INSCRIBE: INTERNALIZING SCAFFOLDS THROUGH AGENTIC REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning with verifiable rewards trains coding and terminal agents from expensive rollouts, yet a task on which every rollout fails yields no group-relative solver signal and is discarded. Such failures need not mark the model's limits: with the weights frozen, different harnesses—the programs that organize the agent's planning, tool use, and verification—recover a substantial share of them, and a short teacher diagnosis of the failed attempts recovers more of the remainder. Can these gains outlast the support that made them possible? We treat harnesses and hints as scaffolds: supports put up during training and taken down once the policy has learned what they made reachable. InScribe is a two-stage on-policy RL recipe for internalizing the benefits of both. In Stage I the policy writes its own harness before each task, so that choosing how to work becomes one of its actions; each harness is used once and discarded. In Stage II, training continues on the tasks that a Stage-I probe leaves unsolved under both the stock harness and sampled rewrites, keeping harness generation and adding a verifier-informed teacher diagnosis as context rather than as a prediction target. Main evaluations run under fixed harnesses, with no generated harness and no teacher hint. A 9B student trained this way gains 10.1 points on Terminal-Bench 2.0 and 11.0 on SWE-bench Verified over its base model; the gains hold under harnesses it never trained with and transfer to function-calling and interactive-agent benchmarks it never saw. What the scaffolds taught lives in the weights.

1 INTRODUCTION

Reinforcement learning with verifiable rewards has become the main route to stronger coding and terminal agents (DeepSeek-AI, 2025; Ivison et al., 2026; Pan et al., 2024; Jain et al., 2025). The policy attempts a task in an executable environment, a verifier scores the outcome, and group-relative methods such as GRPO (Shao et al., 2024) turn the variation in scores across a task's rollouts into a learning signal. Each rollout is costly—an environment, a long sequence of inference and tool execution—and returns a single scalar. When every rollout of a task returns zero, the advantages vanish and the whole group is dropped from the batch (Yu et al., 2025; Ivison et al., 2026). The environment, the trajectories, and the verifier runs are spent, and nothing is learned from them.

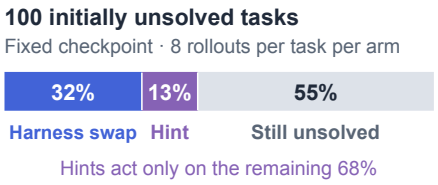


Figure 1: **Diagnosis on a frozen checkpoint.** Of 100 tasks with no success under the stock harness, the two alternative harnesses jointly recover 32; a short diagnosis then recovers 13 of the remaining 68 (Appendix A).

Zero reward is not zero information. The failed attempts record what the agent tried and where it went wrong, and existing work recovers such tasks by changing something outside the policy, along two lines. One changes how the agent works: a hand-written harness organizes planning, tool use, verification, and termination around the model (Merrill et al., 2026; Wang et al., 2025), and harness search rewrites that program around a frozen model (Hu et al., 2024; Che et al., 2026). The other changes what the agent knows: memories carry lessons across episodes (Ouyang et al., 2026; Zhang et al., 2026b), and hints guide it through problems it cannot

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.

yet solve (Xia et al., 2026). Both make the supported agent stronger. Improving that agent, however, is a different objective from improving the policy that remains when the support is withdrawn.

Our position is that harnesses and hints are scaffolds for learning, and what is learned should live in the weights. A scaffold is put up so that the policy can reach tasks it otherwise fails, and taken down once it has learned to reach them. Under this view the two external fixes become two kinds of training-time support—a way of working and a piece of information—and the question is no longer how much they help the agent that has them, but how much of that help the policy retains once they are withdrawn.

Before further training, we ask how much of the failure is really the model's. On 100 training tasks with zero verified successes in eight stock-harness attempts, we hold the weights fixed and vary first the harness and then the information in context (Figure 1; Appendix A). Running the same model under two hand-written harnesses, Terminus2 (Merrill et al., 2026) and OpenHands v2 (Wang et al., 2025), each with the same eight attempts, recovers 32 of the tasks between them; a short teacher diagnosis of the failed attempts, placed in context, recovers 13 of the 68 that remain, under the very harness that had failed them. The weights never moved, so nearly half of these failures were recoverable by a change of method or of information. Harness rewrites require no teacher call; each diagnosis does. We therefore first let the policy change how it works, then reserve teacher diagnoses for the tasks its probe leaves unsolved.

InScribe turns these observations into two stages of on-policy RL (Figure 2). In Stage I, harness evolution, the policy learns to rewrite its own harness specification, within a fixed runtime, before each task, so that choosing how to work becomes one of its actions; each rewritten harness is used once and discarded, while learning accumulates in the policy weights. Stage II continues from the Stage-I checkpoint on the tasks left unsolved by a probe under both the stock harness and sampled rewrites. Harness generation continues, while each task is presented with and without a fixed, verifier-informed teacher diagnosis, used as context rather than as a prediction target.

Our contributions are the recipe and three findings about it.

- InScribe, a two-stage on-policy RL recipe that uses harnesses and hints as training-time scaffolds; main evaluations use fixed harnesses with neither scaffold present (Section 3).
- **The gains live in the weights.** A 9B student gains 10.12 points on Terminal-Bench 2.0 and 11.00 on SWE-bench Verified over its base model; the improvement holds under three fixed harnesses, two of them never used in training, and transfers to function-calling and interactive-agent benchmarks (Sections 4.2 and 4.3).
- **Harness evolution is worth more than longer training.** Letting the policy write its harness beats stock-harness RL with fewer updates, and Stage I learns harnesses that are executable and, by mid-training, better than the default (Sections 4.4 and 4.5).
- **Diagnoses unlock the all-failure tasks.** Continuing RL on the residual tasks without diagnoses leaves the policy where Stage I left it; the same tasks with diagnoses raise it further, and this stage is most effective after harness evolution rather than from the base model or after stock-harness RL (Section 4.4).

2 Related Work

Extracting more from the same rollouts. One line keeps the agent fixed and sharpens what its rollouts teach: step-level credit through implicit step rewards (Liu et al., 2026b), tree search (Ji et al., 2026), group-in-group advantages (Feng et al., 2025b), or critical-action weighting (Shen et al., 2025); failure attribution (Zhang et al., 2026a); and curricula that order tasks from easy to hard (Parashar et al., 2026), track the edge of solvability (Wu et al., 2026), or synthesize experience (Chen et al., 2026). These methods improve credit assignment, task selection, or experience generation. With outcome-only group-relative updates, an all-failure group provides no reward contrast (Equation (1)) and is commonly filtered out (Yu et al., 2025). InScribe changes how the agent attempts the task, making successes appear where the stock harness yielded none. This moves part of exploration from individual tool actions to the execution strategy that organizes them. The resulting trajectories still receive ordinary verifier rewards.

Changing how the agent works. Hand-written harnesses fix planning, tool use, verification, and termination around the model (Yao et al., 2023; Yang et al., 2024; Wang et al., 2025; Merrill et al., 2026); agent-program search evolves prompts, workflows, or code around a frozen one (Hu et al., 2024; Zhou et al., 2024; Liu et al., 2025a; Che et al., 2026). HASE co-evolves model, harness, and solutions (Luo et al., 2026) and EvoHarness-RL learns runtime state management (Ning et al., 2026); in both, the learned harness ships with the deployed agent. SEAL is closer to our position—the model writes self-edits whose only lasting trace is a weight update (Zweiger et al., 2025)—but its edits are fine-tuning data and hyperparameters, not an execution harness inside an RL rollout. In InScribe the policy writes a task-conditioned harness and executes under it within one rollout; the harness tokens receive their own outcome credit, unlike retrieved passages or tool outputs that tool-using RL masks from the loss (Jin et al., 2025; Feng et al., 2025a), and the harness is discarded when the rollout ends.

Changing what the agent knows. Memory systems carry strategies across episodes through a reasoning bank (Ouyang et al., 2026), an evolving context (Zhang et al., 2026b), or a learned tool-graph memory (Feng et al., 2026), retained at deployment. EMPO^2 combines memory-guided exploration with on- and off-policy updates to retain gains without memory at test time (Liu et al., 2026c). Training-time guidance takes several forms: self-proposed interventions (Yang et al., 2026), augmented questions (Fang & Shi, 2026), minimal sufficient knowledge (Yu et al., 2026), the model's own shortened reasoning traces (Liu et al., 2026a), or a co-trained hinter (Xia et al., 2026). Nearest to Stage II, Agent-RLVR trains software-engineering agents on guided reattempts with SFT and DPO and removes the guidance at evaluation (Da et al., 2025). InScribe differs in three ways: the teacher is a fixed external model with access to the private verifier; training stays on-policy RL, with failures in the update; and hints are confined to the residual pool the Stage-I probe leaves unsolved, each paired with a clean copy of its task. The clean copies keep unassisted attempts in the training distribution as the teacher opens up new successful trajectories.

Agentic RL for software engineering. Verifiable executable tasks exist at scale for repositories (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025) and terminals (Pi et al., 2026); RL on them ranges from software-evolution rewards (Wei et al., 2025) to TMax (Ivison et al., 2026), whose training pool and default harness we adopt. SWE-bench and Terminal-Bench score success by execution under their own standard harnesses (Jimenez et al., 2024; Merrill et al., 2026). Because those harnesses are fixed, public, and different from the one we train with, a gain under them is one the training scaffolds cannot account for.

