SCREENRSI: RECURSIVE SELF-IMPROVEMENT FOR GUI AGENT HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern agents combine a large language model (LLM) with a harness that prompts the model and interacts with the environment to complete user tasks. Although this code layer strongly shapes agent behavior, improving it still relies largely on engineers inspecting failures and revising the system by hand. Harness-level recursive self-improvement (HRSI) automates this process by evaluating tasks, analyzing failure modes, and revising the harness over successive rounds while keeping the model and environment fixed. Its central challenge is that sparse, local failure evidence guides changes to shared logic that can affect many other tasks. Successive revisions may therefore accumulate useful behavior or repeatedly break existing capabilities. We study how local repairs can yield broader gains under a limited evaluation budget and whether these gains transfer to unseen tasks. We introduce SCREENRSI, an HRSI framework for GUI agents that uses a semantic task-similarity graph to select failure seeds for coverage and diversity and samples previously successful tasks as preservation tasks. A recurring cycle of task selection, harness repair, and controlled merging turns execution evidence into validated harness revisions. On MobileGym, with Qwen3.8-27B fixed as the acting model and optimization confined to training tasks, the GPT-5.6-Sol optimizer produces a harness with a peak test success rate of 74.77%, improving over the initial harness by 15.71% and reducing false completion by 10.55%. Under the same round budget, graph-guided selection improves test success by 5.73% and reduces false completion by 3.52% relative to random selection. Several key mechanisms for improving the harness emerged during the evolution of SCREENRSI, including state management, stuck detection and recovery, task-specific prompt guidance, and output-format validation. These findings provide insight guidance for designing stronger harnesses. Code is available at `https://anonymous.4open.science/r/ScreenRSI-Review`.

1 INTRODUCTION

A harness is the code layer that connects a large language model (LLM) to its environment and turns model outputs into actions. The harness prompts the model, represents observations, manages interaction history, executes actions, and handles invalid outputs or execution errors. These mechanisms strongly shape agent behavior. Despite its importance, harness improvement still relies largely on manual engineering. Engineers inspect failed trajectories, identify possible causes, revise prompts or program logic, and test the resulting changes.

Research on Recursive Self-Improvement(RSI) explores how systems can instead improve from their own experience (Gao et al., 2026; Fang et al., 2025). We study *harness-level recursive self-improvement* (HRSI), which automates this engineering cycle while keeping the acting model and environment fixed. Experience collected under a harness h_r guides failure analysis, repair, and validation of a successor h_{r+1}. The revised harness then determines the behavior and evidence available in the next round.

The central challenge lies in the gap between local evidence and the broad effects of a repair. A failed trajectory reveals how the harness behaves on one task, yet a change to shared harness logic can affect many others (Zhang et al., 2026a;b). Over successive rounds, a repair may fix its target failure but disrupt tasks the harness previously handled well. Success on repair tasks also does not

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.

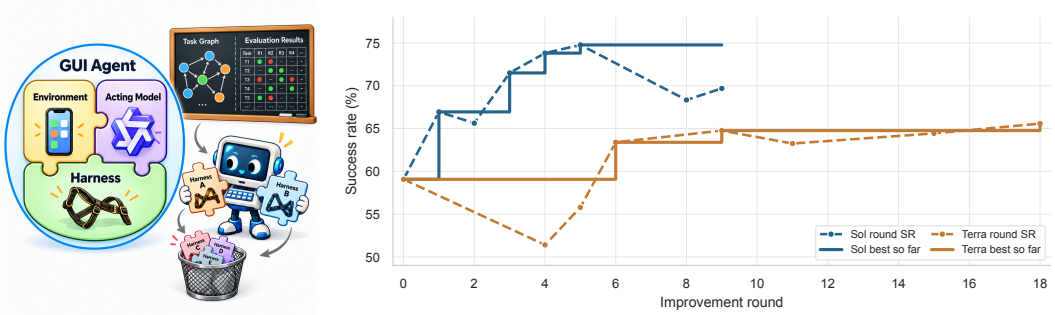


Figure 1: Harness-level recursive self-improvement under a limited evaluation budget. The plot shows test-set success rates across Sol and Terra optimization rounds, both starting from the initial harness at round 0. Dashed lines represents evaluation results of each round on test set; solid lines show the best observed rate.

guarantee gains on unseen tasks (Wang et al., 2026; Zhang et al., 2026b). GUI harnesses make this challenge harder: GUI agents use visual observations and low-level actions across long trajectories and multiple applications (Zhang et al., 2025), which raises the cost of reproducing failures and checking for regressions. On MobileGym (Wu et al., 2026), for example, 16 parallel workers need approximately four hours to evaluate the 256-task test set. HRSI therefore cannot test every candidate revision exhaustively. With a limited evaluation budget, it must choose which failures to repair and which previously successful tasks to use as preservation tasks. We ask: *How can local harness repairs bring broader gains and transfer to unseen tasks?*

We introduce SCREENRSI, an HRSI framework for GUI agents that uses a semantic task-similarity graph to allocate repair and validation effort. Our key insight is that tasks share behavioral structure across application boundaries. Tasks in different applications may require the same ability to retain observations, compare values, or recover from ineffective actions. It guides the selection of failure seeds for coverage and diversity from the failure tasks, and samples previously successful tasks as protected tasks. In each round, SCREENRSI selects tasks, then optimizer agent analyzes failure modes, repairs harness in isolated Git branches. Before merging, SCREENRSI verifies that each candidate fixes its target failures and preserves success on protected tasks.

We evaluate SCREENRSI on MobileGym with Qwen3.8-27B as the fixed acting model. All tasks used during the HRSI cycles come from the training set, and we evaluate the optimized harness on the test set. With GPT-5.6-Sol as the optimizer, the evolved harness reaches a test success rate of 74.77%, a 15.71% gain over the initial harness, while reducing false completion by 10.55%. Under the same round budget, graph-guided selection improves test success by 5.73% and reduces false completion by 3.52% relative to random selection. Analysis of the evolved harnesses identifies changes in state management, stuck detection and recovery, task-specific prompt guidance, and output-format validation. These emerging mechanisms can inform the design of more effective harnesses.

Our contributions are:

- **HRSI framework.** We introduce SCREENRSI, a framework for iterative self-improvement of GUI-agent harnesses through task evaluation, failure analysis, harness repair, and controlled merging, while keeping the acting model fixed.

- **Graph-guided task selection.** We address the task-selection challenge in RSI with a semantic task-similarity graph that jointly selects diverse failure seeds and successful tasks as protected tasks, balancing local repairs with the preservation of overall task performance under a limited evaluation budget.

- **Task performance and generalization.** Our experiments show that the harnesses evolved by SCREENRSI improve task performance with a fixed acting model and generalize to unseen test tasks. On MobileGym, the best evolved harness reaches a test success rate of 74.77%, a gain of 15.7% over the initial harness.

2 Related Work

2.1 Harness Optimization and Recursive Self-Improvement

Recent work treats the agent harness as an explicit optimization target. Existing approaches differ in both optimization scope and update process. Some optimize the complete harness or its prompt–tool middleware (Lee et al., 2026c; Zhao et al., 2026). Others use execution feedback to diagnose and repair concrete harness flaws (Lin et al., 2026; Chen et al., 2026). Related methods change narrower runtime mechanisms through inference-time patches or generated tool actions (Sun et al., 2026; Lee et al., 2026b). Beyond individual revisions, harness evolution can guide candidate quality across versions or recursively update the harness and its improvement process (Zhang et al., 2026b; Jiang et al., 2026; Zhang et al., 2026a; Lee et al., 2026a; Zhou, 2026). These studies provide different mechanisms for revising agent harnesses from execution evidence. Our work focuses on how to allocate expensive GUI evaluation across repair and capability preservation. ScreenRSI uses a semantic task graph to select diverse failures together with nearby and distant successful tasks, then admits isolated harness revisions only after targeted validation.

2.2 GUI Agents and Experience Reuse

GUI agents can improve from interaction experience without updating model weights. Existing methods differ in how they represent and reuse this experience. Some maintain persistent memories of task knowledge and prior interactions (Mi et al., 2026; Xiao et al., 2026), while others distill trajectories into reusable skills, shortcuts, or higher-level action abstractions (Wang et al., 2025; Jiang et al., 2025; Hu et al., 2026). Experience learning can also be coupled with generated environments to expand the available interaction data (He et al., 2026). These approaches are complementary to harness evolution because an evolved harness may itself implement memory, skill, or prompt-based mechanisms. Our direct optimization target is the editable shared harness, which controls prompt construction, output parsing, state management, and interaction flow. The distinction therefore lies in how these mechanisms are produced and evaluated rather than in the form they eventually take.

2.3 Evaluation and Cross-Task Transfer

Evaluating harness evolution requires separating the possible sources of an observed gain. Improvements may reflect cross-task transfer, repeated adaptation to the same task pool, or additional opportunities to evaluate candidates. Final success rate alone does not distinguish these effects. Under matched feedback and inference budgets on Terminal-Bench 2.1, harness evolution does not consistently outperform task-level test-time scaling (Wang et al., 2026). HarnessOpt-Bench uses controlled optimization budgets and explicit development, validation, and test partitions to separate candidate optimization from final measurement (Ursekar et al., 2026). These studies motivate treating within-set adaptation, transfer beyond the optimization tasks, and the value of an optimization policy as distinct empirical claims. In our study, harness revisions use trajectories only from MobileGym training tasks, while test tasks provide no optimizer feedback. This separation allows us to examine whether revisions derived from local training evidence transfer beyond the tasks that produced them.

