FAIL2TASK: REPRODUCING CODING-AGENT FAILURE PATTERNS THROUGH EXECUTABLE TASK SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Failed coding-agent trajectories provide evidence of weaknesses encountered in practice. However, when the original repositories and execution environments are unavailable, these failures cannot be directly replayed for evaluation or agent improvement. We study failure-to-task reconstruction: turning such trajectories into new executable tasks rather than recovering the original tasks. We introduce FAIL2TASK, a closed-loop framework whose goal is to reproduce the source failure pattern when the same agent attempts the constructed task. Evidence-supported failure patterns guide repository selection and task construction, producing tasks with runnable environments, tests, and reference solutions. Execution-based validation and semantic comparison of source and replay trajectories guide iterative refinement. We evaluate FAIL2TASK on 96 source failures from SWE-bench Verified and Terminal-Bench 2.0, producing tasks accepted by the pipeline for all 96 cases. Rerunning the agents on these tasks yields trajectories with 80.18% failure-pattern alignment with the original failure trajectories. Harness self-evolution on the constructed tasks achieves source-task pass rates of 27.1–31.3% on SWE-bench, compared with 14.6–20.8% for the strongest baseline. On Terminal-Bench, the corresponding rates are 16.7–20.8%, compared with 4.2–12.5% for the baselines. These results show that tasks constructed from failure trajectories can help agents solve previously failed tasks.

1 INTRODUCTION

Coding agents are increasingly evaluated on real-world software engineering tasks (Yang et al., 2024). Analyses of agent failures can identify recurring weaknesses and inform subsequent improvement (Zhu et al., 2025). Failed execution trajectories record agents' decisions, tool interactions, and execution feedback, providing evidence for diagnosing these weaknesses (Li et al., 2026). Users may be willing to share these trajectories with agent developers to help improve their products. However, they may prefer not to provide access to the underlying repositories because these may contain private or sensitive information. We therefore consider a setting in which a user-contributed failure trajectory is available, but the original repository and execution environment are not.

Executable repository tasks support agent training (Pan et al., 2025) and feedback-driven harness improvement (Park et al., 2026). Without the original repository and execution environment, the observed failures cannot be directly replayed. This access restriction prevents developers from directly using these failures as executable benchmark instances for regression evaluation or as training tasks for agent self-evolution, even when users explicitly authorize the use of their trajectories for these purposes. One possible approach is to reconstruct the original task and its environment, but the trajectory may not contain all the information needed to do so. We therefore ask *whether an evidence-supported failure pattern can be instantiated in a new executable task on a public repository.*

To this end, we introduce FAIL2TASK, a framework that turns coding-agent failure trajectories into executable tasks in public repositories. Its goal is to reproduce the source failure pattern when the same agent attempts these tasks. Given a source trajectory, FAIL2TASK identifies an evidence-supported failure pattern. This pattern guides the selection of a compatible public repository and the construction of a new task with a runnable environment, private tests, and a reference solution. Execution-based validation checks that the task fails the tests in its initial state but passes after

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.

applying the reference solution. For semantic assessment, FAIL2TASK runs the coding agent that produced the source failure trajectory on the constructed task. A semantic judge compares the two trajectories to assess whether the target failure pattern recurs. Feedback from both checks guides task refinement until both checks pass or the attempt limit is reached.

We evaluate FAIL2TASK on 48 failure instances from SWE-bench Verified (Jimenez et al., 2024) and 48 from Terminal-Bench 2.0 (Merrill et al., 2026). All 96 source instances yield tasks accepted by the pipeline within a budget of 15 construction attempts per instance. We rerun the agent on the constructed tasks to obtain new execution trajectories. These trajectories show 80.18% failure-pattern alignment with the original failure trajectories. For the downstream study, we use 48 source failure cases from each benchmark. Harness self-evolution on the corresponding constructed tasks achieves source-task pass rates of 27.1–31.3% on SWE-bench, compared with 14.6–20.8% for the strongest baseline. On Terminal-Bench, the corresponding rates are 16.7–20.8%, compared with 4.2–12.5% for the baselines.