3 INSCRIBE

InScribe trains one policy, π_θ, through two stages of on-policy RL (Figure 2). Throughout, an attempt at a task is made by an *agent configuration* $a = (\pi_\theta, S, h)$: the policy, the harness S it runs under, and any guidance h placed in its context. Stage I (Section 3.2) lets the policy write S: each rollout rewrites a fixed default harness for its task, executes under the result, and is credited separately for the rewrite and for the solution. Stage II (Section 3.3) also lets a teacher write h for the tasks that still fail under the stock harness and the harnesses sampled in a probe. Both scaffolds are removed once training ends: in the main evaluations the configuration is $(\pi_\theta, S_{\text{eval}}, \emptyset)$, with S_{eval} a fixed benchmark harness, so whatever the two stages taught must be carried by π_θ alone.

3.1 SETUP AND NOTATION

Tasks $x \sim \mathcal{D}$ live in executable environments. An attempt under configuration a ends with a verifier verdict $Y(x; a, \xi) \in \{0, 1\}$, where ξ collects sampling and environment randomness. Episodes that fail for infrastructure reasons—a sandbox that does not start, a verifier that does not run—are excluded from every reward statistic below; an invalid attempt is not a zero.

The unit of training is a *cohort*: $G+1$ rollouts of one task under one policy version, sharing the verifier, environment, sampling temperature, and token budget. Slot 0 runs the stock harness S_0 and yields the anchor reward r_0; slots $1, \ldots, G$ each run a candidate configuration and yield rewards $r_1, \ldots, r_G$. Candidates use policy-written harnesses in both stages, with or without a teacher diagnosis in Stage II. The cohort shape and the credit rules of Section 3.2 remain unchanged.

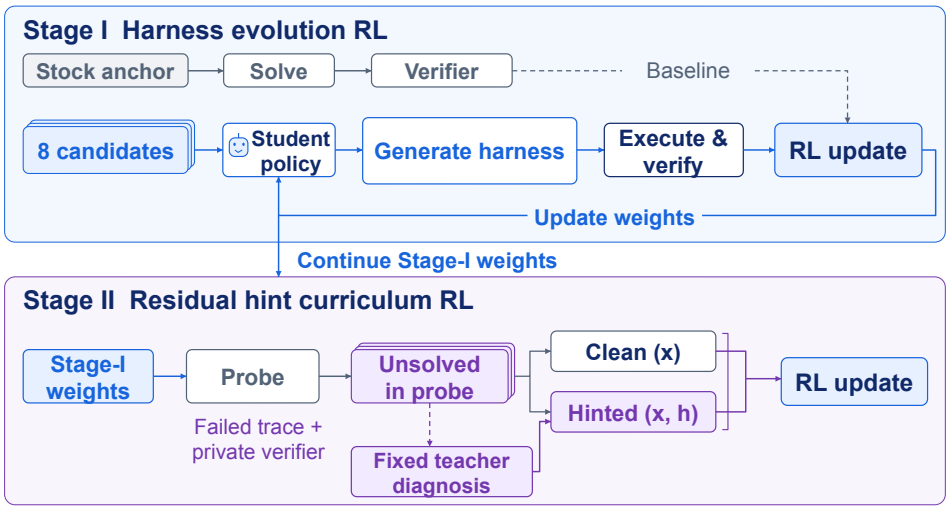


Figure 2: **Training mechanics.** Stage I uses one loss-masked stock anchor and eight candidates that adapt the fixed default for the current rollout. The RL update changes policy weights, not a persistent harness. Stage II uses the 2,500 residual tasks, each present as a clean record and as a record with a fixed teacher diagnosis, shuffled 1:1. The teacher produces each diagnosis from the student's failed traces and the training task's private verifier script.

Group-relative methods learn from disagreement within the cohort. If every valid reward is zero, the centered advantages are zero as well,

$$\sum_{k=1}^{G} r_k = 0 \quad \Longrightarrow \quad r_k - \frac{1}{G}\sum_{j=1}^{G} r_j = 0 \quad \text{for every } k, \tag{1}$$

and the cohort contributes no gradient toward solving the task. Finally, a task is *recovered* by a change of configuration if the new configuration yields at least one verified success in fresh attempts where the old one yielded none; the pilot of Section 1 and Appendix A uses this term.

3.2 Stage I: Harness Evolution

Three choices realize Stage I: the harness is a program the policy writes and then executes under; every cohort carries a stock anchor that did no rewriting; and writing and solving are credited separately.

The harness as a policy action. Each candidate starts from the stock harness S_0, the TMax Bash-agent control program (Ivison et al., 2026), which fixes six concerns: planning, tool use, context retention, verification, recovery, and termination. Before touching the environment, the policy receives the task, the stock specification, and its schema, and writes a task-conditioned replacement S_i as one JSON object covering the same concerns (Appendix C.1). It sees no reference solution, verifier output, or history from other episodes. The runtime validates S_i, hashes it, and loads it; it enforces the tool, loop, and context fields and renders the remaining fields into the solver's system prompt. The hash fixes the candidate's status: EXECUTED if a valid harness other than S_0 ran, CLONE if the policy returned S_0 exactly, FALLBACK if its object failed validation twice and S_0 ran instead. The harness is discarded when the rollout ends; what persists is the policy that wrote it.

The stock anchor. Slot 0 executes S_0 and is excluded from the loss: it is the baseline a rewrite is judged against, not a behavior being trained. We use $G = 8$ candidates, six directed rewrites and two free rewrites, with the direction the only difference in their prompts; directed slots spread the rewrites across the concerns, free slots let the policy change what it judges most relevant, and any slot may return S_0 when a change is not warranted.

Credit for writing and for solving. The solver channel uses the verifier reward r_i. The harness channel scores an executed rewrite against the stock anchor and penalizes invalid generation:

$$\delta_i = \mathbf{1}[\text{status}_i = \text{EXECUTED}]\,(r_i - r_0) \;-\; \lambda\,\mathbf{1}[\text{status}_i = \text{FALLBACK}]\,, \tag{2}$$

so a clone scores 0, a fallback $-\lambda$, and for $\lambda > 0$ an unparseable harness can never silently buy the default. Candidate i's tokens split into harness tokens $\mathcal{H}_i$ from the writing turn and solver tokens $\mathcal{E}_i$ from the solving turns ($\mathcal{H}_0 = \emptyset$). Each channel is centered within the G candidates without variance normalization (Liu et al., 2025b),

$$A_i^{\text{sol}} = r_i - \frac{1}{G}\sum_{j=1}^{G} r_j, \qquad A_i^{\text{har}} = \delta_i - \frac{1}{G}\sum_{j=1}^{G} \delta_j, \tag{3}$$

each token takes its channel's advantage, $A_{i,t} = \mathbf{1}[t \in \mathcal{E}_i]\,A_i^{\text{sol}} + \mathbf{1}[t \in \mathcal{H}_i]\,A_i^{\text{har}}$, and enters the PPO clipped surrogate (Schulman et al., 2017) $\ell_{i,t}(A) = \min\{\rho_{i,t}A,\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\,A\}$ with importance ratio $\rho_{i,t} = \pi_\theta(y_{i,t} \mid c_{i,t})/\pi_{\theta_{\text{old}}}(y_{i,t} \mid c_{i,t})$. Here $y_{i,t}$ is the sampled token and $c_{i,t}$ its autoregressive context. The objective normalizes each channel by its token count and weights the harness channel by β,

$$\mathcal{L}_{\text{I}}(\theta) = -\frac{1}{Z_{\text{sol}}}\sum_{i=1}^{G}\sum_{t\in\mathcal{E}_i} \ell_{i,t}\big(A_i^{\text{sol}}\big) - \frac{\beta}{Z_{\text{har}}}\sum_{i=1}^{G}\sum_{t\in\mathcal{H}_i} \ell_{i,t}\big(A_i^{\text{har}}\big)\,, \tag{4}$$

with $Z_{\text{sol}} = \sum_i |\mathcal{E}_i|$, $Z_{\text{har}} = \sum_i |\mathcal{H}_i|$, $\beta = 0.1$, and $\lambda = 0.05$ (Appendix B.2), so the balance between channels does not depend on harness length.

Where the split matters. When every candidate is EXECUTED, centering cancels r_0 and $A_i^{\text{har}} = A_i^{\text{sol}}$ (Appendix C.3); since almost every candidate executes as training progresses (Figure 3), the anchor and penalty act mainly while harnesses are forming and recede afterward. When every reward is zero, Equation (1) empties the solver channel and only status differences remain: such a cohort teaches the policy to write valid harnesses and nothing about solving the task. Stage II is built for these tasks.

3.3 Stage II: residual-only teacher hints

Stage II changes what the policy trains on, not how it trains. The cohort, the status labels, and the credit rules of Section 3.2 are used unchanged; what changes is the task set and the context. Three choices realize the stage: the tasks are only those Stage I still fails; the guidance is a fixed diagnosis from a teacher that can see the verifier; and every hinted task is paired with an unhinted copy of itself.