3 Framework

3.1 Problem Formulation

We consider a GUI agent comprising an acting model A, an interactive environment $\mathcal{E}$ containing a task suite $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, and an editable harness h_r at improvement round r. At each step, the agent observes the current GUI only through a screenshot and a step index, without access to the environment state or task outcome. Together, A and h_r induce the GUI agent policy π_{A,h_r}. The interaction between π_{A,h_r} and $\mathcal{E}$ produces a trajectory ξ_i^r and a task outcome $y_i^r \in \{0, 1\}$ for task τ_i. In one RSI round, the optimizer agent O analyzes one or more trajectories and revises h_r into h_{r+1}, aiming to improve task performance while A and $\mathcal{E}$ remain fixed.

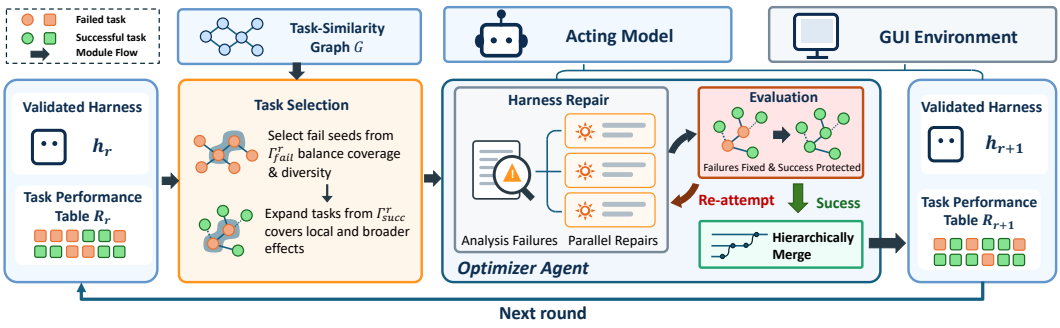


Figure 2: Overview of SCREENRSI. With the acting model and environment held fixed, a semantic task graph allocates failure-analysis and regression-testing effort; isolated harness revisions are promoted only after graph-guided validation.

Harness Editing Space We start from a minimal harness with three editable code files, which together define the Harness Editing Space of SCREENRSI. We represent these components as $h = (h^{\text{agent}}, h^{\text{parser}}, h^{\text{prompt}})$. The `agent.py` component builds messages for the acting model and controls the interaction flow; `parser.py` parses the acting model's outputs; and `prompt.py` configures the system prompt. The optimizer agent is permitted to modify only these three files; it cannot modify the environment or evaluation system.

Evaluation Objective. Before RSI, we evaluate every task to initialize a task performance table $\mathcal{R}_0 = \{(\tau_i, \hat{y}_i^0)\}_{i=1}^N$, where $\hat{y}_i^0 \in \{0, 1\}$ is the observed outcome. Before round r, SCREENRSI maintains $\mathcal{R}_r = \{(\tau_i, \hat{y}_i^r)\}_{i=1}^N$ using the latest available outcome for each task. It partitions the suite into $\mathcal{T}^r_{\text{succ}} = \{\tau_i : \hat{y}_i^r = 1\}$ and $\mathcal{T}^r_{\text{fail}} = \{\tau_i : \hat{y}_i^r = 0\}$, so $\mathcal{T} = \mathcal{T}^r_{\text{succ}} \cup \mathcal{T}^r_{\text{fail}}$. For a harness h, let $s(h, \tau_i)$ be the expected success on task τ_i. Our objective is to increase the suite-wide success rate $J(h) = \frac{1}{N} \sum_{i=1}^N s(h, \tau_i)$.

3.2 SEMANTIC TASK-SIMILARITY GRAPH

Before RSI begins, SCREENRSI constructs a Semantic Task-Similarity Graph $G = (V, E, W)$ from task specifications. Node v_i represents task τ_i, and $w_{ij} = \text{sim}(\tau_i, \tau_j) \in [0, 1]$ measures semantic relatedness. Scores are computed once with a fixed rubric and sparsified to retain a bounded neighborhood per task; downstream selection uses the maximum retained score across both directions of a pair. The graph remains fixed during RSI, providing a stable prior for task selection. Appendix A details its construction and validation.

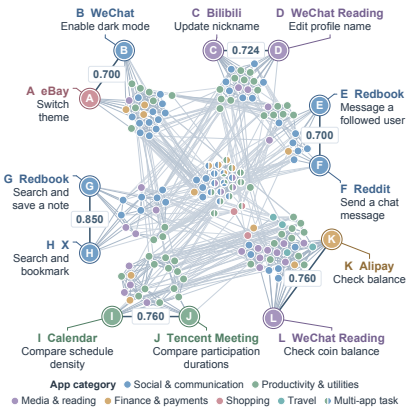


Figure 3: Cross-application structure in the static Semantic Task-Similarity Graph.

3.3 RECURSIVE IMPROVEMENT CYCLE

Figure 2 summarizes the transition from h_r to h_{r+1}. In each round, SCREENRSI selects tasks using the graph, and the optimizer agent analyzes their trajectories, proposes harness revisions, and merges validated repairs. Appendix H provides the optimizer agent's prompts.

Task selection. Because the task suite may be too large to examine in every round, the controller selects a task scope in two steps. First, it chooses K seed failures from $\mathcal{T}^r_{\text{fail}}$. For a candidate seed set $\mathcal{T}^r_{\text{seed}}$, coverage $C(\mathcal{T}^r_{\text{seed}}; \mathcal{T}^r_{\text{fail}})$ measures how well it represents the remaining failures, while

diversity $D(\mathcal{T}^r_{\text{seed}})$ discourages redundant seeds:

$$C(\mathcal{T}^r_{\text{seed}}; \mathcal{T}^r_{\text{fail}}) = \frac{1}{|\mathcal{T}^r_{\text{fail}}|} \sum_{\tau_j \in \mathcal{T}^r_{\text{fail}}} \max_{\tau_i \in \mathcal{T}^r_{\text{seed}}} w_{ij}, \quad D(\mathcal{T}^r_{\text{seed}}) = \text{mean}_{\tau_i \neq \tau_j \in \mathcal{T}^r_{\text{seed}}}(1 - w_{ij}), \tag{1}$$

We set $D(\mathcal{T}^r_{\text{seed}}) = 1$ for a singleton. The controller scores candidate sets by $\lambda C(\mathcal{T}^r_{\text{seed}}; \mathcal{T}^r_{\text{fail}}) + (1 - \lambda) D(\mathcal{T}^r_{\text{seed}})$ and samples seeds sequentially without replacement using softmax, yielding the selected $\mathcal{T}^r_{\text{seed}}$.

Second, for each seed, we select two tasks as protected tasks from $\mathcal{T}^r_{\text{succ}}$ using similarity-weighted softmax: one semantically close task and one semantically distant task. Both are used to check for both local and broader effects of a repair. We re-evaluate them on h_r; only tasks that succeed enter $\mathcal{T}^r_{\text{protect}}$. The round's candidate task set is $\mathcal{T}^r_{\text{cand}} = \mathcal{T}^r_{\text{seed}} \cup \mathcal{T}^r_{\text{protect}}$. Its evaluations provide the trajectories and outcomes used for repair.

Harness repair. The optimizer groups failed trajectories into actionable repair directions and explores each in an isolated harness branch. It tests each revision on its target failures and tasks that have succeeded in $\mathcal{T}^r_{\text{protect}}$, revising within a fixed attempt budget. Only revisions that fix their targets and preserve success on these tasks proceed to merge.

Evaluation and Merge. Only candidate repairs that succeed on their target failures and preserve success on all tasks in $\mathcal{T}^r_{\text{protect}}$ are eligible for merge. The controller merges eligible candidates hierarchically, then evaluates the combined harness on the full candidate task set $\mathcal{T}^r_{\text{cand}}$. If any task fails, the controller attempts to repair the combined harness using the new failure trajectories and re-evaluates the full set, for up to three merge-repair attempts. The combined harness becomes h_{r+1} only when every task in $\mathcal{T}^r_{\text{cand}}$ passes; if failures remain after three attempts, the round retains h_r. Observed outcomes update $\mathcal{R}_{r+1}$, with periodic suite-wide evaluations refreshing it. RSI stops when the round budget is exhausted or no failures remain to repair.

Thus, each round uses graph-guided evidence to produce a validated harness revision and an updated task state for the next round.

4 EXPERIMENTS

Our experiments mainly address four key questions:

- **RQ1:** Does SCREENRSI improve task performance and generalize to test tasks?
- **RQ2:** Does graph-guided task selection outperform random selection?
- **RQ3:** How does the choice of optimizer agent affect harness improvement?
- **RQ4:** What harness mechanisms emerge during RSI?

4.1 EXPERIMENTAL SETUP

We use MobileGym (Wu et al., 2026) as the experimental environment. All RSI optimization is confined to the training split, and we evaluate evolved harnesses on the test split. The test split contains 256 tasks across four difficulty strata, L1–L4. We follow the MobileGym evaluator and report overall success rate (SR), progress rate (PR), and success rate in difficulty stratas. The false completion rate (FC) measures declarations of completion without task satisfaction, and unexpected side effect rate (USE). The acting model is Qwen3.8-27B. The optimizer agent performs analysis, repair, and merge works using DeepSeek-V4.1-Flash, GPT-5.6-Terra, or GPT-5.6-Sol as its backbone model.

4.2 RQ1: OVERALL EFFECTIVENESS AND GENERALIZATION

Test performance. Table 1 summarizes the train and test split results. Both GPT-optimized harnesses improve held-out task success. Terra raises SR from 59.06% to 65.57% (+6.51) and PR from 70.77% to 73.96% (+3.19), while reducing FC from 25.78% to 20.70%. Its SR gains span all four difficulty levels, with a 0.39% increase in USE. Sol achieves the highest SR, raising it to 74.77% (+15.71). Its gains span all difficulty strata. Sol-optimizered harness improves completion and reduces false completions and side effects.

