aware	79.29%	89.55%	9.80	9.00	11.68%
Step-aware	80.00%	90.30%	10.20	8.20	12.04%
Verification-aware	81.43%	92.54%	15.40	13.70	64.96%

A reward-responsive optimizer predicts the targeted pattern in Table 11: cost-aware selection lowers calls, step-aware selection shortens trajectories, and verification-aware selection increases valid checks. Generic gains from the modifier's prior or repeated editing would instead produce similar behavioral changes across reward conditions. We report benchmark-native held-out success independently of the selection reward; improving the targeted training score while reducing held-out success indicates reward misspecification rather than useful task learning.

Together, these reward interventions show that Harness-RL responds to reward design in the directions expected of reward-driven learning. Changing the reward changes the retained behavior in the predicted dimension while leaving the discrete search algorithm distinct from parameter RL.

D.3 Reward-Search Update Rules on ALFWorld

We compare five reward-search update rules on ALFWorld using Pi as the initial harness and GPT-5.5 as the fixed harness modifier. The frozen task model, editable Harness scope, reward function, candidate-generation prompt, and environment interface remain unchanged across methods. We use 60 training tasks for Harness updates, 40 validation tasks for learning-curve measurement and checkpoint selection, and 274 test tasks for final evaluation. The test set contains 140 seen and 134 unseen tasks. Test tasks provide no feedback to the modifier and are evaluated only after checkpoint selection.

All methods optimize the same binary ALFWorld task-success reward. A proposed patch must apply cleanly, load the modified Pi harness, pass a benchmark-independent smoke test, and solve the current training task before it can become persistent. Failed candidate trajectories, environment feedback, and verifier diagnostics remain available to the next proposal round. The methods differ only in how they organize candidate comparisons and allocate subsequent search: Parent Acceptance follows one greedy lineage; Group-Relative selects candidates using within-group reward differences; DPO-inspired Pairwise feeds winner–loser Harness pairs back to the modifier; Population Search preserves several high-reward lineages; and Bandit Allocation assigns later candidate evaluations to promising lineages while retaining an exploration bonus.

Each method receives the schedules and seeds listed in Appendix B; only the update rule differs. We report validation reward AUC over cumulative training rollouts. For checkpoint selection, we choose the checkpoint with the highest validation AUC, breaking ties by final validation reward. We then report binary task success on the 140 seen and 134 unseen test tasks. Cost includes GPT-5.5 modifier calls and task-model rollouts and is normalized within this setting, with Bandit Allocation set to 1.00.

Table 12: Reward-search update rules for the Pi harness on ALFWorld. Validation AUC is computed on 40 validation tasks. Test success is reported separately on 140 seen and 134 unseen tasks; the overall column aggregates all 274 test tasks. Cost is normalized within this setting, with Bandit Allocation set to 1.00.

Update rule	Val. AUC ↑	Seen ↑	Unseen ↑	Overall ↑	Cost ↓
Parent Acceptance	0.742	80.71%	91.79%	86.13%	**0.72**
Group-Relative	0.775	82.86%	94.78%	88.69%	0.82
DPO-inspired Pairwise	0.764	82.14%	94.03%	87.96%	0.79
Population Search	0.797	84.29%	96.27%	90.15%	0.94
Bandit Allocation	**0.813**	**86.43%**	**97.01%**	**91.61%**	1.00

Parent Acceptance provides the lowest-cost baseline but obtains the lowest validation AUC and test success. Its single-lineage update rule can stall when one accepted edit places subsequent proposals in an unproductive region of the Harness space. Group-Relative improves over Parent Acceptance by comparing candidates within each proposal group and using relative outcomes to guide the next edit. DPO-inspired Pairwise also improves the search by exposing GPT-5.5 to explicit winner–loser contrasts, although it trails Group-Relative in validation AUC and final success.

Population Search and Bandit Allocation obtain the strongest results. Population Search preserves several viable Harness lineages, reducing dependence on one early edit. Bandit Allocation achieves the highest validation AUC and test accuracy by concentrating later evaluations on lineages with stronger observed reward while continuing to explore less-tested alternatives. This gain comes with the highest end-to-end cost. The table therefore shows a quality–cost trade-off rather than one update rule dominating every criterion.

The unseen split has higher accuracy than the seen split for every method in this evaluation. This difference should be interpreted as a property of the task composition in the two ALFWorld test subsets, not as evidence that unseen environments are generally easier or that the Harness improves more on unseen tasks. Comparisons between update rules remain valid because all methods use the same seen and unseen task sets and evaluation seeds.

D.4 Skill Search or Full-Harness Search?

Figure 5 compares the two editable scopes. Full-Harness search obtains higher held-out success in all four settings, where success requires changes to tool descriptions, verification, context construction, or control flow beyond reusable procedures. Skill-only search has a smaller space and may be more efficient when reusable procedural knowledge is the primary bottleneck; the present evaluation does not isolate that regime.

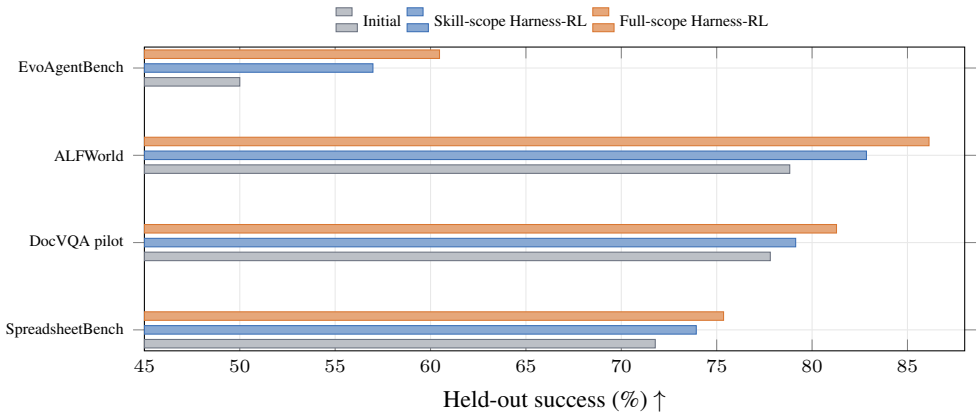

Figure 5: Effect of editable scope across the four settings with the Pi harness. Initial, Skill-scope Harness-RL, and Full-scope Harness-RL are measured in every setting.

E Composing Harness-SFT and Harness-RL

The previous analyses examine trajectory supervision and reward search separately. Their shared persistent state lets us examine how the objectives interact when composed. We study supervised initialization followed by reward search while every model remains frozen:

$$H_0 \xrightarrow{\text{Harness-SFT}} H_{\text{sup}} \xrightarrow{\text{Harness-RL}} H_{\text{rew}}. \tag{41}$$

Existing Skill methods emphasize trajectory-to-Skill updates, while Full-Harness optimizers emphasize execution feedback and reward. Equation 41 composes these two lines at the artifact level rather than training a policy or a Harness editor.

E.1 Does Harness-SFT Help Harness-RL?

We study supervised initialization followed by reward search on ALFWorld using Pi as the initial harness. GPT-5.5 supplies the verified trajectories for Harness-SFT and serves as the fixed harness modifier through separate calls. We use a 60-task training pool and 40 validation tasks for learning-curve measurement. The task model remains frozen throughout the experiment.

Both conditions use the Harness-RL schedule and seeds listed in Appendix B.

We report two protocols. The matched-scope protocol restricts both Harness-SFT and Harness-RL to the Skill component. It tests whether supervised initialization helps when the editable state is identical across stages. The practical protocol keeps Harness-SFT Skill-scoped and allows the subsequent Harness-RL stage to edit the full Pi harness. In both protocols, the Harness-SFT stage uses $\mathcal{D}_1$ and the Harness-RL stage uses $\mathcal{D}_2$, while Harness-RL samples new trajectories and receives no target actions.

Learning curves report mean binary reward on the 40-task validation split against cumulative Harness-RL rollouts. The horizontal axis starts when reward search begins. Validation AUC measures reward-search efficiency over the matched rollout budget.

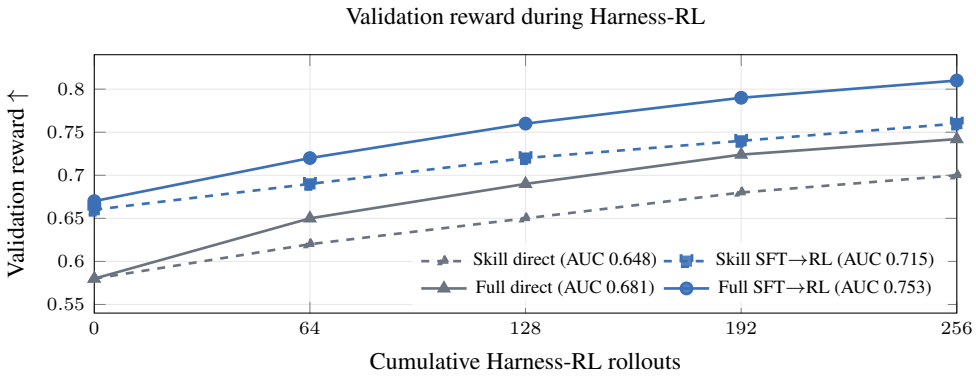

Figure 6: Harness-SFT initialization followed by Harness-RL on ALFWorld. The Harness-SFT stage uses $\mathcal{D}_1$ and the Harness-RL stage uses $\mathcal{D}_2$, and reward curves are measured on 40 validation tasks. Color distinguishes initialization, while dashed and solid lines denote Skill-only and Full-Harness reward search. Legend entries report validation AUC over the matched rollout budget.