Selecting residual tasks. We freeze the end-of-Stage-I checkpoint θ_{I} and probe every training task with one Stage-I cohort—the stock anchor and eight harness-writing candidates, no hints—recording the anchor reward $r_0(x;\theta_{\text{I}})$ and candidate rewards $r_i(x;\theta_{\text{I}})$, $i = 1,\dots,G$. A task enters the residual pool only if all nine outcomes are infrastructure-valid and zero,

$$\mathcal{F}_1 = \{\, x \in \mathcal{D}_{\text{train}} : r_i(x;\theta_{\text{I}}) = 0 \text{ for all } i = 0,\dots,G \,\}. \tag{5}$$

These are the tasks that neither the default harness nor any sampled rewrite solves, so the probe found no solver contrast on them and, since nearly every candidate now executes, little harness contrast either (Section 3.2). The probe's failed attempts are kept as evidence for the teacher (Appendix D.1). Training resumes from $\theta_{\text{II},0} = \theta_{\text{I}}$ with a lower learning rate and every other setting unchanged (Appendix B.2), and each candidate again starts from S_0.

The diagnosis as context. For each $x \in \mathcal{F}_1$ the teacher π_{T}, DeepSeek-V4-Flash (DeepSeek-AI, 2026), receives the task instruction, a compressed record of the probe's failed attempts, and the private verifier script (Appendix D.3). It returns one short diagnosis h_x—a concept-level diagnosis, evidence the student overlooked, a next direction, and an observable self-check—without executing the task or seeing a reference solution. The student receives only the diagnosis; the teacher prompt prohibits verifier-only details, including test paths, test names, and expected values (Appendix D.4 gives two diagnoses verbatim). Each h_x is generated once, fixed for the stage, and placed in the student's context before it writes its harness, so it can shape the rewrite and the solution alike. It is loss-masked: the policy is never trained to reproduce it.

Pairing clean and hinted records. Every residual task contributes two records, a clean one $(x, \emptyset)$ and a hinted one (x, h_x), identical in task, image, verifier, tools, and limits; the records are shuffled in a 1:1 mixture with no other replay. Hinted records guide exploration; clean records train the same policy to solve the task without that guidance. Transfer runs through the shared weights: a success under h_x updates the same policy that the clean record of x then tests without h_x, and once the task is solved clean, its clean cohort stops being all-zero and contributes solver signal of its own. This is why the diagnosis can stay context rather than become a target—what the policy keeps is not the text of the hint but the behavior that earned reward with it. Each record forms its own cohort of one stock anchor and eight candidates, with advantages centered inside that cohort and never across the clean and hinted arms. Writing $a \in \{\text{clean}, \text{hinted}\}$ with $h_{\text{clean}} = \emptyset$ and $h_{\text{hinted}} = h_x$, the curriculum and objective are

$$\mu_{\text{II}} = \tfrac{1}{2}\,\mathcal{D}_{\mathcal{F}_1}^{\text{clean}} + \tfrac{1}{2}\,\mathcal{D}_{\mathcal{F}_1}^{\text{hinted}}, \qquad \mathcal{L}_{\text{II}}(\theta) = \mathbb{E}_{(x,h_a)\sim\mu_{\text{II}}}\left[\mathcal{L}_{\text{I}}(\theta; x, h_a)\right], \tag{6}$$

with Equations (2)–(4) applied within each cohort. Every verifier-scored candidate in an admitted cohort enters the update, failures included; only the anchor and the non-policy context are masked. Stage II is therefore continued on-policy RL rather than fine-tuning on filtered successes, and the teacher is called only on the frozen residual pool. Like the generated harness, the diagnosis is absent at evaluation.

4 Experiments

We test the three claims of Section 1 in turn: whether InScribe improves a 9B student's benchmark performance (Section 4.2), whether these gains transfer across execution harnesses without generated harnesses or hints (Section 4.3), and how harness evolution and diagnostic guidance contribute to the outcome (Section 4.4). Training diagnostics track executable harness generation and reward over time (Section 4.5).

4.1 Experimental setup

All runs share one student, one training pool, and one default harness; the stages and controls differ only in what the policy trains on. We train Qwen3.5-9B (Qwen Team, 2026) with TMax's default Bash harness on executable tasks from the TMax and R2E-Gym-Subset sources (Ivison et al., 2026; Jain et al., 2025). The Stage-I probe admits 2,500 of the 10,000 training tasks to the residual pool, so Stage II trains on 5,000 records. Table 1 summarizes the two stages; the training pool is deduplicated against the two primary evaluation benchmarks. Appendix B.4 describes the asynchronous system that makes nine-rollout cohorts practical.

Table 1: **Training setup.** Both stages train the same student; Stage II uses fixed, verifier-informed DeepSeek-V4-Flash diagnoses.

	Stage I	Stage II
Initialization	Base Qwen3.5-9B	End-of-Stage-I checkpoint
Tasks	10,000-task training pool	2,500 residual tasks from Stage I
Curriculum	Harness adaptation	Clean/hinted records, 1:1; no replay
Updates	200	100 additional

Our primary evaluations use Terminal-Bench 2.0 under Terminus2 (Merrill et al., 2026) and SWE-bench Verified under OpenHands (Jimenez et al., 2024; Wang et al., 2025). Broader evaluations cover terminal tasks, code generation, function calling, and interactive agents (Appendix B.1). Evaluation removes generated harnesses and hints. Scores are mean $\pm$ sample standard deviation over three evaluation runs; reported gaps are differences between displayed means.

Five control paths isolate continued training, harness adaptation, and teacher guidance; hint RL below means continued RL with the diagnoses in context. Stock-harness RL matches the full recipe's 300 updates; the no-hint continuation matches Stage II while replacing each diagnosis with a clean copy. Full control definitions and checkpoint-specific residual selection appear in Appendix B.2, with the training settings in Table 6 and the run footprint in Table 7.

4.2 MAIN RESULTS

Table 2 compares the base model, the released TMax-9B checkpoint (itself trained from Qwen3.5-9B with the TMax recipe), the Stage-I checkpoint, and InScribe across ten benchmarks, all under fixed evaluation harnesses without hints or generated harnesses. All InScribe entries come from the same Stage-II checkpoint. On the two main benchmarks InScribe reaches 31.09% on Terminal-Bench 2.0 and 60.67% on SWE-bench Verified, 10.12 and 11.00 points above the base model and 4.87 and 2.20 above Stage I.

Table 2: **Agent and code benchmark results.** Mean scores (%, higher is better) over three evaluation runs, reported as mean ± standard deviation. All models are 9B. TB2: Terminal-Bench 2.0; SWE-V: SWE-bench Verified; TBLite: OpenThoughts-TBLite; LCB: LiveCodeBench. Bold marks the largest mean per column. All InScribe entries use the same checkpoint.

(a) Terminal, software engineering, and code generation

Model / method	TB2	SWE-V	TBLite	LCB-v6
Qwen3.5-9B	20.97 ± 1.72	49.67 ± 1.72	43.53 ± 1.40	49.93 ± 0.46
TMax-9B	22.85 ± 1.30	54.07 ± 0.81	57.37 ± 1.44	50.66 ± 0.58
Stage I	26.22 ± 1.30	58.47 ± 1.70	59.92 ± 1.41	49.15 ± 0.17
InScribe (Stage II)	**31.09** ± 1.29	**60.67** ± 1.72	**62.61** ± 1.42	**54.29** ± 1.73

(b) Function calling and interactive agents

	BFCL			τ^2-Bench		
Model / method	v1	v2	v3	Airline	Retail	Telecom
Qwen3.5-9B	83.19 ± 0.69	79.55 ± 0.28	31.79 ± 2.15	46.00 ± 2.00	44.74 ± 2.32	81.87 ± 0.51
TMax-9B	81.92 ± 0.72	78.11 ± 0.71	51.34 ± 0.32	55.33 ± 3.05	57.02 ± 3.04	77.49 ± 1.83
Stage I	**83.50** ± 0.48	79.09 ± 0.49	47.96 ± 1.73	52.67 ± 3.05	53.51 ± 1.52	79.82 ± 1.52
InScribe (Stage II)	82.69 ± 0.48	**80.41** ± 0.32	**53.04** ± 1.40	**68.67** ± 3.05	**59.65** ± 3.51	**85.20** ± 1.63

Stage I lifts the training distribution; Stage II lifts the rest. The two stages leave different footprints. Stage I lifts the three terminal and repair benchmarks that resemble its training pool: it exceeds the released TMax-9B by 3.37 points on Terminal-Bench, 4.40 on SWE-bench Verified, and 2.55 on TBLite. Its gains stay close to its training pool. Stage II moves every benchmark but one. It adds 5.14 points on LiveCodeBench, 5.08 on BFCL-v3, and 16.00, 6.14, and 5.38 on τ^2-Bench airline, retail, and telecom, the largest gains in the table, on tasks the recipe never trained on. These gains extend beyond the training task types without replay. Across the ten benchmarks InScribe has the highest mean on nine.

The transfer follows the behaviors Stage II rewards. We did not train on function calling, customer-service dialogue, or competitive programming, so the τ^2-Bench and BFCL-v3 gains are transfer, not fit. The case studies in Appendix C and D.3 point to why: the behaviors that Stage II rewards on residual tasks, checking the final state before declaring success and locating a fault before editing, are also what multi-turn tool-use benchmarks score.