Table 1: MobileGym performance (%). MobileGym test-set scores are reference from MobileGym leaderboard. For our method, DeepSeek-V4.1-Flash, GPT-5.6-Terra and Sol denote the optimizer agent backbone model, while the acting Qwen3.8-27B model is fixed. Bold identifies the best result among the harnesses within each split. ↑: higher is better; ↓: lower is better.

Split	models / optimizer	Best Round	SR↑	PR↑	L1↑	L2↑	L3↑	L4↑	FC↓	USE↓
MobileGym test-set scores by different acting models										
Test	Gemini 3.1 Pro	–	58.8±1.4	72.1	97.5	83.6	63.3	21.9	34.0	5.5
Test	Doubao-Seed-2.0-Pro	–	52.0	63.6	100.0	93.2	48.2	6.2	33.6	4.7
Test	Qwen3.6-Plus	–	45.7	59.2	100.0	78.1	44.6	3.8	34.0	14.5
Test	AutoGLM-Phone-9B	–	20.0±1.3	35.3	86.2	33.6	9.6	1.9	39.6	12.6
Test	UI-Venus-1.5-8B	–	15.4±2.4	28.3	85.0	21.9	6.0	1.9	22.9	7.7
Test	GUI-Owl-1.5-8B-Think	–	15.1±0.9	28.8	76.2	26.0	4.2	1.2	30.4	14.1
Test	UI-TARS-1.5-8B	–	13.8±1.7	26.3	77.5	21.9	3.0	1.6	38.6	11.0
Test	Step-GUI-4B	–	12.9±1.1	25.7	83.8	17.8	2.4	1.6	37.0	7.6
Test	Qwen3-VL-4B	–	9.4±0.6	20.1	71.2	12.3	0.6	0.3	15.9	10.0
Ours										
Train	Initial harness	0	88.75	**95.31**	90.00	93.33	**87.04**	80.77	3.75	1.25
Train	DeepSeek-V4.1-Flash	19	88.61	89.06	**100.00**	93.22	79.24	88.46	**3.12**	1.25
Train	GPT-5.6-Terra	18	91.44	94.07	**100.00**	94.82	84.61	**91.67**	5.92	1.31
Train	GPT-5.6-Sol	6	**92.71**	92.98	94.44	**100.0**	84.61	**91.67**	4.60	**0.65**
Test	Initial harness	0	59.06	70.77	95.00	78.08	54.88	36.71	25.78	5.86
Test	DeepSeek-V4.1-Flash	19	57.53	67.53	**100.00**	76.71	56.79	29.49	23.82	7.03
Test	GPT-5.6-Terra	18	65.57	**73.96**	**100.00**	84.28	64.63	38.89	20.70	6.25
Test	GPT-5.6-Sol	5	**74.77**	69.32	**100.00**	**89.55**	**70.42**	**53.57**	**15.23**	**4.29**

Train–test generalization. Generalization is essential to RSI: useful repairs must improve performance on unseen tasks. We find that although HRSI bring modest gains on the training split, the improvements make harnesses more adaptable to new tasks and more robust to errors, leading to substantial gains on the unseen test split. Both GPT-optimized harnesses improve SR on the held-out split by more than on the training split. Terra increases train SR by 2.69% and test SR by 6.51%. Sol increases train SR by 3.96% and test SR by 15.71%. These results are consistent with repairs that transfer beyond the training tasks.

Task Execution Steps. A well-designed harness can reduce the number of steps needed to complete a task. Figure 5a reports mean task execution steps on the test set. At its best SR round, Sol reduces the mean from 19.71 steps with the initial harness to 14.35 in round 5. Terra reaches its lowest mean of 17.1 steps in rounds 6 and 15. Both GPT-optimized harnesses reduce the number of execution steps.

Harness editing directions. Figure 5b shows how the optimizer agents use the Harness Editing Space. Starting from 168 code lines, the final harnesses contain 607 lines for Sol, 301 for Terra, and 884 for DeepSeek-V4.1-Flash. Sol expands both the agent and prompt components, 300 and 163 lines, while Terra makes a smaller revision 148 and 23 lines in these components. DeepSeek-V4.1-Flash produces the largest harness, with most of its code in the agent component, yet its test SR remains below that of the initial harness. These results show that optimizer agents choose different editing directions. The large amount of code added by DeepSeek-V4.1-Flash makes the harness harder to maintain and does not ensure better task performance.

4.3 RQ2: Contribution of Graph-Guided Task Selection

Graph-guided task selection improves test-set task completion and reduces false completions compared with random selection. Table 2 compares the strategies under the same recursive round budget: graph-guided selection raises test SR from 69.04% to 74.77% and lowers FC from 18.75% to 15.23% relative to random selection. Figure 4 traces test SR across optimization rounds for both strategies.

Table 2: Task-selection strategy ablation on the MobileGym test split. Graph-guided and random selection use the same recursive round budget for each optimizer.

Optimizer	Strategy	SR↑	PR↑	FC↓	USE↓
GPT-5.6-Sol	Random	69.04	77.30	18.75	4.29
	Graph	74.77	69.32	15.23	4.29
GPT-5.6-Terra	Random	60.80	71.80	27.73	6.64
	Graph	65.57	73.96	20.70	6.25

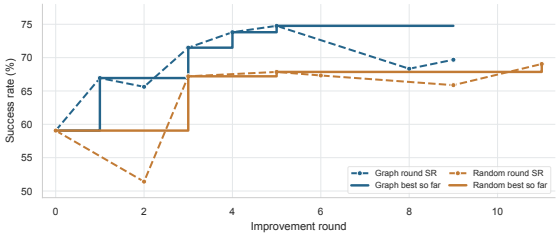
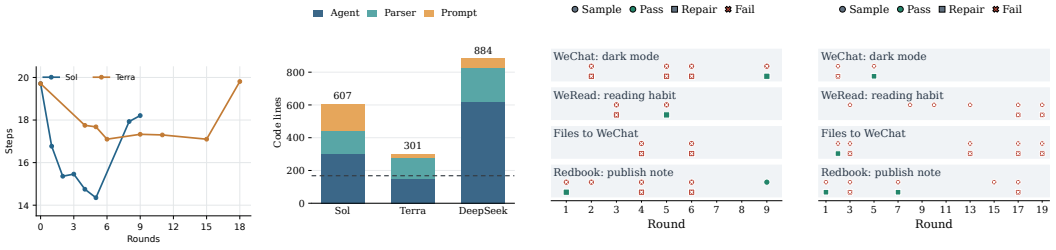


Figure 4: Test-set success rate across rounds. Dashed lines connect observations; solid lines mark the best rate.



(a) Test-set execution steps. (b) Final harness code lines. (c) Sol repair history. (d) Terra repair history.

Figure 5: (a) Mean test-set execution steps by round for Sol and Terra, starting at step 0 with the initial harness. (b) Final harness code lines by component; the dashed line marks the initial harness size of 168 lines. (c,d) Repair histories for four Sol and Terra tasks. Circles show sampled evaluations and squares show branch outcomes; green indicates a pass, red a failure, and blank entries an unrecorded outcome. A passing branch is not necessarily merged.

Task Repair History. Figures 5c and 5d trace four tasks across Sol and Terra optimization rounds; Appendix Figure 10 extends the histories to all tasks. Sol's dark-mode repair fails in rounds 2, 5, and 6 before passing in round 9, and its reading-habit repair fails in round 3 before passing in round 5. Terra's file-transfer repair passes in round 2, but the task fails in later sampled evaluations and repair branches. This suggests that harness changes can affect performance across tasks, causing tasks that previously passed to fail in later evaluations.

4.4 RQ3: Choice of RSI Optimizer Agent

The choice of backbone model matters for the optimizer agent. Table 1 shows that, with the acting model and graph-guided workflow fixed, Sol reaches its best test SR of 74.77% at round 5, and Terra reaches 65.57% at round 18. DeepSeek-V4.1-Flash exhausts its round budget with a best test SR of 57.53%, below that of the initial harness. HRSI is a black-box optimization problem: the optimizer agent must identify promising directions for improvement and implement effective code repairs.

Appendix Figure 11 shows the Git revision histories of the three optimizer agents. Under ScreenRSI, a merged harness is accepted only if it passes every task in the candidate set. Sol's proposed revisions are consistently accepted, while a few of Terra's revisions and more of DeepSeek-V4.1-Flash's attempted rounds fail the merge criterion. These results suggest that HRSI remains challenging even for frontier models.

4.5 RQ4: Mechanisms Emerge During RSI

By analyzing the code changes made by Sol and Terra, we identify four main improvements introduced by RSI. Figure 5b compares the final harness sizes with the initial harness. First, **state management**: The process equips the harness with a compact memory that carries task-relevant evidence forward as the interaction proceeds, reducing the risk that an earlier observation is forgotten or assigned to the wrong source. Second, **stuck monitoring and recovery**: the harness

compares the interface before and after an action and initiates recovery when an action produces no meaningful visual change or the trajectory enters a short cycle, thereby limiting repeated ineffective behavior. Third, **task-related prompts**: RSI adds task-specific prompts to the harness. These prompts require the acting agent to find interface evidence or controls that match each task condition, preventing it from incorrectly concluding that the task is complete. Fourth, **format validation**: RSI adds substantial format-validation code to the harness, improving its robustness. Together, they span state management, harness control flow, and task-specific details, offering useful guidance for the development of more robust harnesses.

4.6 Case Study

Two examples illustrate how harness revisions change task interpretation and GUI control. Additional case studies appear in Appendix E.

Checking temporal scope. In Figure 6, the task asked for the longest reading day over **the last week**, the agent uses WeRead's calendar-week summary and incorrectly submits Wednesday (172 minutes). The repair makes the requested interval explicit: September 17–23. Guided by this constraint, the agent checks both overlapping calendar weeks and correctly submits Friday (226 minutes); the case shows how a prompt-level scope check can prevent a correctly read statistic from answering the wrong question.