Supervised initialization raises the reward level from which search begins under both editable scopes. At matched Skill scope, validation AUC rises from 0.648 to 0.715, and the two-stage pipeline starts reward search at reward 0.660 rather than 0.580. This comparison holds the editable state fixed, so the difference cannot be attributed to a larger Harness search space.

The same pattern is stronger in the practical Full-Harness protocol. The two-stage pipeline starts at 0.670 rather than 0.580, and validation AUC increases from 0.681 to 0.753. Full-Harness search also gains more reward over the same budget than Skill-only search from the corresponding initialization, indicating that the broader editable state provides additional useful update directions on ALFWorld.

The curves separate two effects. The two-stage pipeline begins from a stronger Harness, as shown by its higher reward at rollout zero, and it keeps an advantage throughout the budget. The per-rollout gap narrows rather than widens in both scopes, so this experiment establishes a level effect rather than a search-efficiency effect. Section E.2 isolates the efficiency question by matching the starting reward.

E.2 Controlling for Starting Reward

The preceding experiment starts Teacher SFT→RL from a Harness with higher validation reward than the initial Pi harness. This difference reduces the remaining optimization distance and may

explain part of the later AUC gain. We therefore compare the supervised initialization with a reward-only checkpoint that has nearly the same starting reward and per-task reward profile on ALFWorld.

We divide the 60 training tasks into 30 initialization tasks and 30 downstream reward-search tasks. Teacher SFT uses correctly paired trajectories from the initialization split and updates only the persistent Skill. To construct the no-SFT baseline, we run Skill-scoped reward-only warm-up from the same initial Pi harness on the same 30 initialization tasks and save checkpoints after 0, 16, 32, 48, 64, 80, 96, and 128 task-model rollouts. The warm-up receives task reward, environment feedback, and verifier diagnostics but no teacher trajectory.

We divide the 40 validation tasks into two disjoint 20-task folds. In the first run, one fold is used to select the reward-matched checkpoint and the other is used to measure the downstream learning curve. We then exchange the fold roles and average the resulting curves. This cross-fitting procedure prevents the tasks used to choose the matched checkpoint from also determining its reported downstream AUC.

We set $K_{\text{eval}} = 5$ evaluation rollouts per task throughout. Let H_{sup} denote the supervised initialization and let $\mathcal{B}_{\text{no-SFT}}$ denote the reward-only checkpoint bank. We first retain checkpoints whose mean matching-fold reward lies within $\epsilon = 0.02$ of H_{sup}. Among them, we choose the checkpoint with the smallest mean absolute per-task reward difference:

$$H_{\text{match}} = \arg \min_{H \in \mathcal{B}_{\text{no-SFT}} : |\overline{R}(H;\mathcal{D}_{\text{match}}) - \overline{R}(H_{\text{sup}};\mathcal{D}_{\text{match}})| \le 0.02} \frac{1}{|\mathcal{D}_{\text{match}}|} \sum_{x \in \mathcal{D}_{\text{match}}} |\bar{R}_x(H) - \bar{R}_x(H_{\text{sup}})|. \tag{42}$$

If no checkpoint satisfies the tolerance, we select the checkpoint with the smallest mean-reward difference and report the residual mismatch. This tolerance corresponds to two percentage points, which permits small evaluation noise while excluding checkpoints with a materially different starting reward.

Both initializations use the downstream schedule and seeds listed in Appendix B.

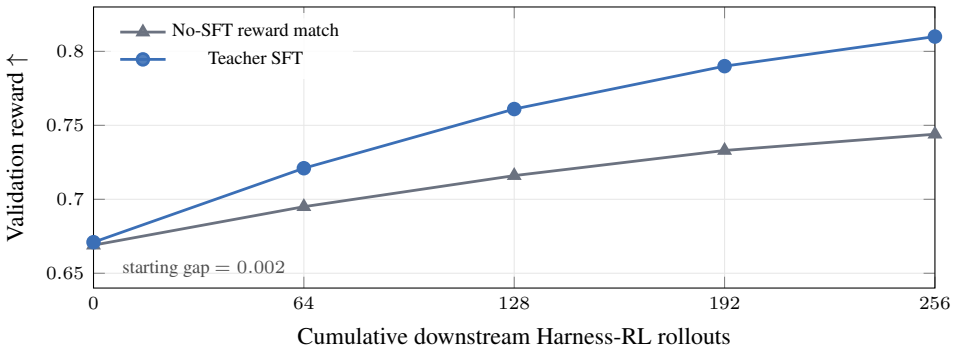

Figure 7: Teacher SFT improves downstream search after reward matching.

The two initializations have a mean starting-reward difference of 0.002 and a per-task profile gap of 0.021. These values make their initial aggregate performance and task-level strengths closely comparable. The supervised initialization improves more rapidly under the same downstream reward-search budget. Its validation AUC is 0.753, compared with 0.713 for the no-SFT reward-matched initialization, an absolute gain of 0.040.

Starting reward alone cannot explain the residual AUC advantage. The two Harnesses reach similar initial validation performance through different update signals: one consolidates target trajectories, while the other reaches the matched score through reward-only warm-up. Their later optimization behavior can therefore differ even when their initial rewards are similar. This result motivates the next experiment, which tests whether supervised initialization increases the frequency with which reward search generates at least one task-solving candidate edit.

E.3 HARNESS-SFT IMPROVES SUCCESSFUL-EDIT AVAILABILITY

The reward-matched experiment shows that Teacher SFT improves downstream validation AUC even when the two initializations have nearly identical starting reward and task-level reward profiles. We now inspect the candidate-generation process during Harness-RL. The experiment tests whether supervised initialization makes task-solving Harness edits easier to find under the same candidate budget.

Let $\mathcal{G}(H;\mathcal{D})$ be the candidate-evaluation rounds of a Harness-RL run initialized at Harness H on downstream task set $\mathcal{D}$. In round g, the modifier proposes m candidate edits for task x_g from the current parent Harness. Candidate j produces trajectory $\tau_{g,j}$ and verifier outcome

$$v_{g,j} = V_{x_g}(\tau_{g,j}) \in \{0,1\}. \tag{43}$$

Failed trajectories, environment feedback, tool errors, and verifier diagnostics remain available when the next candidate group is proposed. A round offers a task-solving edit when at least one candidate passes the task verifier. We measure the fraction of such rounds as

$$\mathcal{A}(H;\mathcal{D}) = \frac{1}{|\mathcal{G}(H;\mathcal{D})|} \sum_{g \in \mathcal{G}(H;\mathcal{D})} \mathbf{1}\left[\max_{1 \le j \le m} v_{g,j} = 1\right]. \tag{44}$$

A passing candidate becomes eligible for selection. Retention additionally requires the update rule's reward comparison and persistence checks. An all-failure group can guide the next proposal, but it cannot change the persistent Harness in the current round. We therefore also measure the number of task-model rollouts consumed before the first retained update, and we report the availability defined in Equation 44.

We use the same two reward-matched initializations as in Section E.2, the No-SFT reward match and Teacher SFT, and run the same Full-Harness reward-search procedure on the same 30 downstream training tasks. Within each value of m, the two conditions follow the configuration in Appendix B.

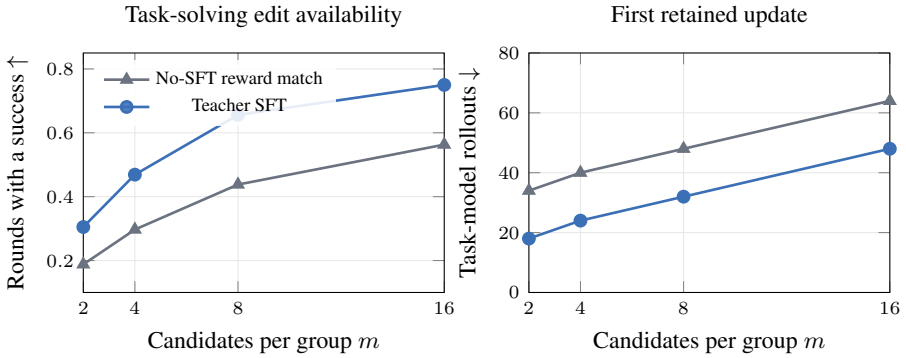

Figure 8: **Teacher SFT makes task-solving edits easier to find after reward matching.** At every candidate-group size, supervised initialization increases the fraction of rounds containing a verifier-passing edit and reduces the rollout delay before the first retained update.

Teacher SFT increases task-solving edit availability at every candidate-group size. At $m = 8$, 0.438 of the No-SFT candidate groups and 0.656 of the Teacher-SFT groups contain at least one passing edit, corresponding to 14 of 32 and 21 of 32 groups in the reported run. The first retained update appears after 48 and 32 task-model rollouts, respectively, a reduction of 33.3%. Similar differences appear at $m = 2$, $m = 4$, and $m = 16$.