4.3 DOES THE GAIN LIVE IN THE WEIGHTS?

If harness training left its benefit in the generated harness, taking that harness away would remove the benefit. Table 3 tests this on Terminal-Bench 2.0 by holding each checkpoint fixed and varying only the execution harness: stock, generated, Terminus2, and OpenHands. No hints are supplied.

The ordering holds under every harness. In all four columns the means rise from stock-harness RL to Stage I to InScribe. The gain therefore does not depend on the policy running under a harness it wrote: under Terminus2 and OpenHands, programs it never saw in training, InScribe leads stock-harness RL by 7.49 and 5.62 points. A policy trained to write its own way of working is stronger when someone else writes it, which is what it means for the lesson to have moved into the weights.

Table 3: **Policy performance across evaluation harnesses on TB2.** Mean scores (%, higher is better) over three evaluation runs, reported as mean ± standard deviation, as in Table 2. No teacher hints are supplied. Stock denotes the fixed default; Generated denotes a rollout-local adaptation of it. Terminus2 statistics are reused from Table 4. Bold marks the largest mean per column.

Checkpoint	Stock	Generated	Terminus2	OpenHands
Stock-harness RL	17.23 ± 1.30	16.10 ± 1.29	23.60 ± 1.13	20.22 ± 1.13
Stage I only	18.72 ± 1.72	20.22 ± 1.13	26.22 ± 1.30	23.79 ± 1.18
InScribe	$\mathbf{19.85} \pm 1.71$	$\mathbf{23.22} \pm 1.30$	$\mathbf{31.09} \pm 1.29$	$\mathbf{25.84} \pm 1.13$

The generated harness is a scaffold, not the product. The Generated column measures the policy's own adaptation. For the stock-harness RL checkpoint, which never practiced writing a harness, running under its own generated harness costs 1.13 points relative to Stock. Stage I turns this into a gain of 1.50, and InScribe widens it to 3.37. The skill of writing a helpful harness thus grows along the training path. Generated trails Terminus2 and OpenHands in every row. Moving from Stock to Terminus2 adds 6.37 points for stock-harness RL, 7.50 for Stage I, and 11.24 for InScribe. The benefit of a good hand-written harness grows with harness training rather than shrinking. InScribe makes the policy better under whichever harness it is given.

4.4 What each stage contributes

Table 4 tests whether harness evolution beats longer stock-harness RL and whether hint RL benefits from it. Section 4.1 and Appendix B.2 define the controls.

Harness evolution beats more of the same training. Our stock-harness RL run closely matches released TMax-9B. With 100 fewer updates, Stage I improves on this baseline by 2.62 points on TB2 and 4.54 on SWE-V. At the same 300-update budget, InScribe widens these gains to 7.49 and 6.74 points.

Table 4: **Stage-level ablations.** Evaluation uses fixed benchmark harnesses without hints; reporting follows Table 2. C and D are matched Stage-I continuations without and with diagnoses, respectively. Training budgets and control definitions are in Appendix B.2.

Training path	TB2	SWE-V
Stock RL	23.60 ± 1.13	53.93 ± 1.30
Stage I only	26.22 ± 1.30	58.47 ± 1.70
Stock RL → hint	24.72 ± 1.12	53.00 ± 1.44
Base → hint	25.09 ± 1.29	52.53 ± 1.55
Stage I → no hint (C)	25.84 ± 1.13	57.40 ± 1.40
Stage I → hint (D, InScribe)	$\mathbf{31.09} \pm 1.29$	$\mathbf{60.67} \pm 1.72$

Practice without new experience does not help. Control C continues from the Stage-I checkpoint on the same record sequence as D, with every diagnosis removed. One hundred further updates leave it where Stage I was: 25.84% and 57.40%, within run-to-run variation of the 26.22% and 58.47% it started from. D, on the same tasks in the same order with the diagnoses visible, reaches 31.09% and 60.67%, 5.25 and 3.27 points above C; since the two differ only in whether the diagnosis is visible, that gap is attributable to the diagnoses. The credit rules of Section 3 predict as much: residual tasks were selected because every rollout failed at θ_I, so their clean cohorts are predominantly all-zero and carry little solver signal, whereas a diagnosis that produces a success restores within-cohort contrast.

Harness evolution is the best first stage among those compared. Two paths put hint RL after something other than Stage I, each probing the residual pool of its own starting checkpoint. One hundred updates of hint RL from the base model reach 25.09% and 52.53%; hint RL after 200 updates of stock-harness RL reaches 24.72% and 53.00%. Both improve on the base model but fall 6 to 8 points short of the full recipe on each benchmark. The two are within a point of each other: 200 updates of stock-harness RL before hint RL add less than a point over skipping them. Both also trail control C, so Stage I without hints already beats hint RL from either other start. Among the three first stages compared, harness evolution leaves the policy best placed for the hint stage, which is the order the pilot suggested.

4.5 What Stage I learns to write

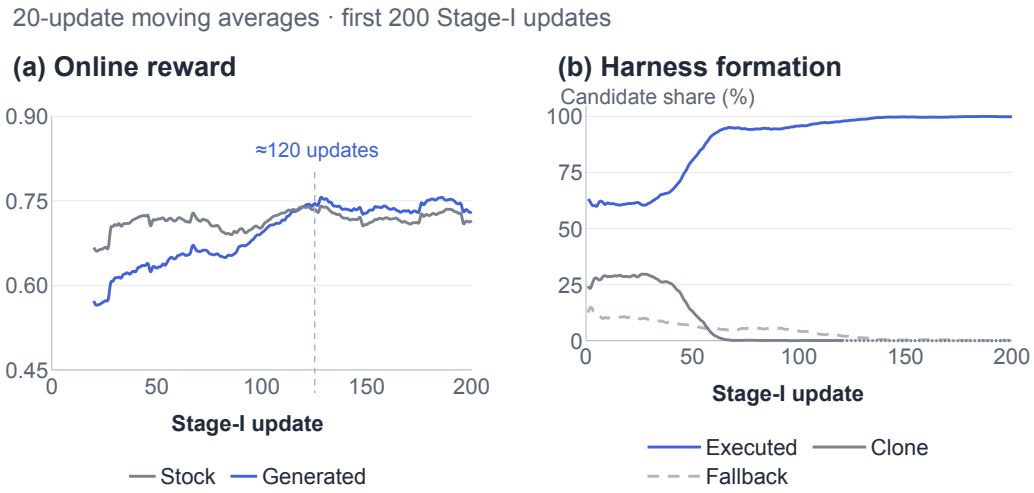


Figure 3: **Stage-I training dynamics** over the first 200 updates of the training run. (a) Online reward of the stock anchor and of the generated-harness candidates. (b) Share of candidates that execute a valid non-stock harness, clone the stock harness, or fall back.

Figure 3 separates harness formation from online utility on the Stage-I training run. Panel (b) shows valid non-stock execution becoming prevalent during training: early on, a substantial share of candidates clone the stock harness or fall back after an invalid generation, and by the second half of the run essentially every candidate executes a harness that passes the schema, validates, and differs from the default. In panel (a), the smoothed reward of generated-harness rollouts overtakes the stock anchor around update 120 and stays above it for the rest of training. Because the anchor and the candidates share the task, policy version, and rollout limits within each cohort, this is evidence that the learned adaptations improve online reward on the sampled training tasks, not merely that the generated harnesses are executable.

5 Conclusion

This paper asked what should happen to a task an agent keeps failing, and where whatever it learns from that task should end up. Our answer is that harnesses and hints are scaffolds for learning rather than the place where learning should live. InScribe acts on that answer in two stages of on-policy RL: the policy first learns to rewrite its own harness for each task, and then, on the tasks that the probe still leaves unsolved, continues with a fixed teacher diagnosis as context.

The evidence supports the position on three points. A 9B student trained this way gains 10.12 and 11.00 points over its base model on Terminal-Bench 2.0 and SWE-bench Verified, and the ordering stock-harness RL $<$ Stage I $<$ InScribe holds under all three fixed evaluation harnesses, including two hand-written ones it never trained with, so the gain lives in the weights. Continuing RL on the residual tasks without diagnoses leaves scores where Stage I left them, whereas the same tasks with diagnoses raise them: a task on which every rollout fails gives the policy little to learn from by itself, and a diagnosis is what puts signal back. Hint RL from the base model or after stock-harness RL falls 6 to 8 points short of the full recipe, so among the first stages we compared, harness evolution is the one to run first.

Taken together, the experiments show that what an agent needs in order to stop failing a task, a way of working or a piece of information, can be supplied during training and then withdrawn, leaving a policy that needs neither at test time. A longer-range question is whether what the policy internalizes under today's harnesses survives the ones that replace them.

AI USE STATEMENT

In this work, we used generative AI tools to aid and polish writing; retrieve and discover relevant literature; support research ideation and execution, including software development and experiment diagnostics; draft sections of the paper; and generate synthetic data, specifically the teacher diagnostic hints used in training. We did not use generative AI tools to prove mathematical claims. The authors reviewed all AI-assisted code, analyses, citations, and prose for correctness and attribution, and validated the empirical claims against the underlying experiment artifacts. Generative AI tools did not make submission or authorship decisions. We take responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies tool-using agents that act in sandboxed environments. Such agents can produce insecure or harmful software if deployed without appropriate controls. Our experiments use publicly released training tasks and benchmark environments with their verifiers; we do not collect new data, target real systems, or use private user data. Training tasks follow the released TMax and R2E-Gym-Subset data and were deduplicated against the evaluation benchmarks. We report infrastructure-invalid episodes separately from model failures so that they do not bias evaluation or residual-task selection.