Figure 6: Reading-day task. The failed run answers from the Sep. 21–27 calendar week and submits Wednesday (172 minutes). The revised harness states that "last week" means Sep. 17–23 and asks the agent to check source date ranges. The successful run taps the previous-week arrow at step 8, compares the two overlapping weeks, and submits Friday (226 minutes) from the Sep. 14–20 view at step 9. All three screenshots use the same crop and scale.

Escaping a time-picker loop. The agent reads the earliest timed event on tomorrow's calendar at 09:30 and needs to set an alarm for 09:00. In the failed trajectory, it repeatedly overshoots the hour wheel between 08 and 10, exhausting the 45-step budget without saving an alarm. The successful trajectory receives runtime prompts that flag a two-step loop and an ineffective click. It then uses a shorter swipe, reaches 09:00, and saves the alarm (Figure 7). The system prompt is the same in both trajectories; the contrast concerns the additional feedback delivered during execution.

5 Discussion

Local repair but global effects. A single trajectory exposes a narrow failure mode, while harness code affects performance across the entire benchmark. Our semantic graph reduces this mismatch by

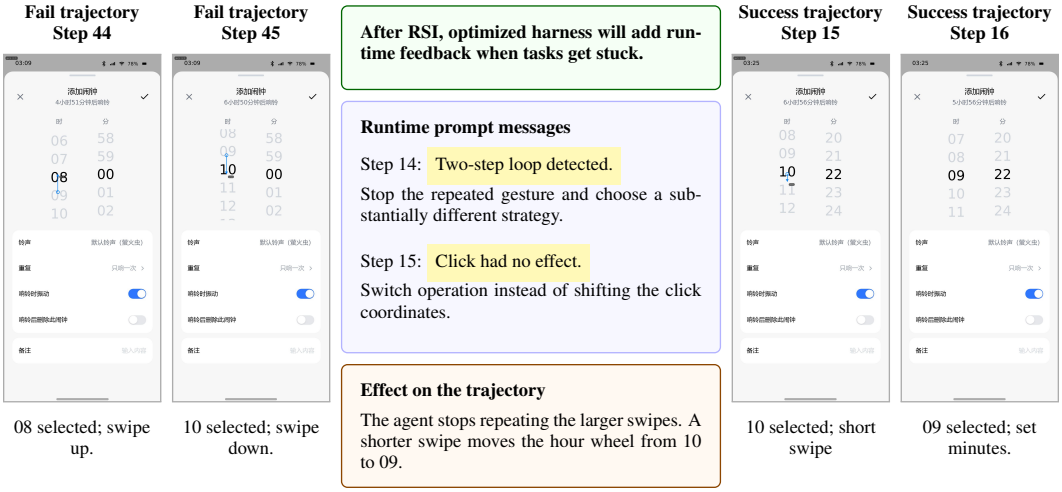


Figure 7: Calendar-to-alarm task. The agent needs to set the alarm to 09:00. In the failed run, the hour wheel alternates between 08 and 10 at steps 44–45. In the successful run, a shorter swipe at step 15 selects 09 at step 16. The center box translates the runtime prompts that flag repeated or ineffective actions.

adding related successful tasks to the repair context and by distributing repair effort across distinct failure regions.

What Next for HRSI? We find that optimizer agents tend to favor prompt edits because they are easy to propose, inexpensive, and broad in scope. The resulting harness mechanisms point toward capabilities that the acting model can acquire directly: task-related experience, awareness of stalled progress, and proactive recording of task evidence. This provides a path to simplify the harness in future HRSI rounds.

The gap among Sol, Terra, and DeepSeek shows that the choice of optimizer is critical to HRSI. The graph controls task selection, but the optimizer still determines whether failures are diagnosed correctly and whether proposed edits generalize beyond their targets. For frontier models, HRSI can be viewed as a black-box optimization problem that provides training signals at each iteration. Thus, HRSI itself offers substantial potential to improve both the acting model and the optimizer agent.

6 Conclusion

We introduced ScreenRSI, a framework for recursive self-improvement of GUI-agent harnesses with the acting model and environment held fixed. It addresses the gap between local failure evidence and the broad effects of shared harness code. A semantic task-similarity graph guides the selection of diverse failures and previously successful tasks, directing a limited evaluation budget toward both repair and preservation. Trajectory analysis, isolated revisions, and controlled merging turn this evidence into successive validated harness updates.

Our experiments on MobileGym show that harness repairs learned from training tasks can transfer to unseen tasks. The evolved harnesses reveal concrete mechanisms for stronger GUI agents: retaining task evidence, detecting and recovering from stalled interaction, grounding task completion in interface evidence, and validating model outputs. These findings establish the harness as a practical target for recursive self-improvement and provide a path toward agents that systematically improve how they use their models. Learning these mechanisms directly in the acting model and strengthening the optimizer's repair ability are promising next steps.

AI Use Statement

We used generative AI tools to assist with code development, the recursive self-improvement (RSI) experiments, statistical analysis of experimental results, figure preparation, literature retrieval and

discovery of related work, manuscript drafting, grammar checking, and language polishing. The reported experimental results, including the numerical data presented in tables and figures, were obtained from actual experiments and statistical analysis, rather than directly generated by AI. The scientific views and conclusions presented in this paper were proposed and confirmed by the human authors. The authors take responsibility for the final content of the paper, including all AI-assisted text, code, analyses, and figures.

Ethics Statement

To the best of our knowledge, this research does not raise significant ethical concerns. The work is conducted for research purposes and evaluates methods for improving GUI-agent harnesses in a benchmark setting. We have considered the potential implications of automated agent behavior and do not identify any ethical issues specific to the experiments reported in this paper. As with other research on autonomous agents, future applications should follow applicable rules and use appropriate human oversight. The authors take responsibility for the conduct and reporting of this study.

Reproducibility Statement

We aim to make the findings of this study reproducible. The paper reports the experimental setting, evaluation procedure, and metrics used to assess harness performance. We will release the implementation code and RSI trajectories so that other researchers can inspect the optimization process and repeat the main experiments. The released trajectories will also provide a record of the evidence and outcomes observed during successive improvement rounds. Together, the paper and these materials are intended to support independent examination and reproduction of the reported results.

References

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From failed trajectories to reliable llm agents: Diagnosing and repairing harness flaws, 2026. URL https://arxiv.org/abs/2606.06324.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. A comprehensive survey of self-evolving AI agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025. URL https://arxiv.org/abs/2508.07407.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenghailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. URL https://arxiv.org/abs/2507.21046.

Tiantian He, Yihang Chen, Keyue Jiang, Ka Yiu Lee, Kaiwen Zhou, Kun Shao, and Shuai Wang. Ee-mcp: Self-evolving mcp-gui agents via automated environment generation and experience learning, 2026. URL https://arxiv.org/abs/2604.09815.

Longtao Hu, Xiao Liang, and Linchao Zhu. From interaction traces to persistent skills: Online evolution for computer-use agents, 2026. URL https://arxiv.org/abs/2609.04869.

Wen Jiang, Mingmin Chu, Yimeng Tian, Qianxin Zhang, Haofei Yang, Rui Yang, Yang Liu, Tao Lv, and Fangming Li. Harnessevolve: Learning from reference trajectories for reliable agent self-evolution, 2026. URL https://arxiv.org/abs/2609.00829.

Wenjia Jiang, Yangyang Zhuang, Chenxi Song, Xu Yang, Joey Tianyi Zhou, and Chi Zhang. Appagentx: Evolving gui agents as proficient smartphone users, 2025. URL https://arxiv.org/abs/2503.02268.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement, 2026a. URL https://arxiv.org/abs/2607.15524.

Juyong Lee, Woogyeol Jin, and Kimin Lee. Improving proficiency and efficiency of android gui agents via self-generating tool actions, 2026b. URL https://arxiv.org/abs/2609.06792.

Yoonho Lee, Roshen Sanjay Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling*, 2026c. URL https://openreview.net/forum?id=tmbOUyFx3R.

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, 2026. URL https://arxiv.org/abs/2604.25850.

Hongze Mi, Yibo Feng, WenJie Lu, Song Cao, Jinyuan Li, Yanming Li, Xuelin Zhang, Haotian Luo, Songyang Peng, He Cui, Tengfei Tian, Jun Fang, Hua Chai, and Naiqiang Tan. Darwinian memory: A training-free self-regulating memory system for GUI agent evolution. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=tNn9bikyXZ.

Xueqiao Sun, Xiaohan Wang, Ludwig Schmidt, Serena Yeung-Levy, and Yuhui Zhang. Learning from failure: Inference-time self-improvement for computer-use agents. In Paolo Favaro, Zuzana Kukelova, Atsuto Maki, Anna Rohrbach, Konrad Schindler, and Federico Tombari (eds.), *Computer Vision – ECCV 2026*, pp. 210–227, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-37577-3.

Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S. Kalmath, Veronica Chatrath, and Yuan Xue. Harnessopt-bench: Evaluating llms at harness optimization, 2026. URL https://arxiv.org/abs/2608.06301.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. In *COLM 2026 The 2nd Workshop on Lifelong Agents: Learning, Aligning, and Evolving*, 2026. URL https://openreview.net/forum?id=WVAeeSlVim.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-agent-e: Self-evolving mobile assistant for complex tasks, 2025. URL https://arxiv.org/abs/2501.11733.

Dingbang Wu, Rui Hao, Haiyang Wang, Shuzhe Wu, Han Xiao, Zhenghong Li, Bojiang Zhou, Zheng Ju, Zichen Liu, Lue Fan, and Zhaoxiang Zhang. Mobilegym: A verifiable and highly parallel simulation platform for mobile GUI agent research, 2026. URL https://arxiv.org/abs/2605.26114.