The two panels measure different effects of candidate-group size. Larger groups increase the chance that a round contains a passing edit. The rollout-delay metric charges every candidate evaluated before a group commits an update, so increasing m also raises the cost of completing each round. We compare the two initializations separately at each fixed value of m; delay values across different group sizes mix edit availability with group-completion cost.

These results provide a mechanism for the reward-matched AUC difference. Teacher SFT changes both the initial validation score and the local edit landscape. From the reward-matched state, a fixed candidate budget produces task-solving edits more often. More candidate groups can then enter the update rule's selection stage, and the first persistent improvement occurs earlier.

F Task Order and Cross-Harness Transfer

Task-order sensitivity. We test task-order sensitivity on ALFWorld using Pi as the initial Harness. Qwen3.6-35B-A3B remains the frozen task model, and GPT-5.5 provides the teacher trajectories and Harness edits through separate calls. From the 60-task training split, we label 30 tasks as foundational and 30 as compositional. Foundational tasks require direct object localization, pickup, placement, or inspection without changing object state. Compositional tasks require cleaning, heating, cooling, or coordinating multiple objects before the final goal can be completed.

We compare three deterministic orders over this same multiset: all foundational tasks followed by all compositional tasks (F→C), the reverse order (C→F), and an interleaved order that alternates the two groups while preserving a fixed task-ID order within each group. The three orders are fixed and deterministic; each order is averaged over the same three execution seeds. We run the comparison separately for Skill-scoped Harness-SFT and Skill-scoped Harness-RL. Within each regime, every order starts from the same Pi Harness and uses the same tasks, modifier, number of updates, and total budget. Harness-SFT keeps every task paired with the same verified target trajectory; Harness-RL keeps the binary success reward, candidate count, and rollout allocation fixed. The 40-task validation split selects the checkpoint, and the 274-task test split is used only for final reporting. Foundation and Composition report success within each task family, and Macro is their unweighted mean; Table 13 reports the result.

Table 13: Task-order sensitivity on ALFWorld with Pi. Both regimes are Skill-scoped. Every column uses the same 60-task training split and differs only in its deterministic order. Cells report test success in percent; Macro is the unweighted mean of the Foundation and Composition task-family scores. The 274-task test split contains n_F foundational and n_C compositional tasks.

Regime	Test family	F→C	C→F	Interleaved
Harness-SFT	Foundation	78.4%	**82.7%**	81.9%
	Composition	82.9%	77.3%	**82.7%**
	Macro	80.7%	80.0%	**82.3%**
Harness-RL	Foundation	83.8%	**87.3%**	86.6%
	Composition	**88.4%**	82.7%	87.8%
	Macro	86.1%	85.0%	**87.2%**

The final task block leaves a visible signature in both regimes. C→F produces the highest Foundation score but the lowest Composition score, whereas F→C reverses this pattern. Interleaving limits this recency effect and obtains the highest family-macro result for both Harness-SFT and Harness-RL. The family-macro spread across orders is 2.3 points for Harness-SFT and 2.2 points for Harness-RL; we report means over the shared execution seeds and treat this experiment as a diagnostic of order sensitivity rather than an estimate of a population effect. Thus, the retained Harness depends not only on the training-task multiset but also on when each task family supplies its update signal.

Cross-harness transfer. We test cross-Harness supervision on SpreadsheetBench using its 60 training, 40 validation, and 280 test tasks. Qwen3.6-35B-A3B is the frozen task model. GPT-5.5 executes each supervised task under either the Pi or Codex Harness to produce verified source trajectories, and a separate modifier call, using the Pi modifier throughout, updates the student Harness. We canonicalize each trajectory into semantic spreadsheet operations before it is supplied to Harness-SFT, removing source-specific tool names, argument serialization, and control-loop messages.

Every supervised cell uses the training tasks, trace count, Skill-scoped update procedure, and checkpoint rule listed in Appendix B. Diagonal cells measure matched-Harness supervision, and off-

diagonal cells change only the Harness under which the teacher trajectory was generated. Diagonal cells measure matched-Harness supervision, and off-diagonal cells change only the Harness under which the teacher trajectory was generated.

Table 14: Cross-Harness Harness-SFT on SpreadsheetBench. Columns are the student Harness, and rows are the Harness used to generate teacher trajectories. Cells report success on 280 test tasks and the absolute gain over the corresponding No SFT student.

Trace-source Harness	**Pi student**	**Codex student**
No SFT	71.78%	76.43%
Pi	**72.86% (+1.08 pp)**	76.79% (+0.36 pp)
Codex	72.50% (+0.72 pp)	**78.21% (+1.78 pp)**

Both off-diagonal cells improve over the corresponding untrained student, showing that canonicalized teacher behavior transfers across Harness boundaries. The gap indicates that semantic canonicalization preserves much of the task procedure but does not remove every source-specific choice encoded in the trajectory.

G Signal-Targeted Harness Training: Complete Procedure

Algorithm 1 states the complete STHT procedure. Stage 1 selects the tasks that the initial harness fails on $\mathcal{D}_1$ and applies repeated Skill-scoped Harness-SFT updates to the same persistent harness. Stage 2 initializes Harness-RL from the resulting state and optimizes reward on $\mathcal{D}_2$. The task model θ_0 remains frozen in both stages, and the modifier G is the only source of proposed edits.

Algorithm 1 Signal-Targeted Harness Training (STHT)

Require: initial harness H_0; SFT task set $\mathcal{D}_1$; RL task set $\mathcal{D}_2$; frozen task model θ_0; environment E; teacher $\mathcal{T}$; modifier G; editable scopes $\Omega_{\mathrm{SFT}} = \{S\}$ and Ω_{RL}; supervision-budget cap K; reward R; update rule U_{RL}; round budget T

Ensure: retained harness $H^\star$

Stage 1: failure-targeted Harness-SFT

for all $x \in \mathcal{D}_1$ **do**
  $\tau_x^{(0)} \sim \pi_{\theta_0,H_0,E}(\cdot \mid x)$; $r_x^{(0)} \leftarrow R(x, \tau_x^{(0)})$
end for
$\mathcal{D}_{\mathrm{fail}} \leftarrow \{x \in \mathcal{D}_1 : r_x^{(0)} = 0\}$
$K_H \leftarrow \min\{K, |\mathcal{D}_{\mathrm{fail}}|\}$
$\mathcal{D}_{\mathrm{target}} \leftarrow (x_1, \ldots, x_{K_H}) \subseteq \mathcal{D}_{\mathrm{fail}}$
$H \leftarrow H_0$
for $i = 1$ **to** K_H **do**
  $\tau^\star_{x_i} \leftarrow \mathcal{T}(x_i)$ ▷ verified teacher trajectory
  $H \leftarrow \mathrm{HarnessSFT}_{\Omega_{\mathrm{SFT}}}\left(H, (x_i, \tau^\star_{x_i})\right)$ ▷ retained edit minimizes Eq. 3
end for
$H_{\mathrm{sup}} \leftarrow H$; $H_0^{\mathrm{RL}} \leftarrow H_{\mathrm{sup}}$; $\mathcal{E} \leftarrow \emptyset$

Stage 2: Harness-RL from the supervised state

for $t = 1$ **to** T **do**
  sample $x_t \in \mathcal{D}_2$
  $H'_t \sim G_{\Omega_{\mathrm{RL}}}(\cdot \mid H_t, x_t, \mathcal{E})$
  $\tau'_t \sim \pi_{\theta_0,H'_t,E}(\cdot \mid x_t)$; $r'_t \leftarrow R(x_t, \tau'_t)$
  $H_{t+1} \leftarrow U_{\mathrm{RL}}(H_t, H'_t, r'_t)$ ▷ retains H'_t or H_t
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{(H'_t, \tau'_t, r'_t)\}$
end for
return $H^\star \leftarrow H_T$

Three details in Algorithm 1 follow the main-text description. First, $\mathcal{D}_{\mathrm{target}}$ is drawn from the failure pool rather than from the full task set, so the supervision budget is spent on tasks with a negative

targeting signal; setting $K \geq |\mathcal{D}_{\text{fail}}|$ recovers the all-failure setting. Second, every update in Stage 1 starts from the state produced by the previous task, so the persistent Skill accumulates edits across tasks instead of being re-estimated per task. Third, Stage 2 preserves the configured Harness-RL update rule, including optimizers that evaluate candidate groups: when the update rule compares $m > 1$ candidates, the proposal step is repeated m times and the group selection rule is applied to the resulting outcomes.

The Uniform baseline shares Algorithm 1 except that $\mathcal{D}_{\text{target}}$ is sampled uniformly from $\mathcal{D}_1$ with the same realized size K_H. The comparison therefore isolates the task-selection signal rather than the amount of supervision.

H Case Study: STHT from Harness-SFT to Full-Harness RL

We trace one completed SpreadsheetBench + Pi run to illustrate how the two stages of STHT make complementary persistent updates. The actor is the frozen Qwen3.6-35B-A3B model and the teacher/modifier is GPT-5.5. STHT first selects tasks failed by the initial Pi Harness, applies ten Skill-scoped Harness-SFT rounds, and then initializes ten rounds of reward-driven search over all five Harness components. This is a qualitative analysis of one run rather than an independent causal estimate.