REPRODUCIBILITY STATEMENT

Section 4 describes the training recipe, the control designs, and the evaluation reporting convention. The appendix gives the shared training settings (Appendix B.2), the Stage-I harness schema and execution-status rules (Appendix C), the residual-task selection rule and the balanced clean/hinted record construction (Appendix D), the teacher prompt and input-compression rules (Appendix D.3), and the motivation-study protocol (Appendix A). The training pool, evaluation benchmarks, and evaluation harnesses are public. We are currently discussing the open release of our code, trained model checkpoints, and data with our organization. We intend to make these artifacts publicly available, subject to organizational approval and compliance with the organization's requirements.

REFERENCES

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. URL https://arxiv.org/abs/2506.07982.

Lirong Che, Yuzhe Yang, Peiwen Lin, Chuang Wang, Xueqian Wang, and Jian Su. DemoEvolve: Overcoming sparse feedback in agentic harness evolution with demonstrations. *arXiv preprint arXiv:2605.24539*, 2026. URL https://arxiv.org/abs/2605.24539.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, Huaxiu Yao, Hengduo Li, Jiacheng Zhu, Xian Li, Dawn Song, Bo Li, Jason Weston, and Dat Huynh. Scaling agent learning via experience synthesis. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/c53793534833d30a40f3d18352735519-Abstract-Conference.html.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. Agent-RLVR: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425*, 2025. URL https://arxiv.org/abs/2506.11425.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948v1.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026. URL https://arxiv.org/abs/2606.19348.

Yangyi Fang and Haolin Shi. Placing puzzle pieces where they matter: A question augmentation framework for reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 21165–21183, 2026. doi: 10.18653/v1/2026.acl-long.969. URL https://aclanthology.org/2026.acl-long.969/.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025a.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. *arXiv preprint arXiv:2505.10978*, 2025b. URL https://arxiv.org/abs/2505.10978.

Xinshun Feng, Xinhao Song, Lijun Li, Gongshen Liu, and Jing Shao. SEARL: Joint optimization of policy and tool graph memory for self-evolving agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 24518–24535, 2026. doi: 10.18653/v1/2026.acl-long.1125. URL https://aclanthology.org/2026.acl-long.1125/.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Yuxiang Ji, Ziyu Ma, Yong Wang, Guanhua Chen, Xiangxiang Chu, and Liaoni Wu. Tree search for LLM agent reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/8c7304e77c832ddc70075dfee081ca6c-Abstract-Conference.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

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

Huanyu Liu, Jia Li, Yihong Dong, Chang Yu, Taozhi Chen, Lecheng Wang, Yongding Tao, Bin Gu, and Ge Li. EvoCoT: Overcoming the exploration bottleneck in reinforcement learning for LLMs. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20589–20603, 2026a. doi: 10.18653/v1/2026.findings-acl.1031. URL https://aclanthology.org/2026.findings-acl.1031/.

Siwei Liu, Jinyuan Fang, Han Zhou, Yingxu Wang, and Zaiqiao Meng. SEW: Self-evolving agentic workflows for automated code generation. *arXiv preprint arXiv:2505.18646*, 2025a.

Xiaoqian Liu, Ke Wang, Yuchuan Wu, Fei Huang, Yongbin Li, Jianbin Jiao, and Junge Zhang. Agentic reinforcement learning with implicit step rewards. In *International Conference on Learning Representations*, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/d1e6739f319be4ba8e748cc05f23bba1-Abstract-Conference.html.

Zeyuan Liu, Jeonghye Kim, Xufang Luo, Dongsheng Li, and Yuqing Yang. Exploratory memory-augmented LLM agent via hybrid on- and off-policy optimization. In *International Conference on Learning Representations*, 2026c. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/b589d92785e39486e978fa273d0dc343-Abstract-Conference.html.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025b. URL https://arxiv.org/abs/2503.20783.

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

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Xuying Ning, Dongqi Fu, Tianxin Wei, Hanqing Zeng, Yuanchen Bei, Bingxuan Li, Zihao Li, Qifan Wang, Xiang Shen, Yifan Wu, Jiayi Liu, Hong Li, Yinglong Xia, Xiangjun Fan, Hanghang Tong, and Jingrui He. EvoHarness-RL: Learning self-evolving runtime harness for long-horizon LLM agents. *arXiv preprint arXiv:2608.05446*, 2026. URL https://arxiv.org/abs/2608.05446.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/980ea04d23d1f6908964eba2a74afe45-Abstract-Conference.html.

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

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, and Shuiwang Ji. Curriculum reinforcement learning

from easy to hard tasks improves LLM reasoning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/fe1d9d163a8b88f413553d99cc48fa5a-Abstract-Conference.html.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling LLM terminal capabilities. *arXiv preprint arXiv:2602.21193*, 2026. URL https://arxiv.org/abs/2602.21193.

Qwen Team. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5, 2026. Model: https://huggingface.co/Qwen/Qwen3.5-9B.

Negin Raoof, Richard Zhuang, Marianna Nezhurina, Etash Guha, Atula Tejaswi, Ryan Marten, Charlie F. Ruan, Tyler Griggs, Alexander Glenn Shaw, Hritik Bansal, et al. OpenThoughts-Agent: Data recipes for agentic models. *arXiv preprint arXiv:2606.24855*, 2026. URL https://arxiv.org/abs/2606.24855.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

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. URL https://arxiv.org/abs/2402.03300.

Leyang Shen, Yang Zhang, Chun Kai Ling, Xiaoyan Zhao, and Tat-Seng Chua. CARL: Critical action focused reinforcement learning for multi-step agent. *arXiv preprint arXiv:2512.04949*, 2025. URL https://arxiv.org/abs/2512.04949.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.18449.

Jiahao Wu, Ning Lu, Shengcai Liu, Kun Wang, Yanting Yang, Li Qing, and Ke Tang. Train at moving edge: Online-verified prompt selection for efficient RL training of large reasoning model. *arXiv preprint arXiv:2603.25184*, 2026. URL https://arxiv.org/abs/2603.25184.

Yu Xia, Canwen Xu, Zhewei Yao, Julian McAuley, and Yuxiong He. Learning to hint for reinforcement learning. *arXiv preprint arXiv:2604.00698*, 2026. URL https://arxiv.org/abs/2604.00698.

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*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025. URL https://arxiv.org/abs/2504.21798.

Matthew Y. R. Yang, Hao Bai, Ian Wu, Gene Yang, Amrith Setlur, and Aviral Kumar. InT: Self-proposed interventions enable credit assignment in LLM reasoning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/89062e4d480c0c3a88d36c20c5694459-Abstract-Conference.html.

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. URL https://arxiv.org/abs/2210.03629.

Linhao Yu, Tianmeng Yang, Siyu Ding, Renren Jin, Naibin Gu, Xiangzhao Hao, Shuaiyi Nie, Deyi Xiong, Weichong Yin, Yu Sun, and Hua Wu. KnowRL: Boosting LLM reasoning via reinforcement learning with minimal-sufficient knowledge guidance. *arXiv preprint arXiv:2604.12627*, 2026. URL https://arxiv.org/abs/2604.12627.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. AgenTracer: Who is inducing failure in the LLM agentic systems? In *International Conference on Learning Representations*, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/134ed7a477770f227f12450ef0cbb8f4-Abstract-Conference.html.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/8a94ff6f922d995d7d3f4ebf4143e442-Abstract-Conference.html.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024.

Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. *arXiv preprint arXiv:2506.10943*, 2025.

A Frozen-Checkpoint Diagnosis Pilot

The pilot of Section 1 holds the weights fixed and changes one part of the setup at a time, so that a recovered task can be attributed to that change rather than to learning.

Cohort. We take the Qwen3.5-9B base model and run it for eight episodes on every task in the Stage-I training pool under the stock harness. From the tasks with no verified success in any of the eight episodes we sample 100 at random. Tasks, verifiers, and the model are then frozen for all interventions.

Interventions. Each task is run for eight episodes under Terminus2 and eight under OpenHands v2, with the same model and the same decoding settings as the baseline (temperature 1.0, top-p 0.95, top-k 20, 262K context, 81,920-token output cap); Terminus2 uses its JSON terminal protocol and OpenHands v2 its code-tool interface. A task counts as recovered if at least one episode passes the verifier. For the tasks that neither harness recovers, a short diagnostic hint written by DeepSeek-V4-Flash (DeepSeek-AI, 2026) with the prompt of Appendix D.3 is placed in the context, and the task is run again under the stock harness for eight episodes. Every arm therefore has the same episode budget as the baseline.

Results. Table 5 reports task counts. Terminus2 recovers 16 tasks and OpenHands v2 recovers 26, with 10 in common, so 32 of the 100 are recovered under at least one of the two alternative harnesses, 16 new attempts per task in total. The hint recovers 13 of the remaining 68 under the very harness that had failed them. Figure 1 plots the same counts over the 100-task denominator.