Han Xiao, Guozhi Wang, Hao Wang, Shilong Liu, Yuxiang Chai, Yue Pan, Yufeng Zhou, Xiaoxin Chen, Yafei Wen, and Hongsheng Li. Ui-mem: Self-evolving experience memory for online reinforcement learning in mobile gui agents, 2026. URL https://arxiv.org/abs/2602.05832.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Large language model-brained GUI agents: A survey. *Transactions on Machine Learning Research*, 2025. URL https://arxiv.org/abs/2411.18279.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves, 2026a. URL https://arxiv.org/abs/2606.09498.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. Harnesscompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses, 2026b. URL https://arxiv.org/abs/2608.01918.

Cen Mia Zhao, Haibo Ruan, Wenjie Chen, Pei fen Tu, Usman Abbasi, and Joel Hesch. Beyond prompts: Measuring and optimizing llm tool-agent harnesses, 2026. URL https://arxiv.org/abs/2609.05736.

Tailin Zhou. Hierarchical self-improvement: A framework for task-specific evolvable agent harnesses, 2026. URL https://arxiv.org/abs/2608.08466.

A Semantic Task-Similarity Graph

Before recursive improvement begins, we construct a fixed weighted graph from the task specifications alone. Each benchmark task $\tau_i \in \mathcal{T}$ is represented as a node. For every pair of tasks $\tau_i, \tau_j \in \mathcal{T}$, semantic relatedness is assessed from behavioral structure rather than application identity alone. The assessment considers the task objective, required actions, interaction pattern, exercised capabilities, composition, and the type of information being located, transformed, compared, or modified. The resulting score $w_{ij} = \mathrm{sim}(\tau_i, \tau_j) \in [0, 1]$ represents the directed semantic similarity from task τ_i to task τ_j.

The graph is sparsified independently for each source task τ_i by retaining only its eight highest-scoring neighbors. Hence, the reported graph has 160 nodes and 1,280 directed neighbor entries, and an edge need not be reciprocal. For downstream task selection, the similarity of a task pair (τ_i, τ_j) is the maximum retained score in either direction (with a missing edge assigned score zero). The graph is constructed once and held fixed throughout RSI. It therefore provides a reproducible semantic prior for failure coverage and preservation-task selection without incorporating outcomes from the improvement process.

B Evaluation Protocol

Each task is evaluated from its prescribed reset state under a fixed maximum interaction budget, timeout policy, action vocabulary, and benchmark evaluator. Baseline and periodic train evaluations follow the same scoring protocol. Candidate-level comparisons bind parent and child harnesses to the same task instances and evaluator conditions; invalid initial states and execution errors are treated as unknown rather than task failures. Fixed seeds are reused where the environment permits deterministic execution.

C Fixed Agent Configuration

The acting agent uses Qwen3.8-27B throughout optimization and evaluation. Its checkpoint, serving stack, numerical precision, context window, decoding parameters, token budget, image resolution, and action vocabulary are held fixed across harness revisions, so measured changes arise from the harness and evaluation variability rather than weight updates.

D Initial Harness

The initial harness defines the system prompt, screen-to-context transformation, output schema, parser, retry behavior, termination rule, and bounded interaction history. It contains no task-specific repair state. All reported variants descend from this common source revision, which serves as the baseline in Table 1.

E Case Studies

Retaining values across applications. The agent reads monthly spending of 1041.08 in Alipay, but enters 1258.50 in Notes and sends 1256.78 through WeChat after the source observation leaves its two-response history. The repair retains numeric observations in a separate ledger and supplies them to later turns. Both destinations then contain 1041.08 (Figure 8), illustrating how explicit state retention preserves source values across application switches.

Selecting the calendar date before creating an event. The task asks for an event named "Business trip reminder" on October 5 with a reminder 30 minutes in advance. In the failed run, the agent opens the event form from the September calendar. The form defaults to September 24, and repeated clicks on its date field do not change the date before the 45-step budget ends. In the successful run, the agent first navigates to October, selects October 5, and only then opens the form. It changes the reminder from 15 to 30 minutes and saves the event (Figure 9). The revised system prompt also

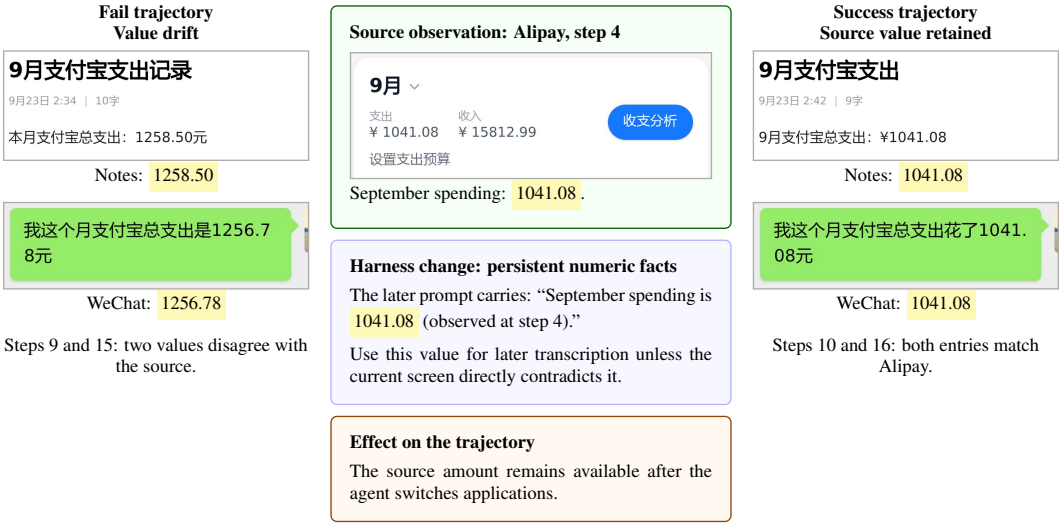


Figure 8: Cross-application spending task. The agent reads 1041.08 in Alipay, but the failed run enters 1258.50 in Notes and sends 1256.78 in WeChat after the source screen leaves its short history. The revised harness carries the observed amount in a persistent numeric-facts block. The successful run uses 1041.08 in both destinations. Screenshot text and the prompt message are shortened English translations.

specifies that a forced path-replanning signal requires leaving a blocked screen and using a different entry path; the successful trace follows a calendar-first route before such a signal is needed.

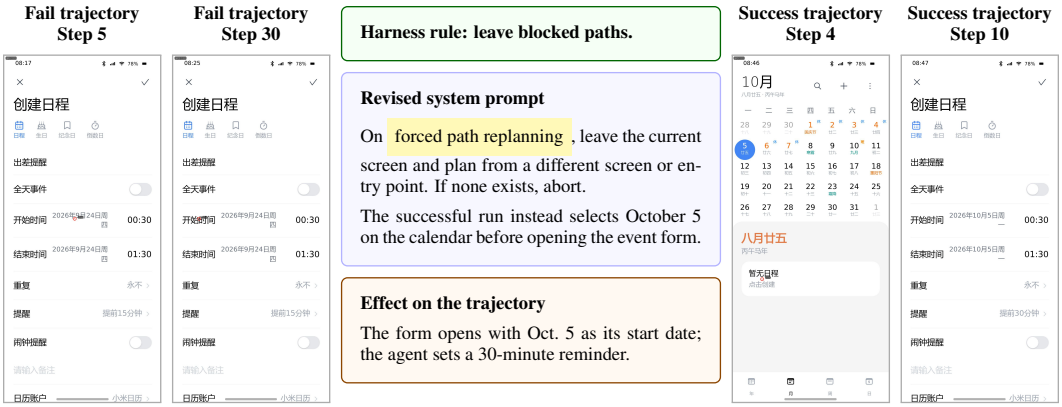
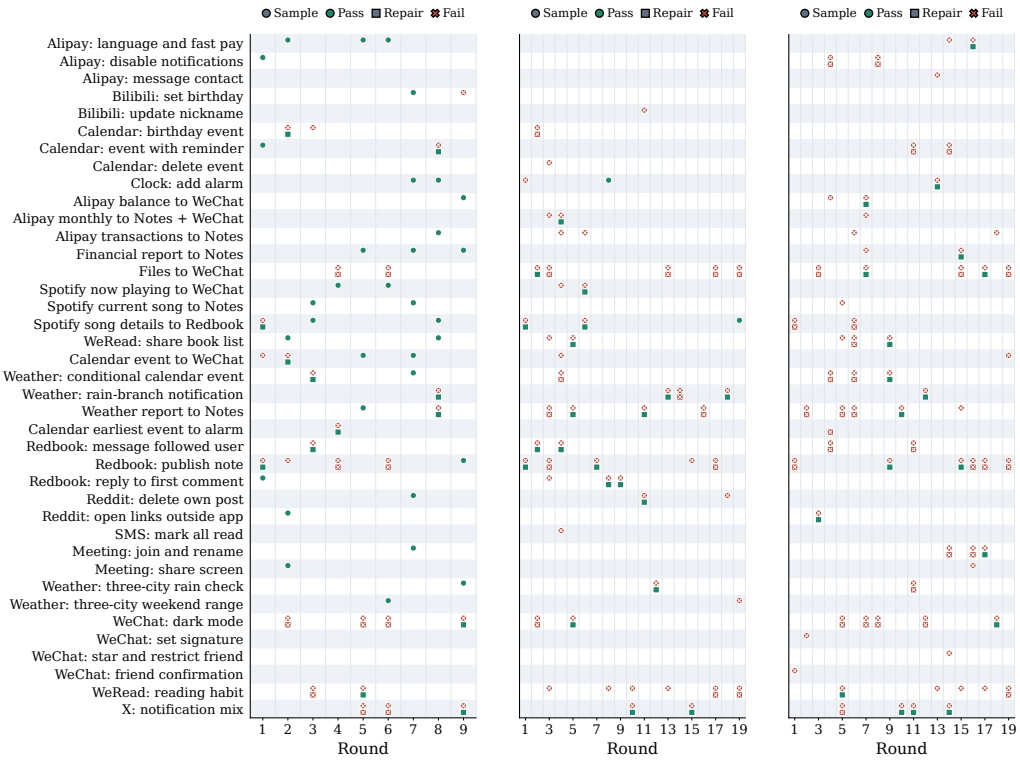


Figure 9: Calendar event task. The failed run enters the form with the wrong default date and repeatedly clicks its unresponsive date field. The successful run selects October 5 in the calendar before creating the event, then sets the requested 30-minute reminder. The center box summarizes a revised path-replanning rule and the observed route.