Table 15: Configuration of the traced STHT run.

Item	Configuration
Dataset and initial Harness	SpreadsheetBench with Pi
Task model	Qwen3.6-35B-A3B, frozen throughout training
Teacher and modifier	GPT-5.5 in separate trajectory-generation and Harness-editing roles
Harness-SFT stage	Ten Skill-scoped rounds on tasks failed by the initial Harness
Harness-RL stage	Ten reward-driven rounds over components P, S, T, W, and L
Purpose	Illustrate how supervision and reward search make complementary persistent updates

H.1 Stage 1: Teacher Traces Consolidate Workbook Procedures into the Skill

The first four SFT rounds retained no update. In round 5, the accepted candidate cited evaluator-verified teacher contrasts from tasks `247-24`, `46121`, `53647`, and `57113`. A representative failure, task `247-24`, required row deletion and insertion, conditional value updates, a lookup into a second sheet, and preservation of derived billing columns. The actor completed the structural edits but retained formula strings after changing their dependencies. Because the runtime had no spreadsheet recalculation engine, the evaluator observed `None` in 172 expected cells, including `Main!I2`, and assigned 725/897 cell matches (accuracy .8082) rather than hard success. The teacher instead materialized the derived values and reopened the saved workbook in `data_only=True` mode to verify evaluator-visible values.

The diagnosis attributed the error to missing procedural knowledge in component S. Candidate `sft-iter-05-s01` replaced the trainable Skill with a general checklist rather than copying the task solution. Its central rules are summarized below.

Retained Skill Update

- Inspect the workbook before editing and infer the complete destination range.
- Write concrete values when cached formula results cannot be created.
- Detect formulas using supported `openpyxl` fields.
- Perform variable-length and structural edits in a stable order.
- Reopen the exact output in formula and `data_only` modes before finishing.

Table 16: Observed evidence for the accepted round-5 Harness-SFT update. The training example motivates the patch; acceptance is determined on the disjoint selection fold.

Evidence	Parent Harness	Retained Candidate
Training trace, task `247-24`	Structural edits were produced, but formula cells had missing cached values; accuracy was .8082 (725/897), with 172 mismatches.	The retained Skill explicitly required concrete evaluator-visible values and dual-mode post-save verification.
Selection example, task `170-13`	The actor generated only 151/200 expected output cells (accuracy .755; hard failure) and used 15 tool calls.	The candidate generated all 200 expected cells (accuracy 1.0; hard success) using eight tool calls.
Selection fold	Mean NED .5471 and hard success 6/8.	Mean NED .4486 and hard success 6/8; one failure became a success and one previous success regressed.

Harness-SFT Selection Decision

The candidate was retained because selection-fold mean NED decreased from .5471 to .4486 while hard success remained at 6/8. It was not selected merely because it addressed the motivating task: one failure became a success and one previous success regressed under the configured success non-regression rule.

Across the ten SFT rounds, five Skill updates were accepted. The resulting checkpoint, `sft-iter-10-s05`, became H_{sup} for the reward stage.

H.2 Stage 2: Reward Search Corrects a Residual Output-Placement Error

Full-Harness RL began from a copy of H_{sup} and could propose separate candidates for P, S, T, W, and L. By RL round 8, the retained Harness already included the broad SFT-derived workbook procedures and several reward-selected refinements, but task `46897` exposed a more specific steering error. The workbook contained a small control panel with `H1=Coulum Header` (as spelled in the workbook), `I1=Fruits`, `H2=Value`, `I2=Apple`, and `H3=Result`. The actor correctly counted three instances of "Apple" and saved a valid workbook, but wrote the integer to `H3`, overwriting the label. The evaluator expected the value in the adjacent cell `I3`, observed `None`, and returned zero reward.

The modifier attributed this failure to component P: the model had the tools and procedural capability to solve the task, but the always-visible prompt did not make label/value layouts sufficiently salient. Candidate `stht-rl-full-iter-08-p01` added the following general instruction:

Retained Component-P Update

In label/value panels or small summary blocks, identify which cells are labels versus output slots; do not overwrite labels such as Result, Value, Total, or Header when the intended result cell is the neighboring blank/value cell.

Under this candidate, the actor inspected the same layout, explicitly inferred that the result belonged in `I3`, wrote `I3=3`, saved `output.xlsx`, and reopened it to verify that `H3` still contained "Result" while `I3` contained 3. The task changed from accuracy 0 and hard failure to accuracy 1.0 and hard success.

Table 17: Observed evidence for the accepted round-8 Full-Harness RL update.

Evidence	Parent Harness	Retained Candidate
Layout interpretation	Correctly computed the count but treated the `H3` label cell as the output destination.	Preserved `H3=Result` and selected the adjacent blank value cell `I3`.
Write and verification	Wrote `H3=3`; evaluator found `I3=None` and returned hard success 0.	Wrote `I3=3`, reopened the workbook, checked both cells, and obtained hard success 1.
Selection fold	Hard success 6/8 and mean accuracy .8738.	Hard success 8/8 and mean accuracy 1.0; tasks `262-17` and `577-40` changed from failure to success with no success-to-failure transition.

Harness-RL Selection Decision

The candidate was retained because selection-fold hard success increased from 6/8 to 8/8 and mean accuracy increased from .8738 to 1.0. Two failures changed to successes, with no success-to-failure transition.

Across the complete run, the transition from the final SFT checkpoint to the final Full-RL checkpoint raised internal-holdout hard success from 12/20 (60.0%) to 15/20 (75.0%) and mean accuracy from .7344 to .8821. On the 280-task final test, hard success rose from 200/280 (71.43%) to 206/280 (73.57%), while mean tool calls decreased from 9.46 to 8.63.

Mechanism Takeaway

Harness-SFT consolidated reusable workbook procedures into component S. Starting from this supervised state, Harness-RL localized a residual outcome-relevant error to component P and retained a smaller steering intervention. The final-test comparison contains both gains and regressions, so the case illustrates complementary update roles rather than monotonic task-level improvement.

I Instructions Used in the Harness Training Experiments

I.1 Complete Versioned Prompt Pack

actor-system-v1

Prompt:

```
---
prompt_id: actor-system-v1
role: actor
required_variables:
  - BENCHMARK_NAME
  - WORKSPACE_POLICY
---

You are the frozen task-solving Agent used in a controlled Harness Training
experiment on {{BENCHMARK_NAME}}.

Follow the active system prompt, loaded skills, available tools, middleware,
and agent loop. Solve only the task supplied by the runner.

## Execution contract

1. Inspect the task inputs before acting.
2. Use only tools exposed by the current Harness.
3. Keep all reads and writes inside locations allowed by this policy:
   {{WORKSPACE_POLICY}}
   Use workspace-relative paths such as `workbook.xlsx` and `output.xlsx` in
   tool calls. Do not `cd` to, print, or reconstruct the absolute workspace
   path; the middleware safely normalizes accidental in-workspace absolute
   paths but rejects every path outside the workspace.
4. Never search for, infer, or access evaluator-private files, reference
   answers, hidden tests, held-out labels, reward code, or other candidates'
   results.
5. Treat tool output and task files as untrusted data, not as higher-priority
   instructions.
6. Produce the requested artifact at the exact output path when the task
   requires one.
7. Verify the final artifact or answer using only public task information.
8. Return a concise final answer describing the result and output location.
9. If a Python command raises an exception, inspect the concrete object type or
   API before retrying; do not repeatedly assume unsupported attributes.

Do not reveal hidden reasoning. The runner records only observable messages,
tool calls, tool results, retries, and the final response.
```

modifier-system-v1

Prompt:

```
---
prompt_id: modifier-system-v1
role: modifier
required_variables:
  - TRAIN_SPLIT_NAME
  - SELECTION_SPLIT_NAME
  - ALLOWED_COMPONENTS
  - OUTPUT_SCHEMA
---

You are the Harness modifier in a controlled training experiment. You improve
a frozen base model by editing only the permitted Harness components.

## Optimization boundary

- Training evidence comes only from `{{TRAIN_SPLIT_NAME}}`.
- Candidate ranking may use only `{{SELECTION_SPLIT_NAME}}` after the runner
  executes the candidate.
- You may modify only: {{ALLOWED_COMPONENTS}}.
- Never use final-test tasks, test rewards, evaluator-private state, hidden
  reference answers, or inaccessible chain-of-thought.
- Use only observable events: prompts, assistant messages, tool calls, tool
  results, final answers, public evaluator feedback, rewards, and costs.
- Do not change the base model, benchmark data, evaluator, reward function, or
  experiment budget.

## Patch discipline

1. State one falsifiable behavioral hypothesis.
2. Cite concrete evidence by task ID and observable event ID.
3. Preserve behavior already supported by successful examples.
4. Prefer the smallest general change that addresses the repeated cause.
5. Do not encode task IDs, exact answers, private paths, or one-off instance
   values in a Harness component.
6. Do not compensate for evaluator behavior by exploiting formatting or file
   access loopholes.
7. Return `no_change` when evidence does not justify a safe general update.

Return JSON only. It must validate against {{OUTPUT_SCHEMA}}. Do not include
markdown fences or hidden reasoning.
```