Our contributions are threefold:

- We formulate failure-to-task reconstruction: constructing new executable tasks from failed coding-agent trajectories without access to the original repositories or execution environments. The objective is to reproduce evidence-supported failure patterns rather than recover the original tasks.
- We introduce FAIL2TASK, a closed-loop framework that uses source failure patterns to guide task construction. It combines execution-based validation with semantic assessment of agent replays to iteratively refine the constructed tasks.
- We demonstrate failure-pattern reproduction across two benchmark sources and downstream benefits for harness self-evolution. Evolution on the constructed tasks achieves higher source-task pass rates than baselines using tasks selected from existing datasets based on the source failure trajectories.

2 RELATED WORK

Agent Failure Diagnosis and Attribution. Who&When (Zhang et al., 2025b) benchmarks attribution of multi-agent failures to responsible agents and failure steps. AgenTracer (Zhang et al., 2025a) develops automated tracing for such attribution. AgentDebug, AgentRx, and AGENTSCOPE (Zhu et al., 2025; Barke et al., 2026; Bi et al., 2026) structure diagnosis through error taxonomies, constraint checks, and behavioral abstractions, respectively. CodeTracer (Li et al., 2026) reconstructs hierarchical state-transition traces from heterogeneous run artifacts and localizes failure origins and their downstream error chains. FAIL2TASK identifies evidence-supported failure patterns from source trajectories and uses them to guide task construction. It then uses agent replay and semantic assessment to check whether the source failure pattern recurs in the constructed task.

Executable Task and Environment Construction. SWE-bench Verified (Jimenez et al., 2024) and Terminal-Bench (Merrill et al., 2026) pair repository-level and terminal-based tasks with executable evaluation. Task construction ranges from collecting existing tasks to synthesizing new tasks and environments. SWE-ReBench (Badertdinov et al., 2025) collects GitHub issue-resolution tasks, while SWE-smith (Yang et al., 2025) synthesizes tasks by introducing test-breaking modifications into existing repositories. For terminal tasks, Terminal-Lego (Yang et al., 2026) converts real-world technical issues into executable tasks, and SETA (Shen et al., 2026) combines environment synthesis with difficulty- and diversity-controlled evolution. CLI-Universe (Hua et al., 2026) synthesizes terminal tasks through capability-taxonomy sampling and evidence-guided research over real-world technical materials, with multi-stage executable verification in Dockerized environments. Terminal-Universe (Wu et al., 2026) reconstructs task-sufficient workspaces from recorded file operations and agentic completion, then generates tasks for agent training. TRACE (Kang et al., 2026) contrasts successful and failed trajectories to identify capability gaps and synthesize targeted training environments. Socratic-SWE (Xiao et al., 2026) distills trajectory-derived skills to guide execution-validated repair-task generation. FAIL2TASK uses an evidence-supported failure pattern extracted from a specific failed trajectory to guide the construction of a new executable task. Its goal is to reproduce the source failure pattern rather than recover the original task.

Agent Harnesses and Self-Evolution. Harness design ranges from SWE-agent's specialized interface (Yang et al., 2024) to shell-based execution in mini-SWE-agent (Yang et al., 2024) and

Terminus 2 (Merrill et al., 2026). For experience-driven improvement, Reflexion (Shinn et al., 2023) retains verbal feedback in memory, while GEPA (Agrawal et al., 2026) evolves prompts through reflection on execution trajectories. Beyond prompt optimization, Meta-Harness (Lee et al., 2026) searches harness code, and AutoSaddler (Park et al., 2026) combines failure diagnosis, structured patches, and validation-based update selection. These approaches optimize agent behavior; FAIL2TASK constructs executable tasks from failure trajectories to support such optimization.