F Task Selection and Repair Histories

Figure 10 expands the selected examples in Figure 5 to all 39 tasks with at least one documented sampling failure across Sol, Terra, and DeepSeek. The three panels share the same task order, with task names shown only on the left. Known sampled successes are retained for these tasks. Sol's sampling records cover all completed rounds; Terra's summary identifies only two task-specific sampled successes, and DeepSeek's summary does not enumerate sampled successes. Missing entries therefore cannot be interpreted as successes or as evidence that a task was not selected.

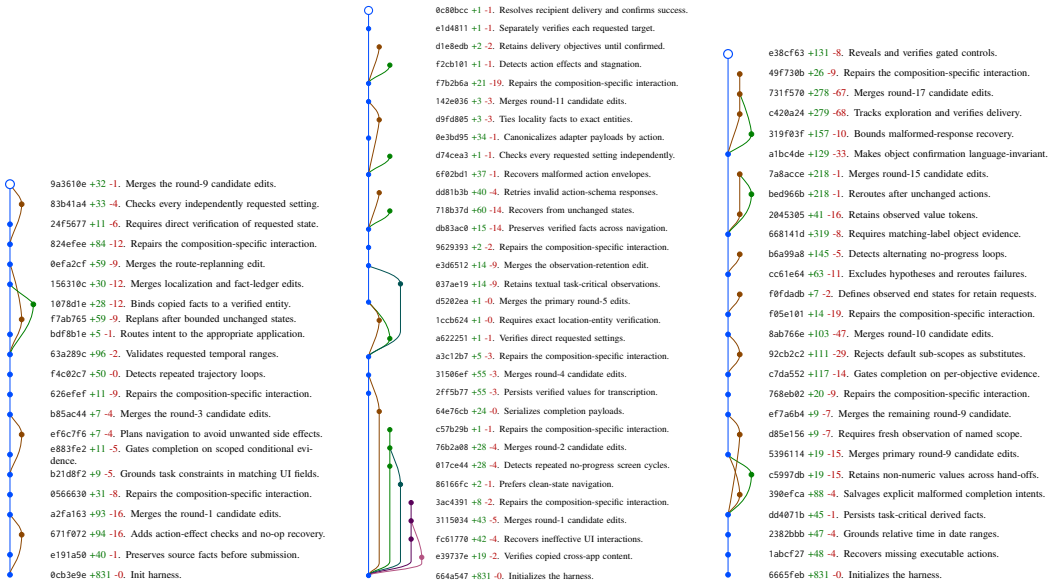


(a) Sol. (b) Terra. (c) DeepSeek.

Figure 10: Task selection and repair histories for tasks with observed failures. Rows are tasks in a shared order; columns are optimization rounds, with each panel spanning its own completed rounds. Within each cell, the upper circle shows a sampled outcome and the lower square shows the final repair-branch panel outcome for a causal task. Green denotes a pass; red with a cross denotes failure (including repair timeouts). A branch's panel result is repeated for each causal task: a failed panel does not mean every causal task failed, and a passing panel does not imply mainline acceptance. Blank cells are undocumented; sampled successes are incomplete for Terra and unavailable for DeepSeek. DeepSeek's round-4 execution error is omitted from sampled outcomes. Incomplete rounds (Sol 10, Terra 20, and DeepSeek 20) are excluded.

G Optimizer Revision Histories

Figure 11 records the commit-level revision history for the Sol optimizer. The two adjacent panels reserve space for histories from other optimizer runs.



(a) Optimizer agent using GPT-5.6-Sol.

(b) Optimizer agent using GPT-5.6-Terra.

(c) Optimizer agent using DeepSeek V4.1 Flash.

Figure 11: Revision histories for optimizer runs.

H Optimizer Agent Prompts

The analysis, improvement, and merge agents operate under separate prompts that delimit their tools, editable files, output schema, validation obligations, and stopping conditions. The analysis agent identifies the earliest harness-addressable error in a failed trajectory; the improvement agent implements and tests an isolated repair; and the merge agent composes revisions that pass validation while preserving their tested task scope.

The following are the complete, versioned prompt templates and agent-facing handoff instructions constructed by the implementation. Values enclosed in angle brackets are representative runtime substitutions; all proposal schemas are rendered directly from the corresponding current Pydantic models.

H.1 Task selection.

Prompt 1: Historical Target Selector

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are a target−selection−only worker in a disposable ‘/workspace‘ sandbox projection. Do not edit source files, tests, task files, evaluator configuration, benchmark inputs, result files, or any other workspace content. The controller owns all source mutation, Git operations, evaluation, and durable records.

Evidence boundary

Use only the supplied read−only protocol evidence index and the historical results, errors, and trajectory paths explicitly listed in it below ‘/evidence/‘. Do not enumerate ‘/evidence‘, guess paths, inspect or use development, validation, sealed, external, or other task data. Do not invent outcomes, traces, paths, or causal facts absent from that evidence.

Evidence overview:

Read the root object of ‘/protocol/evidence.json‘.

Required completion protocol

Write one ‘TargetSelectionProposal‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is the single JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_json_schema(TargetSelectionProposal)}
‘‘‘

Analyse every controller−provided historical failure exactly once across ‘analyses‘.

Decide in this order:

1. Identify the best−supported ‘failure_origin‘ independently from repairability.
2. Choose ‘ATTEMPT_REPAIR‘ when at least one evidence−based, reasonable, actionable, generalizable , in−scope Harness repair hypothesis exists. Root−cause certainty and a previously verified effect are not required. When the evidence supports such a hypothesis and contains no explicit counterevidence, prefer attempting the repair over ending exploration.
3. Choose ‘OUT_OF_SCOPE‘ only when the evidence indicates that the required change is outside the approved Harness change boundary.
4. Choose ‘NO_GENERALIZABLE_FIX‘ only when explicit evidence supports both that the failure was a one−off agent behavior and that no reasonable general Harness modification exists.
5. Choose ‘NO_ACTIONABLE_HYPOTHESIS‘ only when no grounded, executable repair hypothesis can be proposed at all. Uncertainty about which plausible hypothesis is correct does not qualify.

Agent−behavior failures can still have general Harness repairs. Premature task termination, content−fidelity failures, and missing state verification are examples that may be addressed through prompt, context, state−verification, or articulation changes. Evaluator−or−task origin likewise does not prevent ‘ ATTEMPT_REPAIR‘ when an in−scope Harness hypothesis exists.

This historical stage is not given a workspace or a controller−approved change boundary. Do not infer ‘ OUT_OF_SCOPE‘ from missing boundary information. Use it only when the supplied evidence itself explicitly establishes that every reasonable repair requires a change outside the allowed Harness scope. If an actionable Harness hypothesis exists and its scope is not explicitly excluded, choose ‘ ATTEMPT_REPAIR‘.

‘expected_difficulty‘ is optional. Set ‘fault_category‘ exactly when the disposition is ‘ ATTEMPT_REPAIR‘. Select from 1 through ‘3‘ supplied failure IDs in ‘selected_task_ids‘; IDs must be non−empty and unique. Every object forbids extra fields. Do not write source code, make an edit or repair plan, or use unsupplied data. The proposal JSON document is distinct from the terse final stdout summary.

Prompt 2: Task-Similarity Graph Builder

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are a one−time graph−build−only worker. The ‘/workspace‘ projection is empty and read−only. You have no source access and no authority to run commands, inspect a workspace, edit files, inspect candidate code, create branches or commits, perform Git operations, execute evaluations, or trigger an outer loop. The controller owns all durable records and graph validation.

Catalog boundary

Use only the supplied catalog in ‘/protocol/evidence.json‘ (the root object). It is the complete and exclusive source for this graph. Do not use historical outcomes, trajectories, train/dev labels (including train labels and development labels), candidate code, source access, evaluator results, external task data, or facts inferred from unavailable material. Do not invent task IDs, metadata, or similarity evidence.

Similarity semantics

Similarity should primarily reflect how similar two tasks are in their actual behavioral structure, rather than whether they belong to the same app or domain.

Judge task similarity from the combination of:

− the sequence and type of actions required;
− the task objective and interaction pattern;
− the capabilities exercised;
− the composition / control−flow structure;
− the kind of information being located, transformed, compared, or modified;
− the abstract task type.

App identity and application domain are secondary contextual signals only. Sharing the same app does not by itself imply high task similarity, and tasks from different apps may be highly similar when they require the same behavioral pattern.

When comparing candidates, prioritize behavioral correspondence over app identity. In particular, a cross−app task with a closely matching action and workflow pattern should rank above a same−app task whose behavior is substantially different.

Do not infer similarity primarily from the app prefix in ‘task_id‘. Treat task IDs as identifiers rather than semantic evidence except for the task−specific meaning explicitly expressed by the catalog.

Consider candidates across the full catalog before selecting the final nearest neighbors; do not preferentially fill the neighbor list with tasks from the same app.