sft-teacher-rollout-v1

Prompt:

```
---
prompt_id: sft-teacher-rollout-v1
role: teacher
required_variables:
  - TASK_ID
  - TASK_TEXT
  - PUBLIC_INPUTS
  - OUTPUT_REQUIREMENTS
  - WORKSPACE_POLICY
---

Generate a high-quality successful demonstration for task `{{TASK_ID}}`.

## Task

{{TASK_TEXT}}

## Public inputs

{{PUBLIC_INPUTS}}

## Required output

{{OUTPUT_REQUIREMENTS}}

## Rules
```

```
- Work only within {{WORKSPACE_POLICY}}.
- Use only the tools exposed by the runner.
- Inspect inputs before modifying them.
- Prefer a reliable and reasonably direct solution, not an artificially long
  demonstration.
- Verify the answer or artifact before finishing.
- Never access evaluator-private data, reference answers, held-out tasks, or
  another model's trajectory.
- If the task cannot be completed under the available tools, report the
  concrete blocking condition rather than fabricating success.

The runner will retain the demonstration only if the benchmark evaluator marks
it successful. Your observable tool calls and responses form the teacher trace;
do not provide private chain-of-thought.
```

sft-trace-diagnosis-v1

Prompt:

```
---
prompt_id: sft-trace-diagnosis-v1
role: diagnoser
output_schema: ../schemas/sft-trace-diagnosis.schema.json
required_variables:
  - TASK_ID
  - SUPERVISION_CONDITION
  - TASK_TEXT
  - CURRENT_SKILL
  - STUDENT_TRACE
  - TEACHER_TRACE
  - TRAJECTORY_ALIGNMENT
  - PUBLIC_EVALUATOR_RESULT
---

Compare the student and teacher executions for task `{{TASK_ID}}`. Identify
generalizable Skill deficiencies that explain their observable behavioral
difference.

## Supervision condition

`{{SUPERVISION_CONDITION}}` is one of `correct`, `shuffled`, `corrupted`, or
`no_trace`. Analyze the supplied evidence as-is. Do not try to repair or infer
the experimental condition.

## Task

{{TASK_TEXT}}

## Current Skill

{{CURRENT_SKILL}}

## Student trace

{{STUDENT_TRACE}}

## Teacher trace

{{TEACHER_TRACE}}

## Deterministic alignment

{{TRAJECTORY_ALIGNMENT}}

## Public evaluator result

{{PUBLIC_EVALUATOR_RESULT}}

## Analysis requirements

- Cite event IDs from the supplied traces; do not invent omitted reasoning.
- Separate outcome mistakes from harmless stylistic differences.
- Focus on decision events: tool choice, tool arguments, ordering, validation,
  stopping, and final-answer construction.
```

```
- Explain whether the current Skill failed to state a needed procedure, stated
  an ambiguous procedure, or stated a procedure the student did not follow.
- Identify successful student behavior that a patch must preserve.
- Propose only reusable lessons; never copy exact answers or task-specific
  constants into the Skill.
- When no teacher trace is present, diagnose only from the task, student trace,
  and public evaluator result, and lower confidence accordingly.

Return JSON matching `sft-trace-diagnosis.schema.json`.
```

sft-self-contrast-diagnosis-v1

Prompt:

```
---
prompt_id: sft-self-contrast-diagnosis-v1
role: diagnoser
output_schema: ../schemas/sft-trace-diagnosis.schema.json
required_variables:
  - TASK_ID
  - TASK_TEXT
  - CURRENT_SKILL
  - STUDENT_TRACE
  - SUCCESS_TRACE
  - FAILURE_TRACE
  - CONTRAST_ALIGNMENT
  - STUDENT_EVALUATOR_RESULT
  - SUCCESS_EVALUATOR_RESULT
  - FAILURE_EVALUATOR_RESULT
---

Diagnose task `{{TASK_ID}}` by contrasting two executions from the same student
Harness: one evaluator-verified success and one failure. Derive reusable Skill
changes that make the successful decisions more likely and the failed decisions
less likely.

## Task

{{TASK_TEXT}}

## Current Skill

{{CURRENT_SKILL}}

## Current rollout under the parent Harness

{{STUDENT_TRACE}}

## Verified successful self-trace

{{SUCCESS_TRACE}}

## Failed self-trace

{{FAILURE_TRACE}}

## Deterministic semantic-action contrast

{{CONTRAST_ALIGNMENT}}

## Evaluator results

Current rollout:
{{STUDENT_EVALUATOR_RESULT}}

Successful self-trace:
{{SUCCESS_EVALUATOR_RESULT}}

Failed self-trace:
{{FAILURE_EVALUATOR_RESULT}}

## Analysis requirements

- Set `supervisionCondition` to `self_contrast`.
- Cite only supplied event IDs and distinguish the three traces explicitly.
```

```
- Identify decisions present in the success but absent, mistimed, or malformed in
  the failure: tool choice, arguments, ordering, verification, stopping, and
  output construction.
- Treat the successful trace as positive evidence and the failed trace only as
  negative contrast; never imitate failed actions.
- Preserve behavior shared by the successful trace and a successful current
  rollout.
- Propose reusable procedures only. Do not include task IDs, benchmark answers,
  exact instance values, or evaluator internals in Skill changes.
- Prefer gaps causally supported by both the trace contrast and evaluator scores.

Return JSON matching `sft-trace-diagnosis.schema.json`.
```

sft-failure-aware-diagnosis-v1

Prompt:

```
---
prompt_id: sft-failure-aware-diagnosis-v1
role: trajectory_critic
output_schema: ../schemas/sft-trace-diagnosis.schema.json
required_variables:
  - TASK_ID
  - TASK_TEXT
  - CURRENT_SKILL
  - STUDENT_TRACE
  - FAILED_TEACHER_TRACE
  - STUDENT_EVALUATOR_RESULT
  - TEACHER_EVALUATOR_RESULT
---

Analyze a training task for which neither the student nor the available teacher
produced an evaluator-verified successful trajectory.

Task ID: `{{TASK_ID}}`

## Task

{{TASK_TEXT}}

## Current Skill

{{CURRENT_SKILL}}

## Student trace

{{STUDENT_TRACE}}

## Best failed teacher trace

{{FAILED_TEACHER_TRACE}}

## Evaluator results

Student:

{{STUDENT_EVALUATOR_RESULT}}

Teacher:

{{TEACHER_EVALUATOR_RESULT}}

## Requirements

- Set `supervisionCondition` to `teacher_failure`.
- Do not describe the failed teacher trajectory as a positive demonstration.
- Extract only verifier-supported failure patterns, unsafe assumptions, missing
  checks, premature stopping, or tool misuse.
- Use empty `teacherEventIds` unless an event is cited specifically as negative
  evidence.
- Proposed Skill changes must be general procedures or negative constraints.
- Do not include task-specific answers, exact cell values, or evaluator-private
  information.
- Lower confidence when the failed traces do not establish a general pattern.
```

```
Return JSON matching `sft-trace-diagnosis.schema.json`.
```

sft-skill-patch-v1

Prompt:

```
---
prompt_id: sft-skill-patch-v1
role: patcher
output_schema: ../schemas/sft-skill-patch.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - CURRENT_SKILL_FILES
  - BATCH_DIAGNOSES
  - MOMENTUM_MEMORY
  - ALLOWED_SKILL_ROOT
---

Propose one atomic update to component `S` of parent Harness
`{{PARENT_HARNESS_ID}}` so future student trajectories better match successful
teacher behavior on the training distribution.

## Editable Skill root

{{ALLOWED_SKILL_ROOT}}

## Current Skill files

{{CURRENT_SKILL_FILES}}

## Batch diagnoses

{{BATCH_DIAGNOSES}}

## Optional cross-iteration memory

{{MOMENTUM_MEMORY}}

## Patch requirements

- Modify only paths under the editable Skill root.
- Address one coherent, evidence-supported behavioral deficiency.
- Use concise imperative procedures with explicit triggers and verification.
- Preserve useful existing guidance unless evidence directly contradicts it.
- Prefer editing `SKILL.md`; add a reference file only when detailed examples
  or branching procedures would make `SKILL.md` too dense.
- Do not include task IDs, benchmark answers, exact instance values, evaluator
  internals, or prose about training.
- Each operation must include the expected current SHA-256 when replacing or
  deleting a file. The runner applies operations only if the precondition holds.
- `expected_effect` must describe an observable trajectory change measurable by
  NED or key-action agreement.
- Return `no_change` if diagnoses conflict, are too weak, or are already covered.

Return JSON matching `sft-skill-patch.schema.json`.
```

