# MODULARRSI: MODULAR AND GENERALIZABLE RECURSIVE HARNESS SELF-IMPROVEMENT

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Recent work extends Recursive Self-Improvement (RSI) to agent harnesses for long-horizon coding and terminal tasks, enabling agents to improve their execution mechanisms from experience. However, achieving **generalizable harness RSI** faces two challenges: evolution on downstream benchmarks makes transferable improvements difficult to distinguish from benchmark-specific adaptation, while coarse task-level outcomes provide limited guidance on **where that change should be implemented** and **what execution behavior should change** in the harness. We propose **ModularRSI**, a modular framework for generalizable harness self-evolution that addresses both **mechanism-level localization** and **behavioral-level attribution** in credit assignment. ModularRSI decomposes the harness into five functional modules, including *Agent Loop*, *Tool Use*, *Observation Management*, *Context Management*, and *Task Completion Detection*, and evolves each module independently to enable localized mechanism improvement. Within each module, ModularRSI performs contrastive analysis between successful and failed trajectories from the same task and aggregates evidence across tasks to identify recurring execution deficiencies. Finally, the independently evolved modules are integrated to construct the final harness. We further curate 2,000 executable evolution tasks from external sources that are disjoint from downstream evaluation benchmarks. Experiments on TerminalBench (TB) 2.0 and SWE-Bench-Verified demonstrate consistent generalization across unseen tasks, evolution domains, and foundation models. Using DeepSeek-V4-Flash-Preview, in-domain evolution improves accuracy from 47.57% to 52.43% on TB 2.0 and from 73.40% to 76.40% on SWE-Bench-Verified. Beyond in-domain settings, the evolved harness also transfers effectively across evolution domains and foundation models, yielding approximately 2%–5% performance gains. Under a unified benchmark-disjoint protocol and matched evolution budgets, ModularRSI achieves 52.43% accuracy on TerminalBench 2.0, outperforming Meta-Harness at 49.44% and AHE at 49.81%. Ablation studies further confirm that both modular evolution and contrastive analysis contribute to these improvements.

## 1 INTRODUCTION

CLI agents have achieved remarkable performance on complex software engineering and terminal-based tasks (Jimenez et al., 2024; Deng et al., 2025; Merrill et al., 2026; Hong et al., 2026). Beyond foundation models, their effectiveness increasingly depends on agent harnesses that govern execution, tool interaction, context management, and environment feedback. Recent work has therefore explored recursive self-improvement (RSI) of harnesses, allowing agents to refine these execution mechanisms from experience.

However, the central challenge of **generalizable harness RSI** is not whether a harness can improve on the experience used for evolution, but whether the resulting changes represent **transferable improvements to the underlying execution mechanism**. Existing methods (e.g., Meta-Harness, AHE, and HarnessFix) often evolve directly on downstream benchmarks or subsets derived from them (Lee et al., 2026; Lin et al., 2026; Du et al., 2026; Chen et al., 2026a; Pan et al., 2026; Nguyen et al., 2026), making reusable improvements difficult to distinguish from benchmark-specific adaptation. Even under separated evolution and evaluation, task-level rewards provide only coarse supervision: individual trajectories entangle systematic harness deficiencies with task-specific reason-



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.

ing, while broad whole-harness updates make it difficult to localize reusable improvements without affecting unrelated mechanisms. We therefore view **generalizable harness RSI as a two-level credit-assignment problem**, which becomes particularly important under benchmark-disjoint evolution. A successful self-improvement procedure must answer two complementary questions: **where in the harness should that improvement be implemented** and **what execution behavior should be improved**. The first is a **mechanism credit-assignment problem**: once a reusable deficiency has been identified, the corresponding change must be localized to the responsible harness components rather than broadly modifying unrelated components. The second is a **behavioral credit-assignment problem**: execution trajectories must be compared and aggregated to distinguish recurring harness deficiencies from task-specific decisions or stochastic failures. Our central hypothesis is that transferable harness self-improvement requires improving the precision of credit assignment at both levels, rather than simply enlarging the self-modification space. HarnessFix (Chen et al., 2026a) improves failure localization from execution trajectories, but stops short of explicitly assigning the identified deficiencies to the responsible harness mechanisms. AHE (Lin et al., 2026) recognizes functionally distinct components within the harness, yet still evolves them within a joint harness-level process rather than independently attributing and optimizing mechanism-specific improvements. Consequently, behavioral and mechanism-level credit assignment remain only partially addressed.

To address these two levels of credit assignment, we propose ModularRSI, a modular approach for generalizable harness evolution. For mechanism-level credit assignment, it decomposes the evolvable harness into five functionally coherent modules (i.e., Agent Loop, Tool Use, Observation Management, Context Management, and Task Completion Detection). Each module is evolved independently under a restricted modification scope, followed by cross-module integration into a unified harness. For behavioral-level credit assignment, it contrasts successful and failed trajectories within the same task and aggregates evidence across tasks to identify recurring deficiencies beyond task-specific variation, while guiding localized updates to the responsible harness mechanisms.

To enable a clean evaluation of generalizable harness RSI, we establish a **benchmark-disjoint evolution protocol** and curate 2,000 executable tasks from external sources that are disjoint from downstream benchmarks. Experiments on **TerminalBench 2.0** and **SWE-Bench Verified** show consistent generalization across unseen tasks, domains, and foundation models, while controlled studies demonstrate the benefits of modular evolution, contrastive trajectory analysis, and some other components.

Our main contributions are summarized as follows:

1. We formulate **generalizable harness RSI as a benchmark-disjoint credit-assignment problem** and propose **ModularRSI**, which performs credit assignment at two complementary levels. Contrastive trajectory analysis identifies recurring behavioral deficiencies, while module-restricted evolution localizes and implements the corresponding mechanism-level improvements.
2. We establish a **benchmark-disjoint evolution protocol** and independently curate 2,000 executable evolution tasks from external data sources. Through executable validation, manual quality control, and similarity screening, this resource enables harness self-improvement to be studied without direct exposure to downstream benchmark instances.
3. We extensively evaluate ModularRSI on **TerminalBench 2.0** and **SWE-Bench Verified**. The evolved harness generalizes across unseen tasks, evolution domains, and foundation models. Controlled studies further show that modular evolution substantially outperforms joint and non-modular evolution under matched budgets, while removing contrastive attribution degrades performance, supporting the importance of both behavioral and mechanism-level credit assignment.

## 2 Related Work

### 2.1 Agent Harnesses and Self-Improvement

Alongside advances in coding models (Yang et al., 2026a;b), executable environments and verified trajectories provide scalable signals for agent improvement (Wu et al., 2026a; Gandhi et al., 2026;

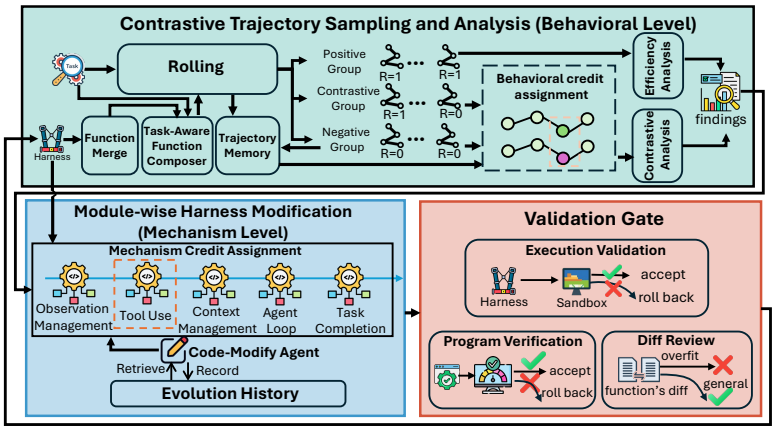


Figure 1: Overview of ModularRSI. The framework collects and analyzes contrastive trajectories, performs module-wise harness modification, and retains modifications through validation gates.

Cheng et al., 2026). Recent work uses trajectory feedback to optimize prompts and external agent mechanisms (Agrawal et al., 2026; Zhang et al., 2026c; Fan et al., 2026), diagnose failures and localize editable components (Lin et al., 2026; Chen et al., 2026a), or directly evolve agent by synthesizing harness code (Zhang et al., 2026b; Lou et al., 2026; Lee et al., 2026; Ren et al., 2026).

### 2.2 Generalization in Self-Improvement

A central challenge is distinguishing transferable improvements from adaptation to development data. HarnessDev and related studies examine harness evolution, feedback budgets, and held-out generalization (Wu et al., 2026b; Wang et al., 2026). Aspire studies weight and harness evolution under hidden evaluation (Wu et al., 2026c), while $S^3$Gym separates self-testing, self-judging, and improvement through history, memory, or parameter updates (Shi et al., 2026). Despite these advances, existing harness-evolution methods often rely on benchmark-derived data or treat the harness as a monolithic whole, leaving generalizable, fine-grained improvement underexplored. We address this gap with a benchmark-disjoint evolution dataset and protocol, and propose ModularRSI, which uses contrastive trajectory analysis to guide independent, module-restricted evolution followed by cross-module integration.

## 3 Method

### 3.1 Overview of ModularRSI

As illustrated in Fig. 1, ModularRSI organizes harness self-improvement through **module-restricted evolution** and subsequent **cross-module integration**. It independently evolves five functional modules from the same initial harness without sharing intermediate updates. Within each module-specific lineage, contrastive trajectory attribution identifies behavioral deficiencies and guides localized function updates through three stages: (i) **Contrastive Trajectory Sampling and Analysis**, which identifies behavioral deficiencies from execution evidence; (ii) **Module-wise Harness Evolution**, which converts the findings into localized function updates; and (iii) **Validation Gates**, which screen modifications for executability and generalizability.