Required completion protocol

Write one ‘TaskSimilarityGraphBuildProposal‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is the single JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_json_schema(TaskSimilarityGraphBuildProposal)}
‘‘‘

Output one node for every supplied catalog task ID, and no other node. Node and neighbor order is not semantically significant: the controller canonicalizes nodes by ascending ‘task_id‘ and neighbors by descending ‘similarity‘, then ascending ‘task_id‘ for ties. For each node, provide at most ‘3‘ non−self, distinct neighbors from the supplied catalog. Similarities must be finite numbers from 0 through 1 inclusive. Output no catalog metadata beyond the schema: the controller preserves and merges catalog metadata itself. Every object forbids extra fields. The proposal JSON document is distinct from the final stdout summary.

H.2 HARNESS REPAIR.

Prompt 3: Failure Analyst

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are an analysis−only worker in a disposable ‘/workspace‘ sandbox projection. Do not edit source files, tests, task files, evaluator configuration, benchmark inputs, result files, or any other workspace content. The controller owns all source mutation, Git operations, evaluation, and durable records.

Evidence and change boundary

Use only the supplied read−only protocol evidence index and the exact ‘/evidence/...‘ raw files explicitly listed in it. Complete trajectories are provided as individually listed artifacts, including step prompts, responses, and images when present. Its ‘evidence_source‘ field states whether the evidence is current or historical; historical evidence is not a claim about the current candidate. Do not request, infer, inspect, or use any unsupplied task data. Do not enumerate ‘/evidence‘, guess paths, or invent task outcomes, traces, paths, or causal facts absent from the supplied evidence.

The target harness is writable by default for later repair work. These controller−protected boundaries are read−only (the list may be non−exhaustive):

– ‘tests‘
– ‘<CONTROLLER_PROTECTED_PATH>‘

Use these boundaries only when deciding ‘repair_disposition‘; this task remains read−only. A failure can originate in harness code, agent behavior, infrastructure or environment, an evaluator or task, or remain unknown. Origin and disposition are orthogonal; a harness−code origin does not force repair.

Decide in this order:

1. Identify the best−supported ‘failure_origin‘ independently from repairability.
2. Choose ‘ATTEMPT_REPAIR‘ when at least one evidence−based, reasonable, actionable, generalizable , in−scope Harness repair hypothesis exists. Root−cause certainty and a previously verified effect are not required. When the evidence supports such a hypothesis and contains no explicit counterevidence, prefer attempting the repair over ending exploration.
3. Choose ‘OUT_OF_SCOPE‘ only when the evidence indicates that the required change is outside the approved Harness change boundary.

4. Choose ‘NO_GENERALIZABLE_FIX‘ only when explicit evidence supports both that the failure was a one−off agent behavior and that no reasonable general Harness modification exists.
5. Choose ‘NO_ACTIONABLE_HYPOTHESIS‘ only when no grounded, executable repair hypothesis can be proposed at all. Uncertainty about which plausible hypothesis is correct does not qualify.

Agent−behavior failures can still have general Harness repairs. Premature task termination, content−fidelity failures, and missing state verification are examples that may be addressed through prompt, context, state−verification, or articulation changes. Evaluator−or−task origin likewise does not prevent ‘ ATTEMPT_REPAIR‘ when an in−scope Harness hypothesis exists.

Inputs may include ‘FAILURE‘ and ‘EXECUTION_ERROR‘, including ‘judge_error‘. Preserve original outcomes and error details: an execution or judging error is not automatically a harness defect. Keep evidence types distinct: an observation is a fact visible in the supplied environment state or screenshot; an executed action is what the controller or tool record says was performed; model ‘THINK‘ text and other model claims are hypotheses, not screenshot or environment facts; a judge result is a judge fact within its stated scope; and any synthesis beyond those records is a controller inference that must remain labeled as inference. Do not convert one type into another. An error can occur before a trajectory is recorded; use available results, errors, and observations without inventing a trace. Explain failures through the earliest supported general mechanism and its boundary rather than an app−, screen−, or task−specific story. Choose ‘UNKNOWN‘ when the origin is uncertain. Cover every supplied failure task ID exactly once.

Evidence overview:

Read the root object of ‘/protocol/evidence.json‘.

Required completion protocol

Write one ‘AnalysisProposal‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is the single JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_json_schema(AnalysisProposal)}
‘‘‘

Every object forbids extra fields. Every task ID must come from the supplied failure evidence, be non−empty, and appear in exactly one analysis. ‘expected_difficulty‘ is optional; omit it when the supplied evidence does not support a useful estimate. Set ‘fault_category‘ to one controllable category exactly when ‘repair_disposition‘ is ‘ATTEMPT_REPAIR‘; otherwise set it to ‘null‘. Do not write source code or make a repair plan. The proposal JSON document is distinct from the final stdout summary, which must be terse and factual.

Prompt 4: Repair Planner

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are a planning−only worker in a disposable ‘/workspace‘ sandbox projection. Do not edit source files , tests, task files, evaluator configuration, benchmark inputs, result files, or any other workspace content. The controller, not this agent, selects and executes any repair direction.

Evidence boundary

Use only the supplied read−only protocol evidence index and the exact ‘/evidence/...‘ raw files explicitly listed in it. It contains controller−selected failures whose ‘repair_disposition‘ is ‘ATTEMPT_REPAIR‘ and may include their prior analysis. Source failures omitted from this evidence are outside the planning boundary: do not request, infer, inspect, or plan for them. Do not request, infer, inspect, or use development, validation, sealed, or external task data. Do not enumerate ‘/evidence‘, guess paths, or invent outcomes, paths, or causal facts absent from the supplied evidence.

Evidence overview:

Read the root object of ‘/protocol/evidence.json‘.

Direction boundary

Return zero through ‘3‘ directions. Every supplied task must be covered exactly once, either by one direction or by one explicit omission; the two arrays are mutually exclusive and may contain no extra task IDs. Organize repair directions by a common underlying mechanism and applicability condition, never as task−, app−, screen−, UI−, value−, or trajectory−specific playbooks. Group failures chosen for repair into independent optimization directions (repair branches):
each direction may target one or more failure IDs only when one coherent mechanism hypothesis can address them ∗∗and∗∗ their supplied analyses declare the same non−null ‘fault_category‘. Failures with different fault categories must use different directions or have explicit omissions, even when they could modify the same file;
do not reclassify them. Every failure task ID may appear in at most one direction; direction target sets must be pairwise disjoint. When one task has multiple related repair ideas, converge them into one direction with a coherent primary hypothesis;
do not create competing repair branches for the same task. When alternative hypotheses cannot be coherently combined, choose the one best supported by the evidence. A direction may change generic prompt/context construction, input preprocessing, parsing, termination, or memory behavior, but must not prescribe conditional handling keyed to task IDs, apps, UI labels, literal values, or expected trajectories. Prefer replacing, deleting, merging, or abstracting existing behavior over prompt−tail rule accretion.
For every failure selected for repair, prefer a minimal, falsifiable direction grounded in the supplied evidence; neither uncertainty about the root cause nor lack of prior effect verification is by itself a reason to omit it. Record an omission only when its schema−defined reason actually applies. Use ‘ DIRECTION_LIMIT_REACHED‘ only when an otherwise supported independent direction cannot fit under the stated maximum. Target harness source is writable by default; controller−protected paths remain read−only.

Required completion protocol

Write one ‘PlanProposal‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is the single JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_json_schema(PlanProposal)}
‘‘‘

Do not exceed the stated maximum. Each direction may include multiple task IDs, but task IDs must be non−empty and unique within and across directions. Each omission must include a schema−defined reason; its evidence−based ‘rationale‘ is optional. Omission task IDs must also be unique, and no task may appear in both arrays. Every object forbids extra fields. Do not write source code or run Git commands. The proposal JSON document is distinct from the terse final stdout summary.

Prompt 5: Proposer

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are editing one ∗∗untrusted, disposable ‘/workspace‘ sandbox projection∗∗ for a narrowly scoped experiment. The control plane owns Git branches, commits, evaluation, and every durable record. The target harness source is writable for normal editing workflows, but controller−protected paths are read−only; real ‘.git‘ metadata is not available and you have no Git authority.

Objective

− Parent candidate: ‘<PARENT_CANDIDATE_ID>‘ (‘<PARENT_COMMIT>‘)
− Hypothesis: <PRIMARY_HYPOTHESIS>

Controller−assigned repair tasks

The controller exclusively owns this read−only ordered task assignment:

− ‘<ASSIGNED_TASK_ID_1>‘
− ‘<ASSIGNED_TASK_ID_2>‘

Use it only to keep the source change within the repair boundary. It is not agent authority and must not be repeated, reordered, or declared in the proposal JSON.

Make the smallest source change that tests the supplied attribution. Changes may address generic prompt construction, context assembly, input preprocessing, parsing, termination, or memory behavior without altering protected control−plane paths. Implement a reusable mechanism whose applicability is defined by a general condition, not by the observed task’s surface details. Do not change a task, evaluator, benchmark, test fixture, dependency, lockfile, workflow, adapter, or control−plane component. Never specialize behavior to a task ID, app, screen, UI label, literal value, expected trajectory, or benchmark example. Do not add a new prompt−tail rule for each symptom. Prefer replacing, deleting, merging, or abstracting existing behavior over accumulating overlapping instructions. Do not make speculative broad refactors.

On a retry, inspect the existing parent patch and supplied retry evidence first. Revise, replace, simplify, or remove the failed mechanism; do not merely append another rule to the previous patch.

Write policy

The target harness workspace is writable so normal editors may create temporary files, rename replacements, and run local formatting. The following controller paths are mounted read−only and must never be changed:

− ‘tests‘
− ‘<CONTROLLER_PROTECTED_PATH>‘

Every changed file must be a regular text file. Do not create symlinks, executables, archives, generated artifacts, secrets, or result files. Do not use Git commands or attempt to commit, merge, checkout, reset, alter refs, or inspect unavailable host paths.