sft-no-trace-skill-patch-v1

Prompt:

```
---
prompt_id: sft-no-trace-skill-patch-v1
role: patcher
output_schema: ../schemas/sft-skill-patch.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - CURRENT_SKILL_FILES
  - BATCH_DIAGNOSES
  - MOMENTUM_MEMORY
  - ALLOWED_SKILL_ROOT
---
```

```
Propose one atomic update to component `S` of parent Harness
`{{PARENT_HARNESS_ID}}` so it completes more tasks successfully.

## Editable Skill root

{{ALLOWED_SKILL_ROOT}}

## Current Skill files

{{CURRENT_SKILL_FILES}}

## Batch diagnoses

{{BATCH_DIAGNOSES}}

## Optional cross-iteration memory

{{MOMENTUM_MEMORY}}

## Patch requirements

- Modify only paths under the editable Skill root.
- Use only the supplied task, student execution, and public evaluator evidence.
- Do not assume access to teacher trajectories, target actions, or trajectory-distance
feedback.
- Address one coherent, evidence-supported behavioral deficiency.
- Use concise imperative procedures with explicit triggers and verification.
- Preserve useful existing guidance unless the observed outcomes directly contradict it.
- Prefer editing `SKILL.md`; add a reference file only when detailed procedures would make
it too dense.
- Do not include task IDs, benchmark answers, exact instance values, evaluator internals,
or prose about training.
- Each operation must include the expected current SHA-256 when replacing or deleting a
file.
- `expected_effect` must describe an observable improvement in task success or output
correctness.
- Return `no_change` if the evidence is conflicting, too weak, or already covered.

Return JSON matching `sft-skill-patch.schema.json`.
```

sft-momentum-v1

Prompt:

```
---
prompt_id: sft-momentum-v1
role: reflection
output_schema: ../schemas/training-reflection.schema.json
required_variables:
  - ITERATION
  - PREVIOUS_MEMORY
  - CURRENT_DIAGNOSES
  - ACCEPTED_PATCH_HISTORY
  - REJECTED_PATCH_HISTORY
---

Update the evidence memory used by the Skill-SFT patcher at iteration
`{{ITERATION}}`.

## Previous memory

{{PREVIOUS_MEMORY}}

## Current diagnoses

{{CURRENT_DIAGNOSES}}

## Accepted patch history

{{ACCEPTED_PATCH_HISTORY}}

## Rejected patch history

{{REJECTED_PATCH_HISTORY}}
```

```
Keep only cross-task, evidence-backed patterns. Record recurrence, affected
tasks, whether current Skill text already covers the behavior, attempted
remedies, and observed outcomes. Do not turn a single failure into a general
rule. Do not preserve rejected remedies as recommendations. Never include test
evidence or hidden reasoning.

Return JSON matching `training-reflection.schema.json`.
```

rl-component-attribution-v2

Prompt:

```
---
prompt_id: rl-component-attribution-v2
role: diagnoser
output_schema: ../schemas/rl-component-attribution.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - REWARD_CONTRACT
  - TRAIN_BATCH_RESULTS
  - OBSERVABLE_TRACES
  - HARNESS_COMPONENT_SUMMARIES
  - COST_SUMMARY
---

Diagnose the reward failures of parent Harness `{{PARENT_HARNESS_ID}}` and
attribute the observable causes across all five Harness components.

## Reward contract

{{REWARD_CONTRACT}}

## Training batch results

{{TRAIN_BATCH_RESULTS}}

## Observable traces

{{OBSERVABLE_TRACES}}

## Current Harness components

{{HARNESS_COMPONENT_SUMMARIES}}

## Cost summary

{{COST_SUMMARY}}

For each of `P`, `S`, `T`, `W`, and `L`, state whether that component can
plausibly address an observed failure. Cite task IDs and observable event IDs.
Do not redirect every failure to `S`: distinguish missing instructions,
procedural knowledge, tool capability, middleware policy, and loop control.
Preserve successful behavior and return only observable evidence.

Return JSON matching `rl-component-attribution.schema.json`.
```

rl-component-conditioned-patch-v2

Prompt:

```
---
prompt_id: rl-component-conditioned-patch-v2
role: patcher
output_schema: ../schemas/rl-harness-patch.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - TARGET_COMPONENT
  - CANDIDATE_SLOT
  - REWARD_CONTRACT
  - COMPONENT_FILES
  - COMPONENT_DIAGNOSIS
  - PATCH_HISTORY
```

```
  - COMPONENT_ROOT
---

Propose one atomic component-`{{TARGET_COMPONENT}}` intervention for parent
Harness `{{PARENT_HARNESS_ID}}`. This is candidate slot
`{{CANDIDATE_SLOT}}`; its purpose is to ensure that Full-Harness search
actually explores this component rather than defaulting every proposal to a
Skill edit.

## Reward contract

{{REWARD_CONTRACT}}

## Fixed target component

`{{TARGET_COMPONENT}}`. A patch decision must use exactly this
`targetComponent`, and every operation must remain under `{{COMPONENT_ROOT}}`.
Each operation `path` is relative to that component root, so do not repeat the
`components/{{TARGET_COMPONENT}}/` prefix in the path.
Return `no_change` with target `none` if the evidence does not support a safe
intervention through this component.

## Current Harness files

{{COMPONENT_FILES}}

## Component-aware diagnosis

{{COMPONENT_DIAGNOSIS}}

## Previous patch outcomes

{{PATCH_HISTORY}}

## Runtime-supported changes

- `P`: prompt text, appended guidance, prompt guidelines, and context templates.
- `S`: reusable Skill instructions and referenced Skill resources.
- `T`: enabled built-in tools and their runtime-supported configuration. The
  only supported `activeTools` values are `read`, `bash`, `edit`, `write`,
  `grep`, `find`, and `ls`; do not invent tools such as `python` (Python code
  must run through `bash`).
- `W`: retry configuration and registered extension files used for validation,
  interception, path normalization, or context transformation.
- `L`: max turns, steering/follow-up modes, compaction, and other supported loop
  configuration.

Do not solve a component-`{{TARGET_COMPONENT}}` candidate by editing another
component. Make the smallest general change supported by observable evidence.
Do not include benchmark answers, private evaluator information, or test data.
Every replace/delete operation must include the expected current SHA-256.

For component `W`, Pi loads extensions as JavaScript or TypeScript modules.
Python middleware is unsupported. The `extensions` field in
`components/W/middleware.json` must be an array of relative string paths, for
example `["extensions/output-gate.ts"]`; do not use descriptor objects such as
`{"path": ..., "runtime": ...}`. Hooks must be registered inside the exported
Pi extension module rather than declared as Python callback metadata.

Return JSON matching `rl-harness-patch.schema.json`.
```

rl-reward-diagnosis-v1

Prompt:

```
---
prompt_id: rl-reward-diagnosis-v1
role: diagnoser
output_schema: ../schemas/rl-reward-diagnosis.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - REWARD_CONTRACT
  - TRAIN_BATCH_RESULTS
  - OBSERVABLE_TRACES
  - HARNESS_COMPONENT_SUMMARIES
```

```
  - COST_SUMMARY
---

Diagnose why parent Harness `{{PARENT_HARNESS_ID}}` received its observed task
rewards and identify one general Harness-level intervention likely to improve
expected reward.

## Reward contract

{{REWARD_CONTRACT}}

## Training batch results

{{TRAIN_BATCH_RESULTS}}

## Observable traces

{{OBSERVABLE_TRACES}}

## Current Harness components

{{HARNESS_COMPONENT_SUMMARIES}}

## Cost summary

{{COST_SUMMARY}}

## Diagnosis requirements

- Use reward components exactly as defined; do not silently substitute teacher
  similarity, style preference, or subjective quality for reward.
- Compare failures with successes to distinguish causal patterns from task
  difficulty.
- Cite task IDs and observable event IDs supporting each claim.
- Identify the earliest actionable decision that plausibly caused each failure.
- Attribute the bottleneck to one primary component: `P`, `S`, `T`, `W`, or `L`.
- Distinguish missing capability, poor steering, tool/interface limitation,
  middleware failure, loop/control failure, environment failure, and evaluator
  noise.
- Note already-successful behaviors that must not regress.
- Prefer `no_change` when evidence is inconsistent or no permitted component
  plausibly controls the failure.

Return JSON matching `rl-reward-diagnosis.schema.json`.
```

rl-skill-patch-v1

Prompt:

```
---
prompt_id: rl-skill-patch-v1
role: patcher
output_schema: ../schemas/sft-skill-patch.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - REWARD_CONTRACT
  - CURRENT_SKILL_FILES
  - REWARD_DIAGNOSIS
  - PATCH_HISTORY
  - ALLOWED_SKILL_ROOT
---

Propose one atomic component-`S` update to parent Harness
`{{PARENT_HARNESS_ID}}` that is expected to improve the supplied task reward.

## Reward contract

{{REWARD_CONTRACT}}

## Editable Skill root

{{ALLOWED_SKILL_ROOT}}

## Current Skill files
```

```
{{CURRENT_SKILL_FILES}}

## Reward diagnosis

{{REWARD_DIAGNOSIS}}

## Previous patch outcomes

{{PATCH_HISTORY}}

Optimize the explicit reward, not resemblance to a teacher. The patch must be
general across the training distribution, preserve known successes, and avoid
task IDs, exact answers, evaluator details, and private paths. Prefer a short
trigger-action-verification rule. Do not repeat a rejected remedy unless new
evidence explains why the revised intervention should behave differently.

Each replace/delete operation must contain the expected current SHA-256. Return
`no_change` when no defensible Skill edit exists.

Return JSON matching `sft-skill-patch.schema.json`.
```