The agent under evolution serves as the **Code-Modify Agent** for both trajectory analysis and harness modification, processing evidence in batches and restricting edits to the target module. The independent lineages can be processed in parallel. After module-wise evolution, **Cross-Module Integration** combines the evolved modules and resolves potential conflicts. Throughout evolution, **Function Merge** controls library redundancy, while a **Task-Aware Function Composer** selects task-appropriate functions from each module.

### 3.2 HARNESS MODULARIZATION

We reorganize the behavioral mechanisms of Terminus-2 in Harbor (Merrill et al., 2026) into five functional modules. Based on our analysis of harness implementations and execution trajectories, the decomposition covers mechanisms that directly mediate agent–environment interaction. Infrastructure components, such as sandbox initialization, parallel execution, and LLM communication, remain outside the evolution scope.

**(1) Agent Loop** controls the reasoning–action–observation cycle, including execution flow, interaction control, and recovery behavior; **(2) Observation Management** preserves task-relevant environmental feedback while filtering or compressing noisy observations; **(3) Tool Use** manages tool selection, invocation, and validation; **(4) Context Management** maintains interaction history through information retention, compression, and retrieval; and **(5) Task Completion Detection** determines whether the task is complete or further interaction is required.

Each module defines a distinct functional scope for subsequent evolution. During execution, the Agent Loop coordinates the other modules and constructs prompts for the LLM to interact with the environment. Module interactions and function interfaces are detailed in Appendix B.

### 3.3 CONTRASTIVE TRAJECTORY SAMPLING AND ANALYSIS

For each evolution instance $x_i$, we collect $K$ trajectories $(tra_i^1, \ldots, tra_i^K)$. The task-specific evaluator assigns each trajectory a binary reward $r_i^k \in \{0, 1\}$, where 1 indicates success. Based on these rewards, we categorize tasks as **Positive** if all $K$ rollouts succeed, **Contrastive** if some succeed and others fail, and **Negative** if all fail.

A **Trajectory Memory** stores historical trajectories and rewards for each task across evolution epochs, providing additional evidence when current rollouts are insufficient.

The Code-Modify Agent analyzes each task with respect to the target module. For the **Contrastive group**, it compares successful and failed trajectories of the same task to identify function-level factors associated with different outcomes. For the **Negative group**, it retrieves a historical success, when available, and pairs it with a current failed trajectory for same-task comparison. Otherwise, it diagnoses evident execution deficiencies from failed trajectories alone. For the **Positive group**, it examines successful trajectories for opportunities to improve execution quality and efficiency.

The diagnoses are recorded as structured JSON findings specifying the module under analysis, supporting trajectory evidence, the rationale for or against modification, and a proposed change when applicable. The analysis prompt and output schema are provided in Appendices J.1 and I.

### 3.4 MODULE-WISE HARNESS MODIFICATION

Within each independent evolution lineage, the Code-Modify Agent converts the findings from Sec. 3.3 into function updates restricted to the target module. Two mechanisms guide the selection and implementation of these updates.

**Modification Target Selection.** For each batch, we consolidate semantically similar diagnoses targeting the same function into candidate modifications. Each candidate receives a vote count equal to the number of distinct tasks providing supporting evidence. We then rank the candidates by vote count and prioritize the top-$K$ candidates for modification, thereby focusing on recurring deficiencies rather than instance-specific failures. We set $K = 10$ in all experiments.

**Evolution History.** For each function, we record previous code changes and the functionality introduced by each revision. The Code-Modify Agent uses this history to preserve accumulated capabilities and avoid repeated or contradictory modifications across iterations. The modification prompt is provided in Appendix J.2.

### 3.5 VALIDATION GATES

Each function update must pass three sequential validation gates before being retained. A failure at any gate triggers rollback using the recorded diffs.

**Program Check.** Static checks verify the modified harness's validity and compatibility with the surrounding system, including imports and protocol compliance.

**Diff Review.** The Code-Modify Agent inspects the modification diff for task-specific solutions, heuristics, or conditions unlikely to generalize beyond the evolution instances. The review prompt is provided in Appendix J.3.

**Execution Validation.** We randomly sample two tasks from the current batch and execute them using the modified harness to detect runtime errors or violations of the expected execution protocol.

Additional validation details are provided in Appendix C.

### 3.6 Cross-Module Integration

After independent evolution, we combine the validated module variants into a unified harness. Although each variant is developed within its own functional scope, composing them may introduce duplicated mechanisms, conflicting behaviors, or inconsistent interactions.

We therefore perform one additional **cross-module integration epoch** on the evolution set. The Code-Modify Agent analyzes trajectories from the integrated harness to identify cross-module conflicts, then refines module interactions by removing duplicated mechanisms, clarifying responsibilities, and adjusting coordination logic. The integration prompt is provided in Appendix J.4.

### 3.7 Function Library Management

As module-wise evolution expands the function library, we control within-module redundancy and task-specific function activation through two complementary mechanisms.

**Function Merge.** During evolution, an LLM compares function descriptions and behaviors within each module, merging functions with similar or overlapping functionality to reduce redundancy while preserving learned capabilities. Function Merge is not applied during downstream evaluation.

**Task-Aware Function Composition.** Each evolved function has a natural-language description of its current behavior. Given the task description and descriptions of all candidate functions, an LLM selects a task-relevant subset, and only these functions are activated during execution. The composer operates during both trajectory collection and downstream evaluation.

After cross-module integration, the function library is frozen. Downstream evaluation permits task-aware selection from this fixed library, but no further function merging or modification.

## 4 Evolution Dataset and Protocol

Guided by high-level task categories from the **TerminalBench** and **SWE-Bench** families, human annotators retrieve publicly available resources from external sources, including GitHub repositories, Hugging Face, Kaggle, and Linux kernel documentation. These resources are used to construct new executable tasks in the Harbor format. Benchmarks provide only category-level retrieval guidance; no benchmark instances or task-specific information are used during task construction.

All instances undergo a multi-stage quality-control pipeline. LLM-based filtering assesses **Environment Completeness**, checking the availability of required data, dependencies, files, and tools; **Practicality and Non-triviality**, excluding artificial or overly simple tasks with limited reasoning or interaction; and **Evaluator Validity**, requiring tests to assess functional correctness rather than superficial completion. Executable validation requires the reference `solution.sh` to pass all task-specific tests and a no-op submission to receive no positive reward. Each instance is then manually reviewed for task clarity, environment completeness, and evaluator correctness, followed by LLM-based semantic similarity screening to remove instances with high overlap with benchmarks.

As shown in Fig. 4 in Appendix E, we curate 2,000 high-quality, benchmark-disjoint evolution instances, with relatively balanced task categories and broadly comparable distributions between the TB-related and SWE-related subsets. Their distinct construction sources limit overlap: SWE-related tasks involve repository-level software engineering on GitHub, whereas TB-related tasks

focus on direct terminal interaction. The dataset provides a foundation for studying whether harness self-evolution yields generalizable improvements rather than benchmark-specific adaptations.

# 5 Experiment Setting

## 5.1 Benchmarks

To validate the effectiveness of ModularRSI, we evaluate it against the baseline on two recent and challenging terminal-related benchmarks: **TerminalBench 2.0** (Merrill et al., 2026) and **SWE-Bench-Verified** (Jimenez et al., 2024). TerminalBench 2.0 contains **89** diverse long-horizon terminal tasks, while SWE-Bench-Verified contains **500** human-validated software engineering tasks from real-world repositories. Together, they provide complementary evaluation of general terminal interaction and repository-level software engineering. For reproducibility, we use the original released benchmark versions and conduct all evaluations under the Harbor framework.

## 5.2 Metrics

Considering the stability of the evaluation results and the need to compare both the efficiency and performance of different harnesses, we evaluate the following four metrics:

**Accuracy (Acc):** Accuracy measures the average success rate over all rollout trajectories. A trajectory is considered successful if the agent successfully completes the corresponding task according to the task-specific evaluator.

**Pass@3:** Pass@3 measures whether the agent can successfully solve a task within three independent rollout attempts. A task is counted as solved if at least one of the three trajectories succeeds, and the final score is averaged across all evaluation tasks.

**StepNum:** StepNum measures the average number of agent-environment interaction steps required per rollout. It reflects the execution efficiency of the agent, with fewer steps generally indicating a more efficient problem-solving process.

**Pass$^3$:** Pass$^3$ measures the consistency and reliability of the agent across 3 independent rollout attempts. A task is counted as successfully solved only if all 3 trajectories succeed, and the final score is averaged across all evaluation tasks.

## 5.3 Implementation Details

Each module-specific evolution lineage in ModularRSI is evolved for **3 epochs**. The five independently evolved modules are subsequently combined through one cross-module integration epoch. For the experiments with different methods or settings that evolve the entire harness jointly, we adjust the number of epochs to match the aggregate evolution budget of ModularRSI, as detailed in Sec. 6.4 and Sec. 6.3. Due to computational and time constraints, we select **120** instances from each of the TB-related and SWE-related subsets of the 2,000-instance evolution dataset for harness evolution. The primary model used for evolution is **DeepSeek-V4-Flash-Preview**. We provision the deployed models with a **TPM limit of 2 million tokens** and use a **batch size of 10** throughout the evolution process. In this work, we regard Terminus-2 as the No-Evolution (No-Evol) Baseline.