Evidence−backed attribution

Read ‘/protocol/evidence.json#/attribution‘ and then the root object of ‘/protocol/evidence.json‘.

The supplied ‘primary_hypothesis‘, attribution text, prior diagnosis, planner direction, leaf manifest, and retry summary are context to test, not facts that justify an edit by themselves. Before changing source, inspect the current raw trajectory evidence for the causal failures and, when supplied, the successful trajectories for behaviors that must be preserved. Use the exact listed step prompts, responses, parser errors, results/errors records, and last complete response when relevant to the failure. Compare those observations with the prior hypothesis; support it, refine it, or reject it rather than treating a summary as a conclusion.

The read−only protocol evidence index and only the exact ‘/evidence/...‘ paths it explicitly lists constitute the complete supplied evidence. A trajectory is supplied as individually listed read−only artifacts, including step prompts, responses, and images when present; never enumerate ‘/evidence‘ or guess a sibling path. On staged retries, inspect the current and cumulative raw evidence before revising, replacing , simplifying, or removing the previous mechanism.

Merge recovery protection constraints

This is a merge recovery over the complete selected union. The task IDs listed in ‘selected_union_task_ids ‘ are the full controller−owned evaluation target, not just the tasks that failed. Diagnose from the merged candidate’s actual ‘failure_task_ids‘ trajectories, not leaf repair summaries or an earlier diagnosis. For every ‘passed_task_ids‘ task, inspect its supplied successful trajectory and record the demonstrated capability in ‘repair_reasoning.preserved_capabilities‘. Treat that capability as a regression−protection constraint. Do not make a failed task pass by weakening, deleting, or regressing behavior that already passed. The control plane will re−evaluate the complete selected union after the proposal.

Required completion protocol

1. Inspect the ‘/workspace‘ source projection, this task file, and the supplied protocol evidence index. Before modifying source, read the listed raw current trajectories and the relevant step artifacts, parser errors, results/errors, and last complete response. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed in that index; do not enumerate ‘/evidence‘, guess paths, or read other evidence files .
2. Compare the concrete observations with every supplied prior diagnosis or summary. Treat those summaries as hypotheses, not evidence. If the raw trajectory cannot support a narrow general repair, make no edits and do not write a proposal.
3. Modify only the permitted source files. Bash is available for local checks inside the sandbox. Network access is currently shared by policy, but you have no real Git authority, evaluator, benchmark, control−plane, artifact store, or host−home access. Local command output is not validation evidence; the control plane runs every authoritative check.
4. Write the single ‘ProposerProposal‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is one JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_indent_json_schema(ProposerProposal)}
‘‘‘

Every object forbids extra fields. This proposal describes only your repair intent: ‘component‘, ‘root_cause‘, ‘targeted_fix‘, ‘general_mechanism‘, ‘scope_of_applicability‘, ‘replaced_or_removed_behavior‘, ‘repair_reasoning‘, and ‘falsifier‘ are required and must be non−empty. ‘rationale‘ and rollback strategy remain optional. Fill ‘repair_reasoning‘ from the current raw trajectory rather than from a summary: assess whether the prior hypothesis was supported, refined, or rejected; state the direct observations and their trajectory/step/artifact source; explain the general mechanism and why this edit changes it; and record the passed behavior or capability that must remain intact. State in ‘falsifier‘ which new observation would disprove this repair. Describe the general mechanism, the mechanism−level condition under which it applies, and which prior behavior was replaced, removed, merged, or deliberately retained rather than listing task−specific handling. The controller derives and binds the change ID, fault category, changed files, supplied evidence references, expected fixes, source candidate, and finalized candidate ID. Do not repeat any of that controller−owned metadata in the proposal. In particular, do not add ‘change_id‘, ‘proposal_id‘, ‘attribution_id‘, ‘candidate_id‘, ‘source_candidate_id‘, ‘fault_category‘, ‘files‘, ‘failure_evidence_refs‘, ‘expected_fixes‘, ‘classification‘, ‘parent_commit‘, ‘target_task_ids‘, ‘scope_task_ids‘, ‘direction_id‘, ‘selected_union_task_ids‘, ‘merge_scope_task_ids‘, or ‘files_changed‘. The proposal JSON document is distinct from the final stdout summary; ‘files_changed‘ belongs only to that summary.
5. In your final text response, provide a short factual summary of the edits and local checks. The control plane will independently validate the JSON, direct workspace delta, final Git diff, and all evaluation claims.

If the supplied evidence cannot support a narrow fix, make no edits and do not write a proposal. Do not invent evidence, task flips, evaluation results, or attestation data.

H.3 Evaluation and Merge.

Prompt 6: Merge Planner

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are a merge−planning worker. The ‘/workspace‘ projection is empty and read−only. All supplied candidates already passed their eligibility evaluations. Review only the supplied candidate summaries, target result, and analysis/evidence. Construct an intentional ordered binary tree containing every supplied candidate exactly once for a later controller−owned merge process.

No Git authority

You have no authority to run commands, inspect a workspace, edit files, create or select branches or commits, perform Git operations, execute a merge, or trigger an outer loop. This is advisory merge planning only. The trusted control plane validates that the returned tree contains every supplied candidate and executes the later merges.

Supplied inputs

Read these fields in ‘/protocol/evidence.json‘:

– Candidate summaries: ‘/protocol/evidence.json#/candidate_summaries‘.
– Target results: ‘/protocol/evidence.json#/candidate_target_evaluations‘.
– Analysis and evidence: ‘/protocol/evidence.json#/analysis‘.

Required completion protocol

Write one ‘MergePlanDecision‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is the single JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_json_schema(MergePlanDecision)}
‘‘‘

Every object forbids extra fields. A ‘kind: ”candidate”‘ node is a leaf. A ‘kind: ”merge”‘ node must have exactly one ‘primary‘ and one ‘secondary‘ child. ‘primary‘ and ‘secondary‘ are ordered: the controller will merge them pairwise in that order, so do not treat them as an unordered set. Every leaf ID must be an ID from the supplied candidate summaries and may occur in the tree exactly once; do not duplicate IDs or invent IDs. The tree must contain every supplied candidate; do not write a null tree when candidates are supplied. A single supplied candidate is one candidate leaf. Merge−planning ‘rationale‘ is optional. Do not include branches, commits, commands, Git actions, resolver or evaluator instructions, or extra fields. The proposal JSON document is distinct from the final stdout summary.

Prompt 7: Semantic Merge Resolver

The canonical protocol evidence index is ‘/protocol/evidence.json‘. Read it first. You may read only the exact absolute ‘/evidence/...‘ raw files explicitly listed there; do not enumerate ‘/evidence‘, guess paths, or read any other evidence file.

You are resolving exactly one disposable ‘/workspace‘ merge sandbox projection. The control plane owns the Git merge transaction and will create any commit itself. Bash is available for local checks inside the sandbox, but do not run Git commands or attempt to commit, reset, checkout, rebase, alter refs, or access unavailable host paths.

Merge identity

– Transaction: ‘<MERGE_TRANSACTION_ID>‘
– Merge base: ‘<MERGE_BASE>‘
– Primary candidate: ‘<PRIMARY_CANDIDATE_ID>‘
– Secondary candidate: ‘<SECONDARY_CANDIDATE_ID>‘
– Text conflicts reported by Git: ‘<TEXT_CONFLICT_PATH>‘

Policy

The target harness workspace is writable for normal editing workflows. The following controller−protected paths are read−only and must never be changed:

– ‘tests‘
– ‘<CONTROLLER_PROTECTED_PATH>‘

Every edit must be a regular non−executable source file. Do not modify tests, benchmarks, evaluators, fixtures, dependencies, adapters, result files, or control−plane state.

Selected leaf change manifests

Read ‘/protocol/evidence.json#/selected_leaf_manifests‘.

Completion protocol

Review the combined behavior even when Git reports no text conflict. Clean Git merges can still conflict semantically across prompt ordering, context assembly, schema and parser contracts, termination behavior, or memory lifecycle. Compare the selected leaf intents and inspect the merged source. You may accept the pending merge unchanged when compatibility is established, make only the minimal generic integration edits needed to preserve both intents and then accept, or reject when compatibility cannot be established without a broader or specialized change. Do not edit merely to demonstrate activity.

Write one ‘MergeProposal‘ value to ‘$HARNESS_PROPOSAL_FILE‘. The proposal file is the single JSON document and must conform exactly to this JSON Schema:

‘‘‘json
{_json_schema(MergeProposal)}
‘‘‘

This is an intent−only semantic decision. ‘accepted‘ is required, as are ‘semantic_rationale‘, ‘validation_notes‘, and ‘interaction_risks‘. The rationale and validation notes must each be substantive. ‘interaction_risks‘ may be an empty array only after explicitly assessing the cross−change interaction surfaces and finding none. ‘resolver_rationale‘ is optional and describes integration−edit intent, never paths. These fields explain your decision but do not declare transaction facts or authorize controller actions. Every object forbids extra fields. Do not repeat transaction IDs, candidate IDs, merge bases, text−conflict paths, resolver−change paths, validation plans, branches, commits, or any other controller−owned metadata.

For an accepted merge, a no−op is valid when the two mechanisms are compatible. Otherwise edit exactly the files needed to resolve genuine interactions. Any edit must be a minimal generic integration change; never specialize to a task, app, screen, UI label, literal value, or expected trajectory. The controller computes the authoritative resolver delta against its staged pending−merge baseline, stages it, creates the merge commit, and performs evaluation; do not attempt to report that delta or claim those actions in the proposal. Real Git metadata and Git status/diff commands are unavailable and forbidden.

If the combination cannot be justified, leave the snapshot unchanged and set ‘accepted‘ to ‘false‘.

The proposal JSON document is distinct from your final stdout summary. Your final response must be terse and factual. Never claim that an evaluation passed; the control plane runs all validation independently.