Table 5: **Frozen-model pilot.** Tasks recovered by each intervention out of 100 tasks with no verified success in eight episodes under the stock harness. Every arm uses eight episodes per task; the hint is applied only to the 68 tasks that neither harness recovers.

Intervention	Recovered	Still failing
Terminus2	16 / 100	84
OpenHands v2	26 / 100	74
Either harness	32 / 100	68
Diagnostic hint, stock harness, on the remaining 68	13 / 68	55

B Experimental Details

B.1 Benchmarks and evaluation

The training pool follows the TMax data and R2E-Gym-Subset sources (Ivison et al., 2026; Jain et al., 2025)[1], with an executable environment and verifier for each task. It is deduplicated against Terminal-Bench 2.0 and SWE-bench Verified.

Terminal-Bench 2.0 is evaluated under Terminus2, following its official implementation; SWE-bench Verified is evaluated under OpenHands 1.6.0 (Merrill et al., 2026; Jimenez et al., 2024; Wang et al., 2025). The broader suite includes OpenThoughts-TBLite (Raoof et al., 2026)[2], LiveCodeBench v6 (Jain et al., 2024), BFCL v1–v3 (Patil et al., 2025), and the airline, retail, and telecom domains of τ^2-Bench (Barres et al., 2025), using an in-house harness that runs each benchmark's official scoring. Decoding follows the officially recommended settings for Qwen3.5-9B. Every checkpoint is evaluated three times on each benchmark; tables report the mean and sample standard deviation of the three run-level scores, in percentage points. No teacher hint or generated harness is present except in the Generated column of Table 3, where the policy adapts the stock harness afresh for each rollout. Cross-harness comparisons hold the checkpoint and tasks fixed. All InScribe entries in Table 2 come from one Stage-II checkpoint.

[1] https://huggingface.co/datasets/R2E-Gym/R2E-Gym-Subset

[2] https://huggingface.co/datasets/open-thoughts/OpenThoughts-TBLite

B.2 TRAINING SETTINGS AND CONTROLS

The student is Qwen3.5-9B. Stage I trains for 200 optimizer updates on the 10,000-task pool of Section 4.1; Stage II continues from that checkpoint for 100 updates on the residual records of Appendix D. Table 6 lists the settings. Stage II lowers the learning rate and leaves every other setting unchanged. Stock-harness RL uses the same optimizer and batch under the stock harness for 300 updates. Two details of the two-channel update of Section 3.2: a cohort is discarded when its harness and solver masks overlap or either channel is empty, and the learner recomputes Z_{sol} and Z_{har} exactly over each padded batch rather than approximating them per minibatch.

Table 6: **Training settings for the two stages.**

Setting	Stage I	Stage II
Optimizer updates	200	100
Cohorts per update	16	same
Rollouts per cohort	1 loss-masked stock anchor + 8 candidates	same
Learning rate	10^{-6}, constant	5×10^{-7}, constant
Optimizer	Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$), weight decay 0.017	same
Gradient clipping	0.3	same
Policy clipping	$\epsilon = 0.15$ below, 0.20 above	same
KL regularization	none	same
Advantage normalization	centered within the 8 candidates, no variance scaling	same
Reward clipping	20	same
Minibatches per update	1, with recomputed reference log-probabilities	same
Staleness gate	cohorts at most 2 policy versions behind	same
Sampling	temperature 1.0, top-p 1.0, no top-k truncation	same
Context / generation length	65,536 / $\leq$16,384 tokens per turn	same
Harness specification cap	8,192 estimated tokens	same
Solver turns	64, plus one harness-writing turn and one repair attempt	same
β (harness-channel weight)	0.1	same
λ (invalid-generation penalty)	0.05	same
Hardware	32 A100 GPUs on 4 nodes	same

The five control paths are defined as follows:

- **Stock-harness RL:** 300 updates under the stock harness with the same optimizer and batch, without harness generation or hints, matching the full recipe's total update count.
- **Stock RL $\rightarrow$ hint RL:** 200 stock-RL updates followed by 100 hint-RL updates, matching the full two-stage schedule.
- **Base $\rightarrow$ hint RL:** 100 hint-RL updates from the base model, without Stage I.
- **Stage I only:** the checkpoint before continuation.
- **Stage I $\rightarrow$ no-hint RL (C):** the same continuation as the full recipe (D), replacing each hinted record with its clean copy while keeping task identities, order, seed, optimizer, and budget.

For Stock RL $\rightarrow$ hint RL and Base $\rightarrow$ hint RL, residual pools are probed from each path's own starting checkpoint and diagnoses are regenerated for that checkpoint. Each hint stage therefore trains on the tasks its own starting policy fails.

B.3 RUN FOOTPRINT

Table 7 reports the resources consumed by the two training stages, read from the run logs: wall-clock time from process start to the last update of the stage, GPU allocation, sandbox episodes, and the number of cohorts rolled out against the number the learner accepted. Sandboxes run on a separate CPU pool, so sandbox-hours and GPU-hours are concurrent rather than additive.

Table 7: **Run footprint of the two stages.** GPU-hours are the 32-GPU allocation held for the stage's wall-clock time. A cohort is one stock anchor plus eight candidates; the learner consumes 16 accepted cohorts per update. Sandbox episodes count every launched episode, including retries after failed launches, so they exceed nine per rolled-out cohort.

	Stage I (200 updates)	Stage II (100 updates)
Wall-clock (h)	114.5	63.5
GPUs	32 A100 (24 inference, 8 training)	same
GPU-hours	3,663	2,032
Sandbox episodes	43,298	20,823
Sandbox-hours	13,538	3,313
Cohorts rolled out / accepted	4,768 / 3,200	2,208 / 1,600

Why more cohorts are rolled out than accepted. The learner accepts a cohort only if all nine of its rollouts were generated by the same policy version (Appendix B.4). Rollouts are long-horizon episodes, a median sandbox lifetime of about seven minutes with a tail above twenty-five, and an optimizer update completes roughly every 34 minutes in Stage I, so a cohort whose nine episodes straddle a weight update is discarded whole rather than mixed across versions. This rule removed roughly a third of the rolled-out cohorts in Stage I and just over a quarter in Stage II; a smaller number were dropped by the controller for infrastructure errors or incomplete slots before they finished. Rejected cohorts never reach the learner and contribute no gradient; the accepted count is what "16 cohorts per update" refers to.

B.4 Asynchronous training system

Generation and optimization run concurrently: an inference pool serves rollouts while the learner consumes completed cohorts, since a synchronous barrier would leave the learner waiting on the slowest of nine tool-using trajectories at every update. One rollout job returns one cohort of $G + 1$ rollouts for the same task, and all $G + 1$ must come from the same policy version; a cohort spanning versions is rejected whole, so the comparison $r_i - r_0$ in Equation (2) is always taken within a single policy. A staleness gate admits cohorts at most two versions behind the current policy, and a per-cohort deadline keeps one slow episode from stalling the queue.

C Stage-I Harness Details

C.1 Harness specification and runtime mapping

Specification. A harness is one strict JSON object, `ScaffoldSpecV1`, with nine top-level components (Table 8). The stock harness S_0 is the TMax Bash agent expressed in this format: one builtin `bash` tool, a six-phase workflow, full history retention with a 64-message and 140,000-character window, verification before submission, diagnose-on-failure recovery, and termination on the completion marker. A generated harness must contain every component; unknown or missing fields, duplicate keys, implicit defaults, and out-of-range values are rejected, and the canonical object is capped at 8,192 estimated tokens. The `bash` tool must keep its builtin implementation; any additional tool must ship Python source that compiles. Two hashes are recorded per candidate: one of the requested object and one of the object that actually ran.

What the runtime enforces and what it renders. Two kinds of field must be distinguished (Table 9). Some fields change the executable behavior of the solver loop: the task template becomes the user message; the tools become the function schemas the model may call, with generated Python installed in the sandbox and run by `python3` on a JSON argument object under the same time limit as bash; the tool policy and tool choice are enforced per turn, so a response with the wrong number of tool calls is returned as a format error; a required-reasoning policy rejects responses whose text is empty; the context fields set the history-truncation policy; and a `one_extra_turn` confirmation defers verification by one turn after the completion marker. The remaining fields are rendered, by a fixed template, into the solver's system prompt as policy text: the planning statement, skills, phases and exit conditions, error and no-progress policies, verification commands, recovery instruction, and

Table 8: **Components of `ScaffoldSpecV1` and their constraints.** Text limits are in characters; enumerations list all admitted values.