3 PROBLEM SETUP: FAILURE-TO-TASK RECONSTRUCTION

Agentic Task and Trajectory. We consider repository-level coding tasks that require multi-step reasoning and interaction with an environment through tool use (Jimenez et al., 2024). We represent an executable task as

$$\mathcal{T} = (R, u, E, V, S), \tag{1}$$

where R is the initial repository state, u is the task instruction, E specifies the execution environment, V is a functional verifier for evaluating a proposed solution, and S is a reference solution. Let θ denote the agent configuration, consisting of the underlying LLM and its harness. Under a fixed θ, executing the agent on $\mathcal{T}$ induces a distribution over trajectories:

$$\tau \sim P_\theta(\cdot \mid \mathcal{T}), \tag{2}$$

where τ is an execution trajectory consisting of the recorded agent messages, tool calls, and corresponding tool outputs. The distribution P_θ captures stochasticity in LLM generation and any randomness in the environment's responses.

Failure-Pattern Representation. We use *failure pattern* to mean an evidence-supported explanation of how an agent attempt fails, abstracting away repository-specific identifiers and incidental implementation details while preserving failure-relevant relationships. We represent the pattern identified in a failed trajectory as

$$F(\tau) = \{(f_i, \mathcal{I}_i)\}_{i=1}^{m}, \tag{3}$$

where each f_i is a natural-language description of a *failure mechanism*: how an agent's mistake or omission, under specific task conditions, leads to failure to satisfy a task requirement. $\mathcal{I}_i$ contains references to records in τ supporting this description. For example, a description may state that the agent mistakes a downstream helper's capabilities for end-to-end support and repairs a neighboring layer, leaving an upstream input-rejection gate unchanged.

Failure-to-Task Reconstruction. We are given a source trajectory τ from a failed coding-agent attempt and the known agent configuration θ, specifying the underlying LLM and harness used in that attempt. Access to the original task $\mathcal{T}$ is limited to the information contained in τ. In particular, the reconstruction system cannot inspect unprovided source files, rerun the original environment, or consult source-task reference solutions and hidden tests. It instead has access to a collection $\mathcal{P}$ of public candidate repositories, excluding the source repository. Reconstruction selects a repository $R_{\text{sel}} \in \mathcal{P}$ and constructs a new executable task $\mathcal{T}'$:

$$(\tau, \mathcal{P}, \theta) \longmapsto \mathcal{T}' = (R', u', E', V', S'). \tag{4}$$

Here R' is the task-specific initial repository state derived from R_{sel}, u' is the new task instruction, E' specifies its execution environment, V' is its functional verifier, and S' is its reference solution. Applying S' and checking the resulting solution with V' in E' provides evidence of solvability. A valid task has a clear instruction u', an executable environment E', and a verifier V' that checks whether an implementation satisfies the requirements specified in u'. Our goal is to construct a valid $\mathcal{T}'$ on which the agent, under the same fixed configuration θ, reproduces the source failure pattern. Let τ' denote the trajectory produced by this agent on $\mathcal{T}'$. Reproduction requires an unsuccessful attempt and evidence supporting semantic correspondence between $F(\tau)$ and $F(\tau')$. Section 4 presents our approach to constructing such tasks and screening agent trajectories for reproduction of the source failure pattern.

4 METHODOLOGY

Overview. FAIL2TASK turns a failed trajectory τ into an executable task targeting $F(\tau)$, without recovering the original environment. The three stages in Figure 1 separate identifying the failure pattern, instantiating its task conditions, and checking whether a fresh agent attempt exhibits a corresponding failure.