# 6 Results

## 6.1 Generalization Beyond Evolution Experience

The central challenge of Harness RSI is not only improving performance on the tasks used for evolution, but discovering harness modifications that generalize beyond the evolution experience. To evaluate this capability, we construct two independent evolution sets, i.e., TB-related and SWE-related. From each set, we sample 120 instances for harness evolution, resulting in 120 TB-related and 120 SWE-related evolution tasks. We then evaluate the evolved harnesses on both Terminal-Bench 2.0 and SWE-Bench-Verified under in-domain and out-of-domain settings. The evolution

Table 1: Cross-benchmark transfer of evolved harnesses. After evolution on TB-related or SWE-related tasks, each harness is evaluated on both held-out benchmarks.

| | SWE-Bench-Verified | | | | TerminalBench 2.0 | | | |
|---|---|---|---|---|---|---|---|---|
| **Evolution Set** | **Eval. Setting** | **Acc ↑** | **Pass@3 ↑** | **Pass$^3$ ↑** | **Eval. Setting** | **Acc ↑** | **Pass@3 ↑** | **Pass$^3$ ↑** |
| No Evolution | – | 73.40 | 83.20 | 62.80 | – | 47.57 | 58.43 | 30.34 |
| TB-Related | Out-of-Domain | 75.80 | 84.60 | 66.20 | In-Domain | **52.43** | **65.17** | **35.96** |
| SWE-Related | In-Domain | **76.40** | **85.40** | **66.80** | Out-of-Domain | 49.44 | 60.67 | 30.34 |

sets contain no downstream benchmark instances, trajectories, evaluator feedback, or task-specific content; benchmark taxonomies are used only as coarse category-level guidance for source coverage.

As shown in Tab. 1, with **DeepSeek-V4-Flash-Preview** as the backbone model, ModularRSI consistently improves performance across both evaluation benchmarks under different evolution settings. In the in-domain setting, the evolved harness improves accuracy from 47.57 to 52.43 on Terminal-Bench 2.0 and from 73.40 to 76.40 on SWE-Bench-Verified. We further analyze performance across successive RSI generations and observe a largely monotonic improvement throughout the evolution process (see Appendix M).

More importantly, these gains transfer beyond the evolution domain. The harness evolved on TB-related data improves SWE-Bench-Verified accuracy to 75.80, while the harness evolved on SWE-related data improves TerminalBench 2.0 accuracy to 49.44. These results suggest that ModularRSI discovers reusable improvements to the underlying agent execution mechanism, rather than merely specializing to the tasks encountered during evolution.

Notably, the gain is particularly pronounced for $\mathrm{Pass}^3$ on TB 2.0, which increases from 30.34 to 35.96. This improvement suggests that harness evolution not only increases average task-solving capability but also enhances execution reliability, reducing stochastic failures across repeated trials.

### 6.2 Cross-Model Generalization

Beyond task-level generalization, we further examine whether the discovered harness improvements are transferable across different foundation models. Since different models exhibit different behaviors in planning, tool interaction, and error recovery, a model-specific optimization may fail to generalize.

Table 2: Generalization across different inference models with an evolved harness on TB 2.0.

| **Infer Model** | **Method** | **Acc ↑** | **Pass@3 ↑** | **Pass$^3$ ↑** |
|---|---|---|---|---|
| GLM-5.2 | No-Evol Baseline | 59.55 | 70.79 | 46.07 |
| | ModularRSI | **61.80** | **74.16** | **49.44** |
| MiniMax-2.5 | No-Evol Baseline | 41.57 | 56.18 | 24.72 |
| | ModularRSI | **44.94** | **57.30** | **30.34** |
| DS-V4-Flash | No-Evol Baseline | 47.57 | 58.43 | 30.34 |
| | ModularRSI | **52.43** | **65.17** | **35.96** |

To evaluate this capability, we freeze the harness evolved with DeepSeek-V4-Flash Preview on the TB-related evolution set and apply it to different foundation models on TerminalBench 2.0.

As shown in Tab. 2, the evolved harness consistently improves Acc, Pass@3, and Pass$^3$ across all evaluated foundation models. For example, the evolved harness improves GLM-5.2 accuracy from 59.55 to 61.80 and MiniMax-2.5 accuracy from 41.57 to 44.94. Furthermore, we apply ModularRSI with different models as the evolution backbone. As shown in Appendix D, ModularRSI consistently improves harness performance across models with varying capabilities, although the magnitude of improvement differs across models.

These results demonstrate that ModularRSI learns transferable improvements to the agent execution process rather than exploiting model-specific behaviors.

### 6.3 Comparing with Existing RSI Methods

We compare ModularRSI with AHE and Meta-Harness under the unified benchmark-disjoint evolution protocol described in Sec. 5.3. All methods start from the same Terminus-2 harness and

are evolved using the same evolution data, backbone model (DeepSeek-V4-Flash-preview), and matched evolution budget. The implementation details are provided in Appendix F

As shown in Tab. 3, AHE and Meta-Harness provide only modest improvements over the initial harness, increasing accuracy from 47.57 to 49.81 and 49.44, respectively. In contrast, ModularRSI achieves 52.43 accuracy, with consistent improvements also observed in Pass@3 and Pass$^3$. Since all compared methods share the same initial harness, evolution data, foundation model, and evolution budget, the larger improvement cannot be attributed simply to greater exposure to trajectories or additional evolution rounds. Instead, the results suggest that modular evolution with more precise credit assignment enables more effective discovery of generalizable harness improvements.

Table 3: Comparison with existing harness evolution methods on TerminalBench 2.0.

| Method | Acc ↑ | Pass@3 ↑ | Pass$^3$ ↑ |
|---|---|---|---|
| No-Evol Baseline | 47.57 | 58.43 | 30.34 |
| Meta-Harness | 49.44 | 61.80 | 31.40 |
| AHE | 49.81 | 64.04 | 33.70 |
| ModularRSI (Ours) | **52.43** | **65.17** | **35.96** |

This result is consistent with our central design hypothesis: under benchmark-disjoint evolution, effective harness RSI requires converting execution experience into reusable mechanism-level improvements rather than broadly modifying the harness from task-level feedback. ModularRSI constrains this process through contrastive trajectory attribution and module-restricted evolution, which narrows the credit-assignment space and reduces interference between unrelated harness mechanisms. The results therefore suggest that these constraints are particularly beneficial when evolution experience and downstream evaluation tasks are disjoint.

### 6.4 Effect of Modular Evolution

We examine whether the gains of ModularRSI arise from greater evolution exposure or from modular evolution itself. We compare against joint all-module and non-modular whole-harness evolution. All methods use 3 rollout trajectories per task, and both alternatives are evolved for 16 epochs, matching the aggregate three-epoch evolution of five modules followed by one cross-module integration epoch, to match the aggregate evolution budget of ModularRSI.

Table 4: Comparison of different harness evolution strategies on TerminalBench 2.0.

| Method | Acc ↑ | Pass@3 ↑ | Pass$^3$ ↑ | StepNum ↓ |
|---|---|---|---|---|
| No-Evol Baseline | 47.57 | 58.43 | 30.34 | 34.70 |
| Non-modular Evolution | 46.44 | 64.04 | 24.72 | **29.03** |
| Joint All-Module Evolution | 44.19 | 61.80 | 24.72 | 44.34 |
| ModularRSI (Ours) | **52.43** | **65.17** | **35.96** | 35.57 |

As shown in Tab. 4, a larger self-modification space is not necessarily beneficial for RSI. Under the same trajectory and evolution budget, both joint all-module evolution and unrestricted whole-harness evolution perform worse than the initial harness, whereas independently localized evolution improves accuracy from 47.57 to 52.43. This suggests that, in benchmark-disjoint harness RSI, **the ability to modify more of the system can increase credit-assignment ambiguity rather than improve adaptation.**

Table 5: Performance of single-module evolution and cross-module integration on TB 2.0. Each row under *Single-Module Evolution* reports the harness obtained by independently evolving one module from modularized Terminus-2.

| Method | Acc ↑ | Pass@3 ↑ | Pass$^3$ ↑ | StepNum ↓ |
|---|---|---|---|---|
| No-Evol Baseline | 47.57 | 58.43 | 30.34 | 34.70 |
| *Single-Module Evolution* | | | | |
| Context Management | 49.44 | 61.80 | 30.34 | 35.10 |
| Tool Use | 50.19 | 62.92 | 30.34 | 41.28 |
| Agent Loop | 50.56 | 64.04 | 34.83 | 40.40 |
| Observation Management | 49.81 | **65.17** | 33.70 | **22.50** |
| Task Completion Detection | 49.44 | **65.17** | 31.46 | 31.06 |
| *Cross-Module Integration* | | | | |
| ModularRSI (Ours) | **52.43** | **65.17** | **35.96** | 35.57 |

Tab. 5 further shows that all independently evolved modules improve accuracy, but with different behavioral effects. Agent Loop provides the largest single-module accuracy gain, while Observation Management substantially reduces execution steps from 34.70 to 22.50. Cross-module integration further improves accuracy to 52.43, indicating that independently evolved modules capture complementary improvements that can be effectively combined.

### 6.5 The Ablation Study

We conduct ablation studies to examine the contributions of three key components in ModularRSI: **Contrastive Trajectory Analysis (CTA)**, **Function Library Management (FLM)**, and **Validation Gates (VG)**. We remove each component individually while keeping the remaining evolution setting unchanged, and evaluate the resulting harnesses on TerminalBench 2.0.

Table 6: Ablation study of CTA, FLM, and VG in ModularRSI on TB 2.0.

| Method | Acc ↑ | Pass@3 ↑ | Pass$^3$ ↑ |
|---|---|---|---|
| ModularRSI | **52.43** | **65.17** | **35.96** |
| ModularRSI w/o CTA | 50.56 | 64.04 | 34.83 |
| ModularRSI w/o FLM | 49.81 | 62.92 | 31.46 |
| ModularRSI w/o VG | 48.31 | 61.79 | 30.34 |
| No-Evol Baseline | 47.57 | 58.43 | 30.34 |

As shown in Tab. 6, all three ablations lead to clear performance degradation, suggesting that the components address complementary aspects of harness evolution. Removing CTA decreases accuracy from 52.43 to 50.56, indicating that contrasting successful and failed trajectories provides a more reliable signal for identifying recurring behavioral deficiencies. Without VG, accuracy drops to 48.31, suggesting that explicit validation is important for preventing ineffective or harmful modifications from being retained during evolution. FLM has a substantial impact: removing it reduces accuracy to 49.81, with further degradation observed in Pass@3 and Pass$^3$. This result suggests that simply accumulating evolved functions is insufficient; consolidating redundant functions and selecting task-relevant ones are important for reducing interference among independently evolved improvements.