Component	Fields and constraints
`v`	constant `"ScaffoldSpecV1"`
`prompt`	`system` (≤32,000); `task_template` (≤24,000, must contain {`task`}); `planning` (≤16,000)
`skills`	1–8 items, each `name` (lowercase identifier), `trigger` (≤4,000), `instructions` (≤16,000)
`tools`	1–8 items, each `name`, `description` (≤4,000), a JSON-Schema `parameters` object, and `implementation`: `builtin:bash` for the bash tool, or Python source (≤32,000) that must compile
`loop`	`phases`: 1–8 items with `name`, `instruction` (≤8,000), `exit_condition` (≤4,000); `reasoning` ∈ {required, optional}; `tool_policy` ∈ {exactly_one, zero_or_one}; `tool_choice` ∈ {required, auto}; `tool_error` ∈ {surface, diagnose_then_retry, replan}; `verify_fail` ∈ {repair, diagnose_then_repair, replan}; `no_progress` ∈ {inspect, replan, terminate}; `observation_template` (contains {`observation`}); `format_error_template` (contains {`error`})
`context`	`retention` ∈ {full, sliding_window}; `max_messages` 2–64; `max_chars` 8,000–140,000; `message_max_chars` 1,000–24,000; `preserve_initial` (bool)
`verification`	`when` ∈ {before_submit, after_changes, every_phase}; `commands`: 1–8 strings (≤4,000 each); `require_all` (bool)
`recovery`	`strategy` ∈ {diagnose, replan, rollback_then_replan}; `diagnostic_prompt` (≤12,000)
`termination`	`condition` ∈ {task_complete, verification_passed, task_complete_and_verified}; `confirm` ∈ {none, self_check, one_extra_turn}; `instruction` (≤8,000)

termination condition. They guide the model but are not executed or checked by the runtime; in particular, verification commands are not run by the harness, and the system prompt closes with a fixed sentence stating that the outer kernel still enforces the turn, token, and time limits, the sandbox, the completion marker, the hidden verifier, and the reward.

Generation turn. Before the first environment interaction, each candidate slot receives one assessment message containing the complete stock specification as JSON, the list of mutable components, the list of fixed outer-kernel constraints, the task instruction, and, for the six directed slots, an assigned exploration direction (planning and decomposition; phase graph and explicit exit conditions; executable verification and falsification; failure recovery and replanning; context and evidence retention; termination and loop efficiency). The direction asks for at least one task-relevant behavioral change on that axis, forbids decorative edits, and permits returning the exact stock specification when the axis is irrelevant. The two free slots receive the same message without a direction. The policy answers with ordinary assistant text containing exactly one fenced `json` block; a rejected object is given one repair attempt with the validator's error. The parsed object is canonicalized, hashed, and compared with the stock hash: an identical object is a CLONE, a valid different object is EXECUTED, and an object that fails validation twice is a FALLBACK, after which the stock harness runs and the candidate receives the penalty λ.

Generation prompt. The assessment message is assembled from the fixed templates reproduced below. The stock specification and the JSON schema that the message embeds are elided here; their content is given in Tables 8 and 9.

Harness-generation system prompt

Verbatim

```
You design the complete task-local solver scaffold.

Return one complete ScaffoldSpecV1 replacement as ordinary assistant
text, inside exactly one fenced `json` block.  Do not call a tool and
do not stringify nested objects or arrays.  Every required component
must be present:  prompt, skills, tools and their executable
```

Table 9: **How each field reaches the runtime.**

Field	Effect
Enforced by the runtime	
`prompt.task_template`	user message, with {`task`} replaced by the instruction
`tools`	function schemas offered to the model; generated tools installed and executed in the sandbox
`loop.tool_choice`, `loop.tool_policy`	request tool-choice setting; number of tool calls per turn checked
`loop.reasoning`	empty reasoning rejected when `required`
`loop.observation_template`, `loop.format_error_template`	rendering of tool output and of format errors
`context.*`	history truncation: retention mode, message and character limits, initial-message preservation
`termination.confirm = one_extra_turn`	completion marker triggers one further turn before verification
Rendered into the system prompt only	
`prompt.system`, `prompt.planning`, `skills`	instruction text
`loop.phases`, `loop.tool_error`, `loop.verify_fail`, `loop.no_progress`	workflow and policy text
`verification.*`, `recovery.*`	suggested checks and recovery instruction; commands are not executed by the harness
`termination.condition`, `termination.instruction`, `confirm = self_check`	stopping instruction text
Fixed outside the harness	
	solver turn cap, model context cap, sandbox and network policy, task verifier, completion marker, reward

```
implementations, agent loop, context retention, verification,
recovery, and termination policy.  The fenced payload must be one
strict JSON object:  duplicate keys, unknown or missing fields,
implicit defaults, coercion, and value clamping are rejected.

You may reproduce the canonical stock specification exactly; that is
audited as a valid clone.  Any non-identical specification is a full
replacement.  There is no partial mutation or short decision branch.
The canonical artifact limit is 8192 estimated tokens and artifact
cost is zero, so all components and generated code should be chosen
for task performance rather than length pricing.  Generated Python
tools read one JSON argument object from stdin and write their result
to stdout.  They run with the same task-sandbox identity, filesystem,
network policy, and time limits as builtin bash.

Do not solve the task during this assessment.  Do not request archive
history, hidden verifier data, credentials, secrets, cross-episode
persistence, or privileges outside the task sandbox.
```

Harness-generation user message

Verbatim skeleton; the embedded stock specification and schema are elided

```
Stock scaffold label:  {label}

### Canonical complete stock ScaffoldSpecV1

[the stock specification as indented JSON]

### Strict replacement JSON Schema
```

```
[the schema as indented JSON]
### Mutable task-local boundary
The replacement owns prompt/task rendering; skills; function schemas
and generated Python implementations; loop phases, reasoning and tool
policy; observation/error formatting; context retention; verification
commands; recovery; and termination policy.
### Immutable outer evaluation kernel
- Solver action hard cap:  64
- Model context hard cap:  65536
- Original task/sample and normal /app or /testbed workdir
- Sandbox identity, isolation, filesystem/network policy and timeouts
- Hidden verifier, reward, paired credit, training algorithm and data
- Completion marker recognition and hard token/turn/time termination
- Episode-local scope; no inheritance, promotion, secrets or archive
read
### Current task
[task instruction]
Assess and configure only; do not solve yet.
### Assigned exploration direction   [directed slots only]
Primary design axis:  {axis}.
Make at least one task-relevant behavioral change on this axis when it
is applicable.  The change must affect an executable solver behavior;
whitespace, quoting, field order, comments, or pure paraphrase do not
count.  Preserve unrelated fields unless changing them is necessary,
and do not invent tools or dependencies merely to satisfy the axis.
You may combine the primary axis with other useful mechanisms.  If the
axis is genuinely irrelevant to this task, return the exact canonical
stock specification instead of a decorative change.  All original
schema, sandbox, and safety constraints remain active.
```

The six axis strings, assigned to the directed slots in this order, are *planning and decomposition*; *phase graph and explicit exit conditions*; *executable verification and falsification*; *failure recovery and replanning*; *context and evidence retention*; and *termination and loop efficiency*. A rejected object receives one retry message: "The full scaffold replacement was rejected: [validator error]. Retry once by returning exactly one fenced json block containing a complete strict ScaffoldSpecV1 object. Re-send every top-level component, including prompt.planning, and keep the canonical object within 8192 estimated tokens. Do not call a tool and do not solve the task yet."

C.2 Qualitative examples

The three cases below each compare a successful adapted-harness rollout with a failed stock-harness rollout from the same Stage-I training cohort. The quoted instructions were generated by the policy and loaded into the solver's context; they guide execution and do not run tests or enforce correctness.

Case 1. Make completion depend on the final state

Network monitoring — Stock: fail — Adapted: pass

Task. Build a network monitor, configure a five-minute cron schedule, and save a backup of the configured crontab.

Harness adaptation. The generic workflow is replaced with task-specific phases: setup → create script → execute → schedule → submit. The new scheduling phase contains these original instructions:

> *"Configure execution scheduling via cron and save the crontab backup."*
> *"The cron job is configured in the user's crontab and backed up."*

The second sentence is the phase's exit condition. The harness also requests route-status and connectivity-log checks.

Stock execution. The student backs up an empty crontab *before* installing the schedule. The final backup is empty even though the schedule itself is installed.

Adapted execution. The student installs the schedule first, then exports and inspects the crontab. The backup contains the five-minute entry. It also performs the specified route-status and log-length checks and passes the verifier.

What this illustrates. The harness makes a multi-step completion condition explicit: back up the *resulting* system state, not its initial state.

Case 2. Stage the pipeline and gate submission on assertions

Video, metadata, and CI pipeline — Stock: fail — Adapted: pass

Task. Parse a state-machine event log, extract video frames with ffmpeg only while the log is in the RECORDING state, write a metadata sanitiser that passes a clean corpus and rejects an evil one, and wrap both in a CI script that asserts the pass and fail rates.

Harness adaptation. The six generic phases become parser → video_extract → sanitiser → pipeline → assertions → submit, each with a checkable exit condition:

> *"Video extraction script extracts frames during RECORDING state only."*
> *"Sanitiser script passes all clean and fails all evil metadata."*

Verification lists three commands: the sanitiser's PASS count on the clean corpus, its FAIL count on the evil corpus, and a check that the frames directory is non-empty. The recovery strategy becomes rollback-then-replan, the loop policy switches to inspect on no progress and replan on failed verification, and termination is gated on the assertions: *"Only submit when all tests pass."*

Stock execution. The log places RECORDING between 5 and 15 seconds, so one frame per second gives ten frames. The student's extraction script produces fifteen, its CI script reports the count without comparing it with the log, and it submits with the wrong frame set and fails.

Adapted execution. The student writes and tests the log parser first, extracts with an explicit 5-to-15-second window and obtains ten frames, catches a sanitiser that lets two evil samples through and rewrites it, then runs the CI script and the three assertions before submitting. It passes in 33 turns.