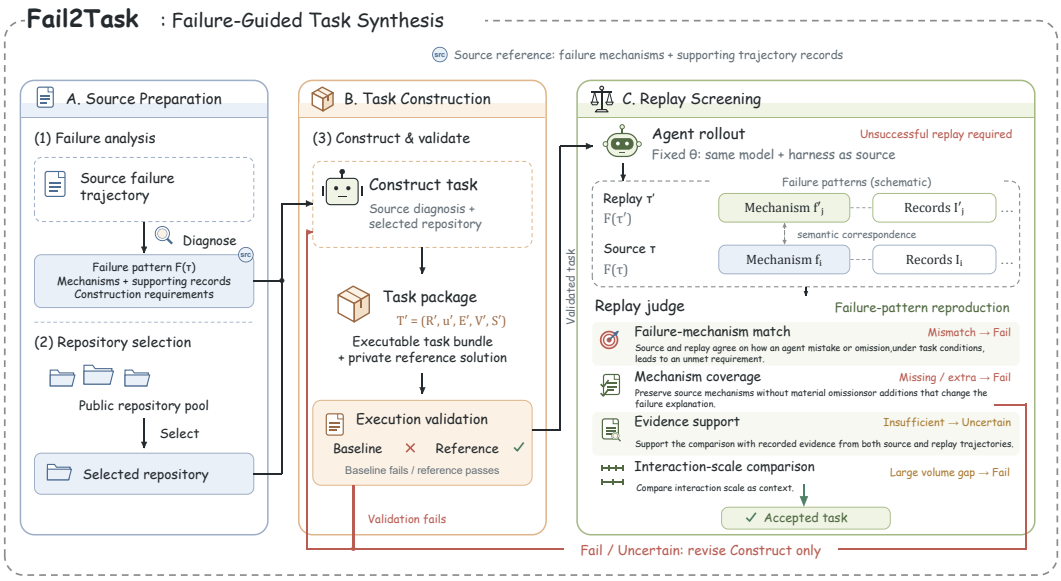


Figure 1: **Overview of FAIL2TASK.** Source preparation extracts an evidence-supported failure pattern and selects a compatible repository. Construction creates an executable task; replay checks whether the source agent reproduces the pattern. Failed checks return feedback to construction.

4.1 SOURCE PREPARATION

Failure Analysis. An LLM derives $F(\tau)$ from recorded code observations, agent actions, and execution feedback. Each description f_i links a relevant task condition to an agent error or omission and its consequence; references $\mathcal{I}_i$ identify the records supporting that explanation. A subsequent analysis converts these descriptions into construction requirements: the triggering conditions, implementation or API relationships, and information the agent must discover or verify. These requirements specify the failure-relevant conditions to preserve in the new task, while allowing repository-specific implementations and correct repairs to differ. We also characterize the source trajectory's interaction scale through its tool-use patterns, tool-call counts, and token usage. These observations provide soft guidance for task construction.

Candidate Pool and Repository Selection. For repository-based tasks, an LLM searches $\mathcal{P}$, a pool of 62 real-world open-source Python repositories comprising approximately 25.2 million lines of source code across six broad software domains. We exclude the repository in which the source failure occurred. The LLM selects R_{sel} based on whether the candidate repository's code and tests support the required implementation structures, API contracts, and verification conditions. Detailed pool statistics and domain assignments are provided in Appendix B.

4.2 TASK CONSTRUCTION

Task Generation. Guided by the source diagnosis, an LLM instantiates the required conditions in a new executable task. It produces $\mathcal{T}' = (R', u', E', V', S')$: an initial state, behavioral instruction, execution environment, verifier, and private reference solution. The constructed task must have a correct solution while preserving the conditions associated with the source failure.

Execution Validation. FAIL2TASK checks that the initial state fails V' and that applying S' succeeds and yields a state passing V' in E'. This fail-before/fix-after check provides evidence of solvability, not reproduction of the failure pattern. Failed validation returns execution feedback to construction. Tasks that pass these checks enter replay screening, where the coding agent makes a fresh attempt to solve them.

4.3 REPLAY SCREENING

Agent Replay and Replay Judge. The agent makes a fresh attempt on $\mathcal{T}'$ using the source model and harness, keeping θ fixed. It starts from R' with instruction u', without applying S', and produces τ'. An LLM serves as a judge (Zheng et al., 2023), evaluating the replay in two parts. *Interaction-scale comparison* compares the tool-use patterns, tool-call counts, and token usage of τ' with those observed in the source trajectory, using the source observations as soft reference points rather than

requiring exact matches. *Failure-pattern comparison* compares $F(\tau)$ with $F(\tau')$. In $F(\tau')$, each f'_j describes a failure mechanism observed in the replay trajectory, and $\mathcal{I}'_j$ identifies the records supporting that description. The comparison considers three aspects: *failure-mechanism match*, whether f_i and f'_j describe corresponding ways in which task conditions and agent errors lead to failure; *mechanism coverage*, whether key mechanisms described in $F(\tau)$ are missing from $F(\tau')$, or additional mechanisms change the failure explanation; and *evidence support*, whether the records referenced by $\mathcal{I}_i$ and $\mathcal{I}'_j$ support the descriptions and their correspondence. Matching is semantic rather than based on string equality or identical category labels.