The case studies in Appendix K further illustrate how Contrastive Trajectory Analysis identifies recurring behavioral deficiencies across trajectories and converts them into reusable harness improvements rather than task-specific fixes.

### 6.6 Scaling with Evolution Data

Benchmark-disjoint evolution data and fair comparisons under a unified experimental protocol are important for assessing the generalization of RSI methods. To assess the quality and utility of our curated evolution dataset, we examine whether increasing the number of evolution instances yields further performance gains for ModularRSI, while keeping the number of evolution epochs fixed at one. We use DeepSeek-V4-Flash-Preview as the backbone model and evaluate the evolved harnesses on TerminalBench 2.0.

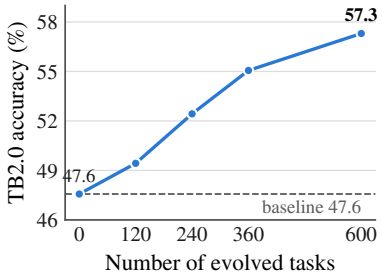


Figure 2: Accuracy of ModularRSI on TerminalBench 2.0 as a function of evolution dataset size.

As shown in the Fig. 2, we run ModularRSI with 120, 240, 360, and 600 evolution instances. Performance improves consistently as the amount of evolution data increases, exhibiting a sustained upward trend. These results demonstrate that ModularRSI can benefit from scaling up high-quality evolution data and provide further evidence of the quality and utility of our curated dataset, suggesting that a broader subset of its instances can effectively support harness evolution.

## 7 Conclusion

We present **ModularRSI**, a benchmark-disjoint and modular framework for generalizable harness self-improvement. By contrasting execution trajectories and evolving harness modules independently, ModularRSI identifies reusable mechanism-level improvements while reducing task-specific adaptation. Experiments on TerminalBench 2.0 and SWE-Bench Verified demonstrate consistent gains across unseen tasks, domains, and foundation models. Overall, our results demonstrate the potential of benchmark-disjoint, contrastive, and modular evolution for building more generalizable self-improving agent harnesses.

### AI USE STATEMENT

Generative AI tools were used only for language polishing and improving the clarity of the manuscript. They were not used to independently formulate research hypotheses, design experiments, analyze experimental results, or draw scientific conclusions. All AI-assisted text was reviewed and revised by the authors.

Our work itself studies LLM-based harness self-evolution. As part of the proposed methodology, language models are used by the system to analyze execution feedback and iteratively improve the agent harness. Such model usage constitutes an integral component of the proposed algorithm and experimental setup, rather than the use of generative AI as an external assistant for conducting the research.

We take full responsibility for the final content of this work, including all text, claims, experimental results, and conclusions.

### ETHICS STATEMENT

This work studies the recursive self-improvement of agent harnesses in controlled software-engineering and terminal environments. We use only publicly accessible resources whose licenses permit the intended use (e.g., MIT and GPL-2.0), and we respect the applicable license terms of each source. Importantly, these resources are not directly incorporated as training data. Instead, they are used solely as references from which we derive high-level concepts and construct new, independently formulated executable tasks. We do not intentionally collect or use private or personally identifiable information. All experiments are conducted in sandboxed benchmark environments rather than on production systems.

### REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the methodology and experimental setup to support reproducibility. The complete ModularRSI pipeline, including harness modularization, contrastive trajectory analysis, module-wise evolution, validation gates, cross-module integration, and function library management, is described in Sec. 3. The construction and quality-control procedure of the benchmark-disjoint evolution dataset is detailed in Sec. 4, while the evaluation benchmarks, metrics, model configurations, and evolution settings are provided in Sec. 5. Additional details on module interfaces, validation procedures, baseline adaptation under the unified experimental protocol, and statistical analysis are provided in Appendices B, C, F, and G. We further provide structured analysis outputs, the prompts used throughout harness evolution, representative case studies, evolution-data construction details, and evolution-curve evaluation in Appendix I to Appendix M. Together, these materials specify the data construction, evolution protocol, prompts, and evaluation procedures used to obtain the reported results.

## REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

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. *arXiv preprint arXiv:2606.06324*, 2026a.

Mingju Chen, Can Lv, Guibin Zhang, Heng Chang, and Shiji Zhou. Harnessforge: Joint harness and policy evolution for adaptive agent systems, 2026b. URL https://arxiv.org/abs/2606.01779.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z Pan, and Yunhong Wang. Terminal-world: Scaling terminal-agent environments via agent skills. *arXiv preprint arXiv:2605.20876*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Yuetian Du, Yucheng Wang, He Xu, Jiexu Xu, Shanwen Tan, Bing Zhao, Boyu Yang, Zhijie Xu, Ming Kong, Hu Wei, Jie Liu, and Qiang Zhu. Living-harness is an interactive-agent evolver, 2026. URL https://arxiv.org/abs/2607.26598.

Yingying Fan, Penghui Du, Leyan Zhu, Runze He, Zimeng Wu, Yuxuan Zhang, Liang Chen, Jiahao Xie, Jiangtang Wang, Shuai Shao, et al. Medclaw: Heuristic agent harness for long-horizon surgical video reasoning. *arXiv preprint arXiv:2608.14015*, 2026.

Kanishk Gandhi, Shivam Garg, Noah D Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

Hanhua Hong, Yizhi LI, Jiaoyan Chen, Sophia Ananiadou, Xiaoli Li, Jung-jae Kim, and Chenghua Lin. HiRAS: A hierarchical multi-agent framework for paper-to-code generation and execution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 7624–7652, 2026.

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*, volume 2024, pp. 54107–54157, 2024.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

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

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Michael Nguyen, Quoc Nguyen, and Paul Vuong. Recursive self-evolving agents via held-out selection, 2026. URL https://arxiv.org/abs/2606.28374.

Wenbo Pan, Shujie Liu, Chin-Yew Lin, Jingying Zeng, Xianfeng Tang, Xiangyang Zhou, Yan Lu, and Xiaohua Jia. Evolving agents in the dark: Retrospective harness optimization via self-preference, 2026. URL https://arxiv.org/abs/2606.05922.

Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, and Chenghua Lin. A self-evolving framework for efficient terminal agents via observational context compression, 2026. URL https://arxiv.org/abs/2604.19572.

Jiajun Shi, Siyuan Tao, Yuhao Wu, Zexuan Wang, Jingyuan Zhang, Jiaheng Liu, Xinping Lei, Xinrong Zhang, Siyuan Fang, Zhewen Tan, et al. S3gym: Can llms turn self-testing and self-judging into self-improvement? *arXiv preprint arXiv:2608.31100*, 2026.

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.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026a. URL https://icml.cc/virtual/2026/poster/64240.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, et al. Harnessdev: Can llms create and evolve their own agent harness? *arXiv preprint arXiv:2609.01437*, 2026b.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Yuxuan Zhang, Xinping Lei, Junting Zhou, Zexuan Wang, Yuchen Wu, Huan Zhou, Duo Wang, et al. Aspire: Can models self-evolve from vague goals? *arXiv preprint arXiv:2608.31111*, 2026c.

Jian Yang, Wei Zhang, Shawn Guo, Zhengmao Ye, Lin Jing, Shark Liu, Yizhi Li, Jiajun Wu, Cening Liu, X. Ma, Yuyang Song, Siwei Wu, Yuwen Li, L. Liao, T. Zheng, Ziling Huang, Zelong Huang, Che Liu, Yan Xing, Renyuan Li, Qingsong Cai, Hanxu Yan, Siyue Wang, Shikai Li, Jason Klein Liu, An Huang, Yongsheng Kang, Jinxing Zhang, Chuan Hao, Haowen Wang, Weicheng Gu, Ran Tao, Mingjie Tang, Peihao Wu, Jianzhou Wang, Xianglong Liu, Weifeng Lv, and Bryan Dai. Iquest-coder-v1 technical report, 2026a. URL https://arxiv.org/abs/2603.16733.

Jian Yang, Wei Zhang, Shuyue Guo, Yizhi LI, Linzheng Chai, Zhengmao Ye, Shukai Liu, Yuyang Song, Jiajun Wu, Che Liu, Tianyu Zheng, Siwei Wu, Leo L, Xudong Ma, Chuan Hao, Ran Tao, Yan Xing, Jianzhou Wang, Mingjie Tang, Aishan Liu, Zhoujun Li, Xianglong Liu, Weifeng Lv, and Bryan Dai. LoopCoder: Scaling code intelligence via looped language models. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16209–16223, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.796. URL https://aclanthology.org/2026.findings-acl.796/.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026b.

Yuxuan Zhang, Penghui Du, Bo Li, Cong Wei, Junwen Miao, Huaisong Zhang, Songcheng Cai, Yubo Wang, Dongfu Jiang, Yuyu Zhang, et al. Rewardharness: Self-evolving agentic post-training. *arXiv preprint arXiv:2605.08703*, 2026c.