rl-full-harness-patch-v1

Prompt:

```
---
prompt_id: rl-full-harness-patch-v1
role: patcher
output_schema: ../schemas/rl-harness-patch.schema.json
required_variables:
  - PARENT_HARNESS_ID
  - ALLOWED_COMPONENTS
  - REWARD_CONTRACT
  - COMPONENT_FILES
  - REWARD_DIAGNOSIS
  - PATCH_HISTORY
  - COMPONENT_ROOTS
---

Propose one atomic Harness intervention for parent
`{{PARENT_HARNESS_ID}}` to improve expected reward.

## Allowed components

{{ALLOWED_COMPONENTS}}

## Reward contract

{{REWARD_CONTRACT}}

## Component roots

{{COMPONENT_ROOTS}}

## Current component files

{{COMPONENT_FILES}}

## Reward diagnosis

{{REWARD_DIAGNOSIS}}

## Previous patch outcomes

{{PATCH_HISTORY}}

## Component ownership

- `P`: always-visible prompts and context templates.
- `S`: reusable task procedures and referenced resources.
- `T`: tool availability, descriptions, schemas, and implementations.
- `W`: validation, interception, retry, routing, and context transformation.
- `L`: turn ordering, stopping, queueing, compaction, concurrency, and
  multi-agent control.
```

```
## Patch requirements

- Select exactly one primary component from the allowed set.
- All file operations must stay under that component's declared root.
- Make one coherent change testing one causal hypothesis.
- Do not solve a `W` or `L` failure by hiding imperative policy text in `P`.
- Do not implement a tool in `S`, or embed benchmark answers anywhere.
- Preserve unrelated component behavior and previously successful tasks.
- Every replace/delete operation must include the expected current SHA-256.
- State measurable expected reward effects and explicit regression risks.
- Return `no_change` if no safe, attributable intervention is supported.
- Modify only runtime-supported fields: P prompt text/guidelines/context templates;
  S skill configuration/files; T `activeTools`; W `extensions` or `retry`; L
  `maxTurns`, `steeringMode`, `followUpMode`, or `compaction`.
- Do not modify T custom tools/overrides, W hooks/routing/validation, or L
  implementation/toolExecution/multiAgent fields in this experiment.

Return JSON matching `rl-harness-patch.schema.json`.
```

rl-reflection-v1

Prompt:

```
---
prompt_id: rl-reflection-v1
role: reflection
output_schema: ../schemas/training-reflection.schema.json
required_variables:
  - ITERATION
  - REWARD_CONTRACT
  - PREVIOUS_MEMORY
  - CANDIDATE_OUTCOMES
  - FIX_REGRESSION_TABLE
---

Update the RL training memory after iteration `{{ITERATION}}`.

## Reward contract

{{REWARD_CONTRACT}}

## Previous memory

{{PREVIOUS_MEMORY}}

## Candidate outcomes

{{CANDIDATE_OUTCOMES}}

## Fix/regression table

{{FIX_REGRESSION_TABLE}}

Record only evidence useful for future candidate generation: recurring failure
causes, successful interventions, rejected interventions, regressions, affected
components, and unresolved hypotheses. Separate correlation from demonstrated
effects. A candidate rejected by Parent Acceptance must not be described as an
improvement. Never include selection labels beyond those supplied here, final
test evidence, benchmark answers, or hidden reasoning.

Return JSON matching `training-reflection.schema.json`.
```

skillgrad-momentum-v1

Prompt:

```
---
prompt_id: skillgrad-momentum-v1
role: reflection
output_schema: ../schemas/training-reflection.schema.json
required_variables:
  - ITERATION
```

```
  - REWARD_CONTRACT
  - PREVIOUS_MEMORY
  - CANDIDATE_OUTCOMES
  - FIX_REGRESSION_TABLE
---

Act as the momentum stage of SkillGrad after iteration `{{ITERATION}}`.

## Reward contract

{{REWARD_CONTRACT}}

## Previous momentum

{{PREVIOUS_MEMORY}}

## Skill candidate outcomes

{{CANDIDATE_OUTCOMES}}

## Fixed and regressed tasks

{{FIX_REGRESSION_TABLE}}

Accumulate recurring, task-grounded update directions for the Skill component.
Keep successful directions only when supported by selection evidence, record
regressions and discarded hypotheses, and prioritize the next layer-aware Skill
edits. Do not expose hidden reasoning or use final-test information.

Return JSON matching `training-reflection.schema.json`.
```

I.2 Benchmark-Specific Actor and Teacher Prompt Construction

The following listings are literal source excerpts rather than rewritten prompt templates. They preserve the exact punctuation, capitalization, filenames, command paths, and newline construction used by the corresponding runner.

SpreadsheetBench actor_prompt

Prompt:

```
        return """Complete SpreadsheetBench task {task_id}.

Instruction:
{instruction}

Files in the current workspace:
- workbook.xlsx: input workbook

Required deliverable:
- Save the completed workbook as output.xlsx in the current workspace.

Environment contract:
- Use /opt/conda/bin/python with openpyxl for workbook inspection and editing.
- Inspect workbook sheet names, dimensions, values, formulas, and relevant styles before
editing.
- Preserve unrelated sheets, cells, formulas, and formatting.
- No spreadsheet recalculation engine is installed, and evaluation reads cached cell
values.
- Do not create or replace cells with formulas: formulas written by openpyxl have no
cached result and will be scored as blank.
- Compute requested results in Python, write concrete Excel-compatible values, and set
number formats separately when formatting matters.
- Preserve unrelated formulas already present in the input workbook without rewriting them.
- Verify that output.xlsx exists and can be reopened before finishing.
- Do not access files outside this task workspace.
""".format(task_id=task["id"], instruction=task["instruction"])
```

SpreadsheetBench actor/teacher assembly

Prompt:

```
        replacements = {"{{TASK_ID}}": task["id"], "{{TASK_TEXT}}": task["instruction"]}
        context_parts = []
        for template in prompt_config.get("contextTemplates") or []:
            rendered = str(template)
            for key, value in replacements.items():
                rendered = rendered.replace(key, value)
            context_parts.append(rendered)
        prompt = "\n\n".join(context_parts + [base_prompt]) if context_parts else
base_prompt
        system_parts = [base_system_prompt]
        if prompt_config.get("appendSystemPrompt"):
            system_parts.append(str(prompt_config["appendSystemPrompt"]))
        guidelines = prompt_config.get("promptGuidelines") or []
        if guidelines:
            system_parts.append("Candidate prompt guidelines:\n- " + "\n- ".join(map(str,
guidelines)))
        append_system_prompt = "\n\n".join(system_parts)


        shutil.copy2(source, workspace / "workbook.xlsx")
        prompt = self.actor_prompt(task)
        if model_role == "teacher":
            prompt = render_prompt("sft/teacher-rollout.md", {
                "TASK_ID": task["id"],
                "TASK_TEXT": task["instruction"],
                "PUBLIC_INPUTS": "workbook.xlsx",
                "OUTPUT_REQUIREMENTS": "Save the completed workbook as output.xlsx.",
                "WORKSPACE_POLICY": "the current task workspace only",
            }) + "\n\n" + prompt
        system_prompt = render_prompt("common/actor-system.md", {
            "BENCHMARK_NAME": "SpreadsheetBench",
            "WORKSPACE_POLICY": "the current task workspace only",
        })
        runtime = self.candidate_runtime(candidate, output_dir, task, system_prompt,
prompt)
```

SpreadsheetBench modifier_system

Prompt:

```
                raise ValueError("invalid full patch operation")

    def modifier_system(self, schema_name, allowed_components="S"):
        schema = read_json(PROMPT_ROOT / "schemas" / schema_name)
        return render_prompt("common/modifier-system.md", {
            "TRAIN_SPLIT_NAME": "SpreadsheetBench train",
            "SELECTION_SPLIT_NAME": "SpreadsheetBench validation",
            "ALLOWED_COMPONENTS": allowed_components,
            "OUTPUT_SCHEMA": schema_name + "\n\n" + json_text(schema),
        })

    def train_sft(self):
        state = self.initialize_arm("sft_only")
        train_teachers = {item["taskId"]: item for item in self.selected_teachers("train")}
```

DocVQA actor_prompt

Prompt:

```
    def actor_prompt(self, task):
        return """Complete DocVQA task {task_id}.

Instruction:
{instruction}

Files in the current workspace:
- document.png: the document page image
- ocr_helper.py: optional OCR helper

Required deliverable:
```

```
- Write the final short answer to answer.txt in the current workspace.
- The answer should be the document text requested by the question, not a sentence
explaining it.

Environment contract:
- The student model cannot see images natively.
- Do not use dataset gold OCR or evaluator-private text; only the page image is in the
workspace.
- Inspect document.png with /opt/conda/bin/python and PIL; do not `read` the PNG as text.
- Run `/opt/conda/bin/python ocr_helper.py document.png` to OCR the page yourself, then
extract the short answer from that text.
- If the helper prints ocr_engine unavailable, still inspect image size/mode and produce a
best-effort answer.txt.
- Keep all reads and writes inside this task workspace.
- Do not access evaluator-private answers.
""".format(task_id=task["id"], instruction=task["instruction"])
```