What this illustrates. Phase exit conditions turn each component into a deliverable, and the assertion-gated termination makes submission depend on checks that the stock rollout never made.

Case 3. Keep the student on the required build path

Go and C log analyzer — Stock: fail — Adapted: pass

Task. Repair a polyglot log analyzer: initialize the Go module, fix a buffer overflow in a C parser linked through CGo, fix a data race in the Go orchestrator, build, run the analyzer on a log, and write the flagged URIs sorted and deduplicated.

Harness adaptation. The phases follow the repair: analyze → setup_module → fix_c_memory → fix_go_race → build_success → run_analyzer → process_output → submit. The module phase reads:

> *"Initialize Go module with path 'security/loganalyzer', identify build dependencies, configure CGo linking."*

and the race phase exits only when "Data race in concurrent goroutines fixed." A security-analysis skill directs the student to read the C and Go code for memory-safety and concurrency issues before editing. Verification checks that the binary and the sorted output file exist. This harness was written midway through Stage I training.

Stock execution. When CGo fails because the C headers are missing, the student abandons CGo: it deletes the C sources, rewrites the orchestrator in pure Go, later recreates a "fixed" parser in a subdirectory the build does not use, and after 63 turns submits a binary without the C parser, which fails.

Adapted execution. Facing the same missing headers, the student inspects the empty include directory, installs the C development package, keeps the CGo structure, fixes the overflow and the race, builds, runs the analyzer, and writes the sorted output. It passes in 36 turns.

What this illustrates. A phase graph that names the required components keeps the student repairing the environment instead of removing the component that fails to build.

C.3 Credit derivation

When every candidate is EXECUTED, $\delta_i = r_i - r_0$ and the common stock reward cancels under centering:

$$A_i^{\mathrm{har}} = (r_i - r_0) - \frac{1}{G}\sum_j (r_j - r_0) = r_i - \frac{1}{G}\sum_j r_j = A_i^{\mathrm{sol}}. \tag{7}$$

The anchor therefore affects credit only in cohorts with mixed execution status. When every verifier reward is zero, every solver advantage is zero, and the raw harness delta is 0 for executed candidates and clones and $-\lambda$ for fallbacks. With k fallbacks among G candidates, centering gives the non-fallback candidates $+\lambda k/G$ and the fallbacks $-\lambda(G-k)/G$; if all candidates share one status, the harness channel carries no signal either.

The weight $\beta = 0.1$ keeps the short harness channel from dominating an update whose solver channel carries far more tokens; $\lambda = 0.05$ is small enough that a fallback ranks below a clone without outweighing task deltas. We did not tune either value.

D Stage-II Residual Curriculum and Teacher Hints

D.1 Residual task selection

The residual pool contains 2,500 tasks, not trajectories. Its construction follows Equation (5):

1. Freeze the checkpoint obtained after Stage I.
2. Run each training task through the Stage-I cohort: six directed and two free harness-writing candidates plus one stock anchor, without hints.
3. Admit the task only if all nine rollouts are infrastructure-valid and all nine verifier rewards are zero; an infrastructure failure is not a zero reward. Keep the failed traces for the teacher.
4. Freeze the resulting task list before Stage II begins.

D.2 Clean and hinted records

Section 3.3 defines the clean and hinted records, their 1:1 mixture, and the per-cohort credit they inherit from Stage I. The hint itself is generated once per task by DeepSeek-V4-Flash from the inputs and prompt of Appendix D.3. The no-hint control (C) replaces each hinted record with its clean twin and keeps task identities, order, seed, optimizer, and budget.

D.3 Teacher access and prompt

The two cases at the end of this section come from hint-assisted reattempts under the Stage-II teacher protocol of Section 3.3: the teacher sees the task instruction, a compressed failed attempt, and the private verifier script, and the student receives only the resulting hint. They illustrate the protocol, not the trained policy's hint-free performance. The teacher uses one system instruction for terminal and software-engineering tasks; its opening is reproduced below with line wrapping adjusted.

Teacher system prompt: role and inputs

Verbatim excerpt

```
You are an expert terminal-task tutor.  A student agent (a 9B model
working in a Linux sandbox via bash tool calls) failed a task.  You
see:
1.  the task instruction (the student saw this),
2.  a compressed trace of the student's failed attempt (the student
produced this),
3.  the PRIVATE verifier script that decides pass/fail (the student
can NEVER see this).
Produce ONE short hint that helps the student succeed on a FRESH
attempt from scratch.
```

Output contract. The remaining instructions request concept-level diagnosis, evidence already visible to the student, a next direction, and an observable self-check. They prohibit disclosure of verifier-only paths, test names, expected values, or other hidden specifics, and prohibit ready-made code and exact repair commands. The hints name source files, faulty conditions, and repair strategies; in a shuffled sample of 300 hints that we inspected, none disclosed anything visible only in the verifier.

The requested JSON fields are `failure_type`, `hypothesis`, `overlooked_evidence`, `next_direction`, `success_check`, and `confidence`. Failure type is selected from localization, planning, implementation, verification, and infrastructure; confidence is low, medium, or high. The four free-text fields have respective word/character limits of 40/320, 30/240, 40/320, and 25/200. Both limits apply to each field.

Teacher user-message template

Template with line wrapping and typography adjusted

````
## Task instruction (visible to student)
{instruction}

## Student failure trace (compressed; the student's own attempt)
- finish_reason: {finish_reason}, tool_calls: {num_calls},
  assistant_turns: {n_turns}
- last actions and observations:
{actions}
- final assistant message:
{final_msg}

## PRIVATE verifier script
(hindsight ground truth -- never reveal its internals)
```bash
{verifier}
```

Return the JSON hint now.
````

Input compression and variants. The failed attempts from the probe are concatenated into one record; the compressor then keeps the first 2,500 characters of the task instruction and at most the final eight assistant actions with their observations. Actions are truncated to their final 800 characters; observations retain error-biased excerpts of up to 700 characters. The compressor drops the earliest retained actions until the serialized failure record fits a 9,000-character budget. The verifier is truncated to its first 6,000 characters. A no-trace variant supplies the instruction and verifier without the failed attempt; a no-verifier variant supplies the instruction and failed attempt without the private script.

D.4 HINT EXAMPLES

Each case contrasts four unassisted retries with four hinted reattempts. The quoted fields are the teacher's output verbatim; the execution summary is from a successful student trajectory.

Hint case 1. Separate aggregation from self-confirmation

SQLite validation — No hint: 0/4 — Hint: 4/4

Task and initial failure. Repair a backup validator that reports each customer's order amount and item count. The student replaces an implicit cross join with explicit joins, but summing order amounts after joining items counts each order repeatedly. Its self-check reuses the same query and reports zero error; the verifier instead reports a mean absolute error of approximately 820.66.

Teacher diagnosis. *Implementation; high confidence.*

Hypothesis. *Joining orders to items multiplies each order row by its item count, inflating SUM(o.amount). Amount must be summed over orders alone; only item count needs the items join.*

Overlooked evidence. *Your self-check reused the same flawed join query, so MAE 0 against itself proves nothing. Customer_1 amount 306.96 is inflated by fan-out.*

Next direction. *Restructure: aggregate orders per customer independently from the items count (e.g., separate aggregation or subquery), then combine. Verify one customer's amount by hand against raw orders rows.*

Success check. *Pick one customer_id; SUM(amount) straight from orders table must equal your JSON total_order_amount for that customer.*

Student response. The student separately aggregates order amounts and item counts, combines the results, and independently checks a customer's sum against the orders table. It passes the verifier. The hint supplies the diagnosis and checking strategy; the student writes and executes the repair.

Hint case 2. Move debugging to the correct layer

Pillow image resizing — No hint: 0/4 — Hint: 2/4

Task and initial failure. Preserve transparent-white pixels when resizing RGBA or LA images with nearest-neighbor sampling. The failed attempt makes 50 tool calls and repeatedly edits and rebuilds C sampling code, while the incorrect alpha conversion occurs earlier in the Python wrapper.

Teacher diagnosis. *Localization; high confidence.*

Hypothesis. *Bug is in Python, not C: resize converts RGBA/LA to premultiplied RGBa before sampling regardless of filter. NEAREST must skip premultiplication; Geometry.c edits are the wrong layer.*

Overlooked evidence. *Your uint32 macro hack only patched C sampling; premultiplication happens earlier, in Image.py via convert('RGBa'), before any C code runs.*

Next direction. *Revert Geometry.c. In src/PIL/Image.py, gate the RGBa premultiply step on resample not being NEAREST, in resize and the sibling transform path. Pure-Python change, no rebuild.*

Success check. *Issue example's getcolors yields (255,255,255,0) for transparent half, and Pillow's existing resize/transform tests still pass.* (The expected pixel value is quoted from the issue text in the task instruction, which the student sees.)

Student response. The student edits the alpha-conversion conditions in the Python image module, checks the issue reproduction and resize tests, and passes the verifier. The trajectory includes further corrections, rather than a perfect one-shot patch. The hint itself is concrete repair guidance, not merely encouragement.

For the Pillow task, a hint generated by the no-verifier variant gives a similar diagnosis but yields zero successes in four reattempts, so the two hints are not interchangeable.