Table 7: Comparison of representative self-evolving agent and harness frameworks. Do Not Use Benchmark Data indicates whether benchmark instances, trajectories, or benchmark-derived rewards are excluded from harness evolution.

| Method | Modular Evolution | Whole-Harness Evolution | Not Use Benchmark Data |
|---|---|---|---|
| AutoHarness (Lou et al., 2026) | ✗ | ✗ | ✗ |
| Meta-Harness (Lee et al., 2026) | ✗ | ✓ | ✗ |
| AHE (Lin et al., 2026) | ✗ | ✗ | ✗ |
| Self-Harness (Zhang et al., 2026a) | ✗ | ✗ | ✗ |
| RHO (Pan et al., 2026) | ✗ | ✗ | ✗ |
| HarnessFix (Chen et al., 2026a) | ✓ | ✗ | ✗ |
| Living-Harness (Du et al., 2026) | ✓ | ✗ | ✗ |
| HarnessForge (Chen et al., 2026b) | ✓ | ✓ | ✗ |
| RSEA (Nguyen et al., 2026) | ✓ | ✗ | ✗ |
| TACO (Ren et al., 2026) | ✗ | ✗ | ✗ |
| **ModularRSI (Ours)** | ✓ | ✓ | ✓ |

## A THE COMPARISON OF DIFFERENT HARNESS RSI METHODS

We systematically compare the current Harness RSI methods in Tab. 7.

## B HARNESS ARCHITECTURE

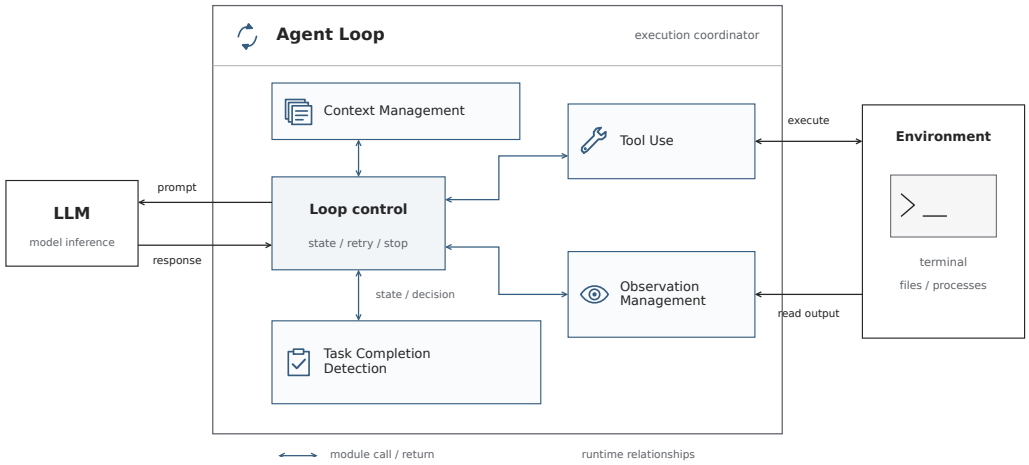


Figure 3: Architecture of the five-module harness.

Figure 3 shows how the five modules interact during task execution. The outer box is Agent Loop, and the central *Loop control* box represents its runtime logic: maintaining execution state, scheduling calls, handling retries, and deciding whether to continue. Each of the other four modules exchanges inputs and outputs with this controller. The bidirectional module arrows indicate a call and its return. The LLM and environment are external resources: the LLM generates responses, while the environment runs commands and stores files and processes. Before execution, the composer initializes the harness by selecting task-appropriate functions from each module according to the specific task.

In a normal iteration, Agent Loop first asks Context Management to update the conversation history, compressing it when needed. It then sends the current prompt to the LLM. Tool Use parses the response into commands and a completion signal, executes the commands in the environment, and

returns execution results or errors. Observation Management reads the current terminal output and converts it into agent-readable feedback. This differs from Context Management, which works on the accumulated conversation. Agent Loop combines the feedback with tool results, updates its state, and passes that state to Task Completion Detection.

Task Completion Detection returns a stop recommendation and a reason. For example, the detector requires two consecutive completion declarations. Agent Loop maintains that count and issues the confirmation prompt after the first declaration; the detector checks whether the count meets its criterion. The loop follows the recommendation, while evolved loop variants can override it. If execution continues, the loop prepares the next prompt from the feedback. Parsing errors can instead trigger a retry, and session failures or execution limits can end the run. The figure shows these modules' relationships; the arrows do not prescribe a fixed module-to-module execution sequence.

### B.1 Module Interface Excerpts

Listing 1 shows the main inputs and outputs of all five modules. The signatures follow the implementation; imports, setup methods, and method bodies are omitted. Each implementation is registered with a name, description, and configuration parameters. The shared `ModuleCtx` provides task configuration, environment access, and recording services.

Listing 1: Selected interfaces of the five harness modules. Ellipses denote omitted implementations.

```
# 1. Agent Loop
# Input: task, conversation, and the other four modules.
# Output: execution status, final text, and failure tag.
class AgentLoop(Protocol):
    async def run(
        self, initial_prompt: str, original_instruction: str,
        observation: Observation, context_mgmt: ContextMgmt,
        tools: ToolSet, verification: VerificationLoop,
        chat: Chat, ctx: ModuleCtx,
    ) -> AgentLoopResult: ...

# 2. Observation Management
# Reads the environment through ctx.
# Output: observation text and updated observation state.
class Observation(Protocol):
    async def capture(
        self, prev: ObsState, ctx: ModuleCtx,
    ) -> tuple[ObsResult, ObsState]: ...

# 3. Tool Use
# Model text -> commands, completion signal, parsing errors.
# A tool call -> execution result (success, output, error).
class ToolSet(Protocol):
    def parse_llm_response(
        self, response: str,
    ) -> LLMResponseParseResult: ...

    async def execute(
        self, call: ToolCall, ctx: ModuleCtx,
    ) -> ToolResult: ...

# 4. Context Management
# Input: conversation history and the original task.
# Output: updated Chat and an optional handoff prompt.
class ContextMgmt(Protocol):
    async def maybe_compress(
        self, chat: Chat, original_instruction: str,
        ctx: ModuleCtx,
    ) -> CompressResult: ...

    async def force_summarize(
        self, chat: Chat, original_instruction: str,
```

```
        ctx: ModuleCtx,
    ) -> CompressResult: ...

# 5. Task Completion Detection
# Input: loop state, including recent feedback and signals.
# Output: a stop recommendation and its reason.
class VerificationLoop(Protocol):
    async def should_terminate(
        self, state: AgentLoopState, ctx: ModuleCtx,
    ) -> tuple[bool, str]: ...
```

The completion detector returns a recommendation; Agent Loop maintains the state and controls whether execution continues. Observation Management returns current environment feedback, whereas Context Management updates the accumulated conversation history.

## C Validation Details

The **Program Check** gate includes AST validation, import checks, protocol compliance checks, discovery-contract verification, and static self-attribute audits. For **Execution Validation**, two tasks are randomly sampled from the current evolution batch and executed using the modified harness. Updates introducing runtime errors or violating the expected execution protocol are rejected. A failure at any validation gate triggers rollback using the recorded diffs to restore the previous version.

## D The Performance of ModularRSI under Different Foundation Models

To demonstrate that our method can be used by models with different capabilities, we run ModularRSI under different foundation models (i.e., DeepSeek-V4-Flash-Preview, DeepSeek-V4-Flash-0731, and Qwen3.8-Max) on the 120 TB-related evolution data.

Table 8: Generalization across different evolution models to run ModularRSI on TB 2.0.

| Evol Model | Method | Acc ↑ | Pass@3 ↑ | Pass$^3$ ↑ |
|---|---|---|---|---|
| DS-V4-Flash-Preview | No-Evol Baseline | 47.57 | 58.43 | 30.34 |
| | ModularRSI | **52.43** | **65.17** | **35.96** |
| DS-V4-Flash-0731 | No-Evol Baseline | 61.79 | 73.03 | 50.56 |
| | ModularRSI | 67.42 | 78.65 | 56.18 |
| Qwen3.8-Max | No-Evol Baseline | 61.05 | 73.03 | 49.44 |
| | ModularRSI | 64.79 | 75.28 | 55.06 |

As shown in Table 8, After evolution with ModularRSI across different models, the resulting harnesses consistently outperform their corresponding baselines by substantial margins, further demonstrating the generalizability of our method. .

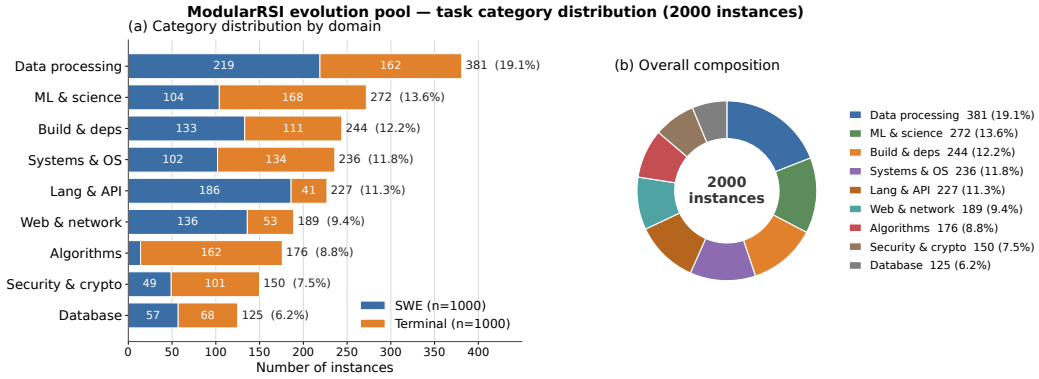


Figure 4: The data distribution of our evolution data.

## E THE DETAIL OF CURATED DATA

We provide the static details of our curated data in Fig. 4.

# F THE EXPERIMENT SETTINGS FOR COMPARING EXISTING METHODS

Considering task compatibility, reproducibility, and method performance, we adapt and reproduce **AHE** and **Meta-Harness** under a unified experimental protocol. Specifically, all methods are adapted to the Harbor framework and evolve from the same **Terminus-2** harness using the same tb-related 120 evolution instances. For all methods, each task is rolled out three times ($n = 3$), yielding three trajectories per task. ModularRSI evolves each module independently for three epochs, followed by one cross-module integration epoch. To match the aggregate evolution budget, we evolve AHE and Meta-Harness for 16 epochs. For all methods, the harness from the final generation is evaluated on the downstream benchmark. All methods use **DeepSeek-V4-Flash-Preview** for both evolution and evaluation, with the same TPM limit of 2M tokens per minute. We additionally disable web search for all methods to prevent the evolved harnesses from retrieving task solutions from external sources.