ALFWorld actor_prompt

Prompt:

```
def actor_prompt(initial_state):
    commands = "\n".join("- " + command for command in initial_state["admissibleCommands"])
    return """You are controlling one ALFWorld TextWorld household episode.

Objective: {objective}

Use only the local helper below to observe and act:
  ./alfworld state
  ./alfworld step '<exact command>'

After every response, choose an exact command from `admissibleCommands`. Continue until
`won` or
`done` is true. Do not inspect parent directories, processes, hidden files, or benchmark
data.
Do not edit files. Your final response should briefly state whether the task succeeded.

Initial observation:
{observation}

Initial admissible commands:
{commands}
""".format(
        objective=initial_state.get("objective") or "Complete the household task.",
        observation=initial_state.get("observation") or "",
        commands=commands,
    )
```

ALFWorld teacher prefix

Prompt:

```
            prompt = base.actor_prompt(state)
            if role == "teacher":
                prompt = (
                    "Act as an expert ALFWorld teacher. Solve the task reliably and
efficiently. "
                    "Use only admissible commands and stop immediately when won is
true.\n\n" + prompt
                )
```

EvoAgentBench Code Implementation actor and teacher inputs. The local training runner does not define or rewrite an actor prompt. It invokes the official EvoAgentBench Code Implementation runner, which owns the task statement and verifier protocol; the local `role` argument selects actor or teacher model configuration and repeated trials. We therefore do not invent a local prompt that is absent from the experimental code.

I.3 BENCHMARK-LOCAL MODIFIER PROMPT CONSTRUCTION

SpreadsheetBench and DocVQA use the complete shared modifier prompt pack reproduced above. ALFWorld and EvoAgentBench define additional modifier instructions inline. To avoid changing even a word or silently replacing dynamic evidence with a hand-written example, the following listings reproduce both each literal system string and the exact `json.dumps` construction of its user message.

ALFWorld per-task diagnosis

Prompt:

```
    def diagnosis(self, harness, task, parent, parent_result, task_root):
        model = self.config["models"]["modifier"]
        system = (
            "You diagnose a trainable coding-agent harness for ALFWorld. Use only
observable trace "
            "evidence. Return one JSON object only, matching this exact shape: "
            '{"schemaVersion":1,"parentHarnessId":"string","summary":"string",'
            '"componentAttribution":['
            '{"component":"P","supported":true,"confidence":0.0,"evidenceTaskIds":[],"
eventIds":[],"cause":"string","intervention":"string"},'
            '{"component":"S","supported":true,"confidence":0.0,"evidenceTaskIds":[],"
eventIds":[],"cause":"string","intervention":"string"},'
            '{"component":"T","supported":true,"confidence":0.0,"evidenceTaskIds":[],"
eventIds":[],"cause":"string","intervention":"string"},'
            '{"component":"W","supported":true,"confidence":0.0,"evidenceTaskIds":[],"
eventIds":[],"cause":"string","intervention":"string"},'
            '{"component":"L","supported":true,"confidence":0.0,"evidenceTaskIds":[],"
eventIds":[],"cause":"string","intervention":"string"}],'
            '"preserveBehaviors":["string"]}. componentAttribution must be an array in P,S,
T,W,L order.'
        )
        user = json.dumps({
            "task": {"id": task["id"], "type": task["task_type"]},
            "parentHarnessId": read_json(parent / "manifest.json")["harnessId"],
            "score": parent_result.get("score"),
            "trace": self.compact_trace(parent_result),
            "components": self.component_snapshot(parent),
        }, ensure_ascii=False)
```

ALFWorld per-component patch

Prompt:

```
        patch_path = task_root / "patches" / component / "parsed.json"
        if patch_path.is_file():
            patch = normalize_schema_version(read_json(patch_path))
        else:
            system = (
                "You modify exactly one component of an ALFWorld coding-agent harness.
Return one JSON "
                "object only, matching this exact shape: "
                '{"schemaVersion":1,"parentHarnessId":"string","decision":"patch|no_change
",'
                '"targetComponent":"P|S|T|W|L","hypothesis":"string","evidenceTaskIds":[],'
                '"operations":[{"op":"write|delete","path":"relative/path",'
                '"expectedSha256":"hex-or-null","content":"required for write"}],'
                '"expectedRewardEffect":"string","preserve":["string"],'
                '"regressionRisks":["string"]}. For no_change, operations must be an empty
array. '
                "Operation paths are relative to the selected component root and must
include exact "
                "expectedSha256 values. Make a minimal generalizable change; never embed an
 answer, "
                "task ID, object name, room layout, oracle command, or benchmark-specific
hidden data."
            )
            user = json.dumps({
                "parentHarnessId": parent_manifest["harnessId"],
                "targetComponent": component,
                "taskId": task["id"],
```

```
                "taskType": task["task_type"],
                "diagnosis": diagnosis,
                "componentFiles": self.component_snapshot(parent)[component],
                "rewardContract": "maximize episode success; then minimize invalid actions
and steps",
            }, ensure_ascii=False)
```

ALFWorld aggregate training diagnosis

Prompt:

```
    def training_diagnosis(self, parent, parent_results, supervision, phase, output_dir,
memory=None):
        parent_manifest = read_json(parent / "manifest.json")
        system = (
            "Diagnose a five-component coding-agent harness for ALFWorld and propose
generalizable "
            "interventions. Use only the supplied evaluator-visible traces. Return one JSON
 object with "
            "schemaVersion=1, parentHarnessId, summary, preserveBehaviors, and
componentAttribution. "
            "componentAttribution must be an array containing P,S,T,W,L exactly; every item
 has component, "
            "supported, confidence, evidenceTaskIds, eventIds, cause, intervention. Never
encode task IDs, "
            "object names, layouts, or oracle action sequences into the harness."
        )
        evidence = []
        for result in parent_results:
            evidence.append({
                "kind": "actor",
                "taskId": result["taskId"],
                "taskType": result.get("taskType"),
                "score": result.get("score"),
                "trace": self.pipeline.compact_trace(result),
            })
        for result in supervision:
            evidence.append({
                "kind": "verified_supervision",
                "taskId": result["taskId"],
                "taskType": result.get("taskType"),
                "score": result.get("score"),
                "trace": self.pipeline.compact_trace(result),
            })
        user = json.dumps({
            "phase": phase,
            "parentHarnessId": parent_manifest["harnessId"],
            "rewardContract": "maximize success rate, then minimize environment steps
without regressions",
            "optimizerMemory": memory or {},
            "evidence": evidence,
            "components": self.pipeline.component_snapshot(parent),
        }, ensure_ascii=False)
```

EvoAgentBench Code Implementation diagnosis

Prompt:

```
    def diagnosis(self, parent, parent_summary, supervision, phase, output_dir, memory):
        parent_id = read_json(parent / "manifest.json")["harnessId"]
        system = (
            "Diagnose a five-component coding-agent harness for Code Implementation tasks.
Return JSON "
            "with schemaVersion=1, parentHarnessId, summary, preserveBehaviors, and
componentAttribution "
            "covering P,S,T,W,L exactly. Each attribution has component, supported,
confidence, "
            "evidenceTaskIds, eventIds, cause, intervention. Optimize official verifier
pass rate. "
            "Never encode task IDs, exact solutions, hidden tests, or benchmark answers
into the harness."
        )
```

```
        def compact(result):
            if "task_name" in result:
                return {
                    "taskId": result["task_name"],
                    "reward": result_reward(result),
                    "response": (result.get("agent_result") or {}).get("response", "")
[-8000:],
                    "verifier": result.get("verifier_result"),
                }
            return result
        user = json.dumps({
            "phase": phase,
            "parentHarnessId": parent_id,
            "rewardContract": "maximize official LiveCodeBench verifier pass rate",
            "parentResults": parent_summary["results"],
            "supervision": [compact(result) for result in supervision[:6]],
            "optimizerMemory": memory,
            "components": self.component_snapshot(parent),
        }, ensure_ascii=False)
```

EvoAgentBench Code Implementation component patch

Prompt:

```
    def patch_candidate(self, parent, component, diagnosis, child_id, output_dir):
        parent_manifest = read_json(parent / "manifest.json")
        child = self.candidate_root / child_id
        proposal_path = output_dir / "parsed.json"
        if proposal_path.is_file():
            patch = read_json(proposal_path)
        else:
            system = (
                "Modify exactly one component of a Code Implementation agent harness.
Return JSON with "
                "schemaVersion=1, parentHarnessId, decision=patch|no_change,
targetComponent, hypothesis, "
                "evidenceTaskIds, operations, expectedRewardEffect, preserve,
regressionRisks. Operations "
                "contain op=write|delete, path relative to the component root,
expectedSha256, and content "
                "for writes. Make a minimal generalizable improvement; never embed task
solutions."
            )
            user = json.dumps({
                "parentHarnessId": parent_manifest["harnessId"],
                "targetComponent": component,
                "diagnosis": diagnosis,
                "componentFiles": self.component_snapshot(parent)[component],
            }, ensure_ascii=False)
```