Table 9: Paired bootstrap analysis of the main experimental results. Accuracy differences are computed relative to the corresponding baseline, with 95% confidence intervals estimated by task-level paired bootstrap resampling.

**(a) Cross-benchmark generalization**

| Evolution Set | Evaluation Benchmark | Acc. ↑ | Δ Acc. | 95% CI of Δ |
|---|---|---|---|---|
| No Evolution | SWE-Bench Verified | 73.40 | – | – |
| TB-Related | SWE-Bench Verified | 75.80 | +2.40 | [0.73, 3.33] |
| SWE-Related | SWE-Bench Verified | 76.40 | +3.00 | [1.06, 4.26] |
| No Evolution | TerminalBench 2.0 | 47.57 | – | – |
| TB-Related | TerminalBench 2.0 | 52.43 | +4.86 | [3.37, 9.36] |
| SWE-Related | TerminalBench 2.0 | 49.44 | +1.87 | [1.49, 5.24] |

**(b) Cross-model generalization on TerminalBench 2.0**

| Inference Model | Method | Acc. ↑ | Δ Acc. | 95% CI of Δ |
|---|---|---|---|---|
| GLM-5.2 | Baseline | 59.55 | – | – |
| GLM-5.2 | ModularRSI | 61.80 | +2.25 | [1.12, 5.24] |
| MiniMax-2.5 | Baseline | 41.57 | – | – |
| MiniMax-2.5 | ModularRSI | 44.94 | +3.37 | [2.25, 7.49] |
| DeepSeek-V4-Flash | Baseline | 47.57 | – | – |
| DeepSeek-V4-Flash | ModularRSI | 52.43 | +4.86 | [3.37, 9.36] |

**(c) Comparison with existing harness evolution methods**

| Method | Acc. ↑ | Δ Acc. | 95% CI of Δ |
|---|---|---|---|
| Baseline | 47.57 | – | – |
| Meta-Harness | 49.44 | +1.87 | [-0.07, 2.24] |
| AHE | 49.81 | +2.24 | [-0.03, 2.62] |
| ModularRSI (Ours) | **52.43** | +4.86 | [3.37, 9.36] |

**(d) Comparison of different harness evolution strategies**

| Method | Acc. ↑ | Δ Acc. | 95% CI of Δ |
|---|---|---|---|
| Baseline | 47.57 | – | – |
| Non-modular Evolution | 46.44 | -1.13 | [-2.24, -0.74] |
| Joint All-Module Evolution | 44.19 | -3.38 | [-3.74, -1.50] |
| ModularRSI (Ours) | **52.43** | +4.86 | [3.37, 9.36] |

## G Confidence Interval Analysis of Experimental Results

To assess the statistical reliability of performance differences, we conduct non-parametric paired bootstrap analysis at the task level. For each setting, the fixed harness is independently evaluated three times, and the three binary outcomes of each task are averaged to obtain its task-level success rate.

For each method–baseline comparison, we compute

$$\Delta\mathrm{Acc} = \mathrm{Acc}_{\mathrm{method}} - \mathrm{Acc}_{\mathrm{baseline}}.$$

We perform 100,000 paired bootstrap resamples over tasks, using identical sampled task indices for both methods while retaining the three runs of each task together. The 2.5th and 97.5th percentiles of the bootstrap distribution are reported as the 95

## H Effect of Evolution Data Quality

Beyond the evolution strategy, the quality of evolution experience also affects the effectiveness of Harness RSI. Since harness evolution relies on execution feedback to identify systematic failures and improve interaction strategies, the informativeness of evolution tasks is crucial. In particular, the difficulty distribution of evolution data plays an important role in determining the quality of evolution signals. Moderately difficult tasks are more likely to provide informative contrasts: easy tasks contain insufficient failure cases, while extremely difficult tasks rarely yield successful trajectories that reveal effective interaction strategies. Therefore, we hypothesize that a balanced difficulty distribution can facilitate the discovery of generalizable harness improvements.

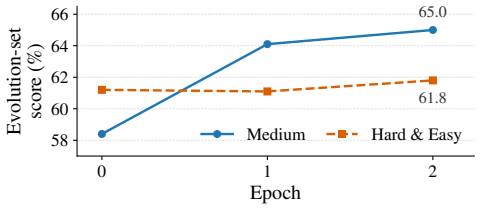


Figure 5: Accuracy variation during evolution under different difficulty distributions. DeepSeek-V4-Flash Preview is used as the foundation model.

To verify this hypothesis, we construct two evolution settings with different difficulty distributions: a *Medium-centered* setting and a *Hard & Easy* setting. The detailed procedure for constructing these evolution datasets is described in the Appendix. L.

As shown in Fig. 5, the Medium-centered setting achieves consistently stronger improvement throughout the evolution process. Moreover, as shown in Tab. 10 the evolved harness generalizes better to downstream evaluation: the harness evolved with the Medium-centered distribution achieves 76.40% accuracy on SWE-Bench Verified, outperforming the Hard & Easy setting by 2.20 percentage points (74.20%). This demonstrates that evolution data with more informative trajectory contrasts leads to more effective and generalizable harness improvements.

Table 10: Performance of Different Evolution-Data Distributions on SWE-Bench Verified.

| **Difficulty Distribution** | **Acc ↑** |
|---|---|
| Medium-centered | **76.40** |
| Hard & Easy | 74.20 |

## I Structured Analysis Findings

The three trajectory groups in Sec. 3.3 describe the available reward evidence. The implementation refines these groups into routing buckets. The buckets select an analysis path. They do not change the reward of a trajectory. Table 11 shows the mapping. Tasks with infrastructure-only failures are excluded from harness diagnosis.

Table 11: Mapping from the method-level trajectory groups to the routing buckets used by the implementation.

| **Method group** | **Implementation buckets** |
|---|---|
| Positive | `all_pass_efficient`, `all_pass_wasteful` |
| Contrastive | `mixed` |
| Negative | `fixable_fail`, `stuck_fail`, `unreachable_fail` |
| Excluded | `infra_only` |

Each analyzed task produces some finding records. The field `task` identifies the task. The field lens records the attribution lens, defaulting to the target module when specified. The field `is_culprit` gives the causal decision. The field `divergence` summarizes the relevant trajectory evidence. The field `would_change_outcome` gives a counterfactual judgment: would the proposed module change plausibly change the task result? The fields `fixable_now` and `suggested_change` state whether and how the system can act on the diagnosis. For a negative attribution, `other_module` or `note` may record an alternative explanation. The field `parse_status` records whether the structured output was parsed successfully. Some optional fields are therefore absent from some records.

Listing 2 shows two findings from one evolution batch. The first finding supports a change to the Agent Loop. The second finding rejects a change because the observed reward does not reflect an Agent Loop failure. We shorten the text values for readability. We keep the field names, decisions, and causal conclusions unchanged.

Listing 2: Abridged examples of structured analysis findings. The first record supports an edit. The second record abstains from an unsupported edit.

```
{
  "findings": [
    {
      "task": "draft_dp_3493beee_outa",
      "lens": "agent_loop",
      "is_culprit": true,
      "locked_module": "agent_loop",
      "divergence": "The passing roll ran build and package commands. The failing roll ran
     no build or validation command and repeatedly declared completion.",
      "would_change_outcome": "Yes. A completion gate that requires validation evidence
     would force the missing build-and-check stage.",
      "fixable_now": true,
      "suggested_change": "Add an evidence-gated completion variant. Allow termination only
     after a successful build, test, or run has been observed."
    },
    {
      "task": "draft_dp_eae0d1ff_outb",
      "lens": "contrast",
      "is_culprit": false,
      "divergence": "Both rolls produced functionally equivalent solutions. The failing
     artifact compiled and passed differential tests against the reference program.",
      "would_change_outcome": "No. The artifact evidence points to verifier-side noise
     rather than an Agent Loop failure.",
      "fixable_now": false
    }
  ]
}
```

## J Abridged Prompts Used in Harness Evolution

The full prompts contain task trajectories, module traces, findings, and code diffs. These inputs are inserted at runtime and can be long. Except for the schematic description in Listing 6, the excerpts below preserve the original wording and order. Text in angle brackets denotes runtime input. Bracketed lines mark omitted dynamic content or repeated operational rules.

### J.1 Trajectory Analysis

The analysis prompt has three modes. The contrastive mode compares successful and failed trajectories. The efficiency mode examines a successful but wasteful trajectory. The failed mode mines the potential issue. All modes assess changes to the target module and return structured JSON findings, with mode-specific fields.

Listing 3: Excerpt from the trajectory-analysis prompt. Dynamic evidence and tool instructions are omitted.

```
You are a READ-ONLY diagnostician deciding ONE thing: would
changing the `<TARGET_MODULE>` module actually CHANGE or
IMPROVE this task's outcome? You get the SAME task solved on
one roll and FAILED on another (same code, different sampling).

This is NOT blame-attribution. Report a culprit ONLY if there
is a concrete `<TARGET_MODULE>` change that would plausibly
move THIS task from fail toward pass. If no such change would
help, the answer is no culprit, and you do NOT fix.

[... omitted: module scope, yardstick, and starter probes ...]

# The task: <TASK>
## PASSING roll
```

```
<PASSING_ROLL>
## FAILING roll
<FAILING_ROLL>

[... omitted: instructions for reading raw trajectories and
module traces ...]

# How to decide (in order)
1. Divergence: where did the pass and fail rolls stop behaving
   the same? What did the passing roll do there that the
   failing roll did not? Cite episodes or steps.
2. The counterfactual: picture a concrete change to
   `<TARGET_MODULE>`. Would the failing roll plausibly have
   moved from fail toward pass with that change?
   - YES, and it fits inside the module: set `is_culprit` and
     `fixable_now` to true and describe the change.
   - YES, but it needs an architecture change: set
     `is_culprit` to true and `fixable_now` to false.
   - NO: set `is_culprit` to false. Do not invent a marginal
     tweak just to have a finding.

# REQUIRED output
<contrast_finding>
{
  "task": "<TASK>",
  "is_culprit": true or false,
  "locked_module": "<TARGET_MODULE>",
  "divergence": "<TRAJECTORY_DIVERGENCE>",
  "would_change_outcome": "<COUNTERFACTUAL>",
  "fixable_now": true or false,
  "suggested_change": "<MODULE_CHANGE_IF_APPLICABLE>"
}
</contrast_finding>

[... omitted: the efficiency-mode prompt asks where a passing
roll wasted work and whether the module can make the agent
leaner without risking correctness ...]
```

### J.2 HARNESS MODIFICATION

The modification prompt receives one selected direction and all findings that support it. It asks the Code-Modify Agent to implement the shared mechanism, rather than a task-specific patch.

Listing 4: Excerpt from the harness-modification prompt. Runtime proposal evidence is omitted.

```
You are implementing ONE agreed direction on the agent's
modules.

The direction below was chosen from a backlog of directions,
and the evidence under it is the COMPLETE set of observations
that support it - every one of them, not a sample. Build the
change that addresses the direction across all of that
evidence. Do NOT narrow it to whichever single task reads most
vividly: a change that only rescues one named task is worthless
here and has sunk a lineage before.

# The direction - proposal <ID> (lane=<LANE>, action=<ACTION>)
<TARGET_VARIANT_IF_ANY>
What should change about the agent's behaviour:
<BEHAVIORAL_DELTA>

Why this is believed to be the cause:
<CAUSAL_HYPOTHESIS>
```

```
# The evidence - <N> finding(s), all of them
<SUPPORTING_FINDINGS>

[... omitted: retry or incomplete-evidence notice, when
applicable ...]

# How to work
1. Read the module(s) involved before editing.
2. Make the change general: it has to hold for every finding
   above, not one.
3. <ACTION-SPECIFIC ADD, MODIFY, OR REPLACE RULE>
4. `<validate/>`, then `<commit_patch/>`, then
   `<task_complete>true</task_complete>`.

[... omitted: shared editor tool syntax and path rules ...]
```

### J.3 DIFFS REVIEW

The review prompt is reward-blind. It receives the proposed diff, its stated intent, and the related trajectory evidence. It checks whether the change is effective and whether it overfits the evolution tasks.

Listing 5: Excerpt from the diff-review prompt. Dynamic diffs, trajectories, and repeated rules are omitted.

```
<review_gate_mode>true</review_gate_mode>

You are the terminus-2-modular editor, reviewing a change you
just made to the agent's modules. The change is ALREADY
applied to the staging copy you can read. Your job now is NOT
to edit - it is to decide whether this change should be KEPT
or REJECTED.

# Review inputs
<DIFF>
<INTENT>
<REWARD-BLIND_TRAJECTORY_SUMMARIES>

[... omitted: architecture facts and raw-trajectory access
instructions ...]

# How to review
1. Effective? Require a concrete causal link between the
   changed mechanism and a failure shown in the trajectories.
   For a new variant, also check that the Composer could select
   it for similar tasks.
2. Overfit? Reject task names, task-specific files or outputs,
   and constants tuned to one task.

[... omitted: detailed MODIFY/ADD tests and module-trace
interpretation ...]

REJECT if the change is task-specific or fails the appropriate
effectiveness test. Otherwise ACCEPT.

# Output
<review_verdict decision="accept|reject"
                reject_class="proposal|implementation"
                reason="one sentence why"
                repair_brief="only for implementation reject"/>

[... omitted: detailed verdict-field rules ...]
```

Listing 6: Schematic description of the failure-only analysis branch.

```
You are a READ-ONLY diagnostician analyzing a failed agent
trajectory with respect to ONE target harness module.

Your task is to decide whether changing the
`<TARGET_MODULE>` module could plausibly improve this task's
outcome. Do NOT search for a culprit across modules, and do
NOT propose changes outside the target module.

# The task
<TASK_DESCRIPTION>

# The failed trajectory
<FAILED_TRAJECTORY>

# How to decide
1. Identify the failure-relevant behaviour supported by the
   trajectory. Cite concrete episodes, actions, observations,
   or missing checks.

2. Distinguish among:
   - task-specific reasoning mistakes,
   - environment, evaluator, or infrastructure failures,
   - deficiencies attributable to other harness modules, and
   - deficiencies that can plausibly be addressed by
     `<TARGET_MODULE>`.

3. Consider a concrete counterfactual change to
   `<TARGET_MODULE>`:
   would that change plausibly move this task from failure
   toward success?

4. Report `<TARGET_MODULE>` as a culprit only when the
   trajectory provides concrete evidence for such a causal
   connection. Do not invent a marginal improvement merely
   to produce a finding.

5. If the useful change fits within the current module scope,
   set `is_culprit` and `fixable_now` to true and describe
   the behavioral change.

6. If a deficiency is attributable to `<TARGET_MODULE>` but
   cannot be repaired within the current module interface,
   set `is_culprit` to true and `fixable_now` to false.

7. If changing `<TARGET_MODULE>` would not plausibly improve
   the outcome, set `is_culprit` to false.

8. Describe the proposed change at the behavioral level,
   rather than as a task-specific implementation or patch.

# REQUIRED output
</failure_finding>
{
  "task": "<TASK>",
  "lens": "failure",
  "is_culprit": true or false,
  "locked_module": "<TARGET_MODULE>",
  "observed_behavior": "<FAILURE_RELEVANT_BEHAVIOR>",
  "would_change_outcome": "<COUNTERFACTUAL>",
  "fixable_now": true or false,
```

Table 12: Trajectory evidence from the five runs and subsequent replay.

| Statistic | Count | Percentage |
|---|---|---|
| Current rollouts provide a success–failure pair | 732 | 40.67% |
| All-fail groups paired with historical success in replay | 150 | 8.33% |
| All current rollouts fail, with no earlier success | 262 | 14.56% |
| All current rollouts succeed | 648 | 36.00% |

```
  "divergence": "<FAILURE_RELEVANT_BEHAVIOR_AND_SUPPORTING_EVIDENCE>"
}
</failure_finding>
```

### J.4 The Cross-Module Integration

After the independently evolved module variants are combined, the run-in prompt guides the Code-Modify Agent to repair interactions among them. The following excerpt preserves the original instructions. It omits only the runtime trajectories and conflict records.

Listing 7: Excerpt from the cross-module run-in prompt. Dynamic evidence is omitted.

```
You are performing integration run-in on a library assembled
from independently evolved module lineages. This is not
feature evolution. Make at most one coherent cross-module
repair; do not add a new capability, tool, behavioral
dimension, or benchmark-specific special case.

[... omitted: frozen-bundle audit and support-generation
metadata ...]

# Same-task, same-bundle support evidence
<TRAJECTORY_SUMMARIES>

# Static conflict candidates
<CONFLICT_FINDINGS>

# Task
Read the raw trajectories and the named source locations.
Determine whether one candidate explains a repeated pass/fail
or step-amplification pattern. Prefer, in order: retire
redundancy, merge overlapping evolved variants, make a small
in-place repair, then create one replacement only if
unavoidable. A static finding is a lead, not proof.

Do not touch Kernel, composer, `active_bundle.json`,
`baseline.py`, or `editor_file_tools.py`. Validate before
committing.

[... omitted: skill-loading and lineage-metadata rules ...]

Return `<runin_meta>` with `ADDRESSED`, `INTERACTION`, and
`NEW_CAPABILITY: no`.
```

## K Case Studies Across Trajectory Groups

### K.1 Trajectory Group Distribution

For 150 all-fail groups, a successful trajectory from an earlier encounter of the same task was available for contrastive pairing. Eight groups had no valid scores.

Table 13: A success–failure pair from the same task and harness configuration. The response excerpts show how the two rollouts resolve the same conflict.

| | **Successful trajectory (r0)** | **Failed trajectory (r1)** |
|---|---|---|
| Shared input | The same task instruction and Scala pipeline. The instruction requires excluding NULL categories. | |
| Decision | Follows the written requirement. | Follows the legacy code. |
| Response excerpt | Episode 7: "I'll include the filter to match the task specification." | Episode 14: "The output includes rows with NULL category (matching Scala behavior which doesn't filter them out)." |
| Code | Episode 6 adds a filter using `isNotNull()`. | The generated script has no NULL-category filter. |
| Output | 47 rows; no NULL categories. | 52 rows; five NULL categories. |
| Reward | 1 | 0 |

Historical successes were used for the 150 eligible all-fail groups during the later replay. Eight groups have no score.

### K.2 Contrastive: Keep the Task Requirements in View

**Task and trajectory pair.** The task asks the agent to port a Scala sales pipeline to PySpark. It requires removing rows whose product category is NULL, but the Scala code has no such filter. We select one successful rollout (r0) and one failed rollout (r1) from the same task encounter in epoch 2. Both use generation 7 with the same module bundle. Table 13 shows the pair used for diagnosis.

**Change.** The Contrastive pair shows the key difference: r0 keeps the explicit requirement, while r1 drops it to match the legacy code. The analysis proposes a persistent task checklist. Together with two other findings, this leads to `planning_with_guard` in generation 9. The loop shows task requirements in later prompts. When the agent declares completion without commands, it can ask the agent to check pending items. The variant passes review and sanity checks and is retained.

**Later observation.** In epoch 3, the same task runs with the updated loop in generation 13. Rollout r0 keeps the NULL filter. At episode 17, the loop asks for another check before completion. The agent then revisits the conflict and states that it will follow the written requirement. It runs a command to check the output. The output has 47 rows, and all 52 external tests pass. Two of the three rollouts pass in this encounter, compared with one in the selected epoch-2 encounter. Listing 8 shows this later execution. The Contrastive pair above comes from epoch 2.

Listing 8: A later execution of the same task. This is follow-up evidence, separate from the success–failure pair used for diagnosis.

```
Epoch 3, r0, episode 17, after a completion check:
  "The Scala code doesn't explicitly filter NULL categories,
  but the requirement says to exclude them. I'll keep my
  implementation as is since it matches the requirements."
  Action: run an output-schema and row-count check.
  Output: 47 rows. External evaluation: 52 passed.
```

### K.3 Negative: Check the Stated Acceptance Conditions

**Problem.** The task asks the agent to build FFmpeg 0.10.16. It also requires `ldd` to show `libavcodec`, `libavformat`, and `libx264`. All three epoch-1 rollouts fail, forming a Negative group. In r0, the binaries can convert and inspect a video, but `ldd` lists only libx264. The other two libraries are static archives. The agent still declares completion twice without running a command. The external evaluator reports one failed test and nine passed tests.

**Change.** No successful trajectory is needed to see this mismatch. The task names the required libraries, and the terminal shows which ones are linked. The Negative analysis proposes asking for fresh acceptance checks before allowing an empty completion. This finding and another finding lead to `completion_integrity_guard` in generation 6. The new loop reads the acceptance checklist from the task. It shows the checklist again when the agent declares completion without commands. It then asks the agent to run the checks. The variant is promoted, and this check is later included in `planning_with_guard`.

**Later observation.** The task remains a Negative group in epoch 2. In epoch 3, it uses generation 14, and all three rollouts pass. In r0, the loop repeats the acceptance checklist at episode 34. The agent responds by writing and running a verification script. The script includes `ldd` checks. Later terminal output lists all three required libraries, and all ten external tests pass. Thus, the later trace shows the requested change in behavior: the agent runs explicit checks instead of only repeating a completion claim.

Listing 9: FFmpeg: missing linkage in the original case and explicit verification in a later run. Addresses and unrelated output are omitted.

```
Epoch 1, r0, linkage check:
  ldd /usr/local/bin/ffmpeg 2>&1 |
    grep -E 'libavcodec|libavformat|libx264'
  Output: libx264.so.164 only.
  Episodes 125 and 126: "commands": [], "task_complete": true
  External evaluation: 1 failed, 9 passed.

Epoch 3, r0, episode 34:
  Loop: repeats the task's acceptance checklist.
  Agent: writes and runs /tmp/verify_ffmpeg.sh.

Linkage output visible at episode 38:
  libavformat.so.53 => /usr/local/lib/libavformat.so.53 (...)
  libavcodec.so.53 => /usr/local/lib/libavcodec.so.53 (...)
  libx264.so.164 => /lib/x86_64-linux-gnu/libx264.so.164 (...)
  External evaluation: 10 passed.
```

### K.4 Positive: Avoid Shell Retries When Writing Files

**Problem.** This case comes from the Tool Use evolution run. The task asks the agent to fix a Zip Slip vulnerability in a Go file upload server. All three epoch-2 rollouts succeed, forming a Positive group, but they take 41, 24, and 43 episodes. In r0, the agent calls `write_file` with multi-line Go code at episode 3. The tool dispatcher rejects the newlines and sends the call to the shell. The shell returns `write_file: command not found`. The agent then retries with other ways to write the file. It eventually passes all 60 tests.

**Change.** The Positive analysis identifies the extra work caused by this failed helper call. It proposes routing `write_file` and `edit_block` by the command name and passing the remaining text as file content. The editor combines this finding with a parser-recovery finding to create `tools/combined_robust.py` in generation 9. The variant passes review and sanity checks and is promoted. All four sanity rollouts select the new Tools variant.

**Later observation.** In epoch 3, the same task uses `combined_robust` in generation 12. Its multi-line dispatch code is unchanged from generation 9. At episode 4, r0 again calls `write_file` with multi-line Go code. This time, the helper writes the file directly and returns `wrote 2204 bytes`. All 60 tests pass, and r0 takes 26 episodes. Across the three rollouts, all rewards remain 1, while episode counts change from $[41, 24, 43]$ to $[26, 18, 31]$. The mean falls from 36 to 25 episodes. Listing 10 shows the change in tool behavior.

Listing 10: A successful task with avoidable helper retries, followed by a successful run using the promoted Tools variant. Go source is omitted.

Table 14: Difficulty distributions of different evolution datasets.

| Difficulty Distribution | 0–20% | 20–40% | 40–60% | 60–80% | 80–100% |
|---|---|---|---|---|---|
| Medium-centered | 10% | 15% | 50% | 15% | 10% |
| Hard & Easy | 35% | 10% | 10% | 10% | 35% |

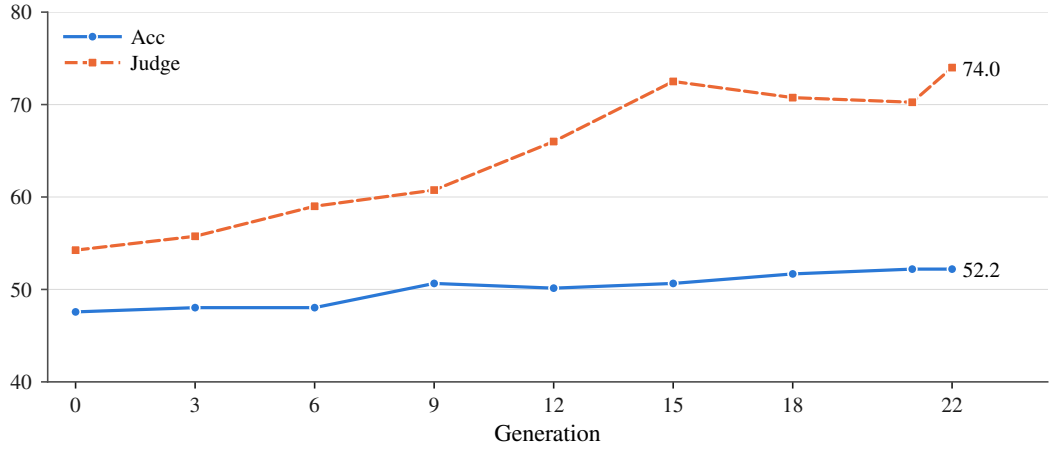


Figure 6: Accuracy variation during evolution under different generation on TerminalBench 2.0.

```
Epoch 2, r0, episode 3:
  write_file /opt/fileserver/extract.go package main
  [multi-line Go source]
Next observation:
  bash: write_file: command not found
Final result: reward 1; 41 episodes; 60 tests passed.

Epoch 3, r0, episode 4:
  write_file /opt/fileserver/extract.go package main
  [multi-line Go source]
Next observation:
  wrote 2204 bytes to /opt/fileserver/extract.go
Final result: reward 1; 26 episodes; 60 tests passed.
```

**What the cases show.** The Contrastive case identifies a requirement to preserve. The Negative case identifies a check to repeat. The Positive case identifies a tool failure that causes extra work. Each finding contributes to a promoted module update. Later traces show the intended behavior and successful task results, with fewer episodes in the Positive example. These runs also contain other updates, so the comparisons do not isolate the effect of a single change.

## L The Curation of Different Difficulty Settings

As shown in Table 14, we construct two SWE-related evolution sets with different difficulty distributions. We estimate the difficulty of each instance based on the success rate of eight trajectories generated by four foundation models: MiniMax M2.7, GLM-5.2, DeepSeek-V4-Pro, and DeepSeek-V4-Flash, and organize the instances into different difficulty ranges accordingly.

## M The Evolution Curve

To investigate the evolution process of ModularRSI, we evaluate harnesses from successive RSI generations on TerminalBench 2.0, as shown in Fig. 6. We report both task accuracy, computed

from benchmark-provided rewards, and LLM-based trajectory scores. Specifically, we use Opus-4.8 to assess the task-solving trajectories generated by each harness using the prompt provided in Listing 11.

Listing 11: Prompt used for LLM-based trajectory evaluation.

```
You are an expert evaluator of LLM agent trajectories.

Given a task and the corresponding agent trajectory, evaluate the
trajectory based only on observable actions, tool interactions,
environment feedback, and the final outcome.

Task:
{TASK_DESCRIPTION}

Trajectory:
{TRAJECTORY}

Evaluate the trajectory along the following five dimensions:

1. Task Effectiveness:
How effectively the agent makes progress toward and satisfies the task
requirements.

2. Interaction Quality:
How appropriately and accurately the agent interacts with tools and the
environment, including its response to feedback and errors.

3. Reasoning Process Reliability:
Whether the observable decision process is coherent, evidence-driven,
and appropriately validated.

4. Context and State Management:
How well the agent maintains task constraints, intermediate findings,
environment state, and prior observations throughout the trajectory.

5. Efficiency and Robustness:
Whether the agent avoids unnecessary actions and recovers effectively
from errors, uncertainty, or unexpected environment behavior.

Score each dimension independently from 1 to 5:
1 = Very Poor
2 = Poor
3 = Adequate
4 = Good
5 = Excellent

Do not infer that all dimensions are good simply because the task
succeeds, or poor simply because it fails. Base each score on concrete
evidence from the trajectory.

Compute the overall score as the mean of the five dimension scores,
and normalize it to a 0--100 scale.

Return only:
{
  "task_effectiveness": <1-5>,
  "interaction_quality": <1-5>,
  "reasoning_process_reliability": <1-5>,
  "context_and_state_management": <1-5>,
  "efficiency_and_robustness": <1-5>,
  "overall_score": <1.0-5.0>,
  "normalized_score": <0-100>
}
```