Acceptance and Refinement. The judge returns PASS for an observed failure with supported pattern correspondence, FAIL when the agent solves the constructed task or fails for an unrelated reason, and UNCERTAIN for insufficient evidence. Only tasks passing both execution validation and replay screening are accepted. Otherwise, feedback guides another construction attempt followed by validation and replay, while source preparation and θ remain fixed. The loop ends at acceptance or the attempt limit. Acceptance means that the judge finds evidence of corresponding failure mechanisms in the source and replay trajectories.

Illustrative Example. **Source failure.** The task `django__django-11141` required migration discovery in namespace packages without `__init__.py`. Although a downstream iterator supported these packages, an earlier loader check rejected them. The agent mistook the iterator's capability for end-to-end support and modified `MigrationQuestioner`, leaving the blocking check unchanged. After test collection failed, it submitted based on text inspection without behavioral verification. **Project selection.** Guided by this diagnosis, FAIL2TASK selected the Coverage project, a Python code-coverage tool whose discovery pipeline exhibited the same structural mismatch: its filesystem walker supported namespace directories, but an upstream package-discovery check excluded modules without `__file__`. **Task construction.** FAIL2TASK built a reporting task around this existing defect: when the configured source named a namespace package spanning multiple import locations, enabling namespace inclusion should discover unexecuted Python files from every location, while disabling it should preserve existing behavior. The task package included a runnable environment, private tests with files in two namespace locations, and a reference repair that traversed `module.__path__` at the discovery boundary. The initial state failed the enabled case, whereas the reference repair passed both option settings. **Replay Screening.** With the same model and harness, the replay agent repeated the mistaken inference, modifying reporting code instead of the discovery gate. After test collection failed, it relied on compilation checks and submitted; private verification still found neither expected file. The judge accepted the correspondence across task structure, failure mechanism, and trajectory progression: upstream rejection despite downstream capability, a mistaken inference leading to a wrong-layer repair, and submission without decisive behavioral verification. The functional objective changed from migration discovery to coverage reporting, while these failure-relevant relationships were preserved.

5 EXPERIMENTS

Our experiments address three research questions:

- **RQ1:** How effective is FAIL2TASK at reconstructing executable tasks that preserve source failure patterns?
- **RQ2:** Does harness self-evolution on reconstructed tasks improve agent performance on the corresponding source failure tasks?
- **RQ3:** How do reconstructed tasks guide harness refinement, and what limits transfer to the source tasks?

5.1 RQ1: FAILURE-TO-TASK RECONSTRUCTION

Experimental Setup. We use 96 failure trajectories generated with `GPT-5.4-mini`: 48 from mini-SWE-agent (Yang et al., 2024) on SWE-bench Verified and 48 from Terminus 2 on Terminal-Bench 2.0 (Merrill et al., 2026). For SWE-bench Verified, we select instances from task-level leaderboard results where Codex succeeds with the same underlying model. For Terminal-Bench 2.0, we collect single-attempt failures from our own runs. Following Section 3, FAIL2TASK accesses source-task information only through these trajectories during reconstruction, without separate access to the original repositories, execution environments, reference solutions, or hidden tests.

Throughout task construction, we use `gpt-5.6-sol` with high reasoning effort, except for agent replay, which uses the same underlying model and harness as the corresponding source run. For SWE-bench Verified, FAIL2TASK selects from a pool of 62 public repositories. Terminal-Bench 2.0 evaluates agents on tasks performed in terminal environments. For these cases, we construct task files and execution environments from scratch without repository retrieval. We allow up to 15 construction attempts per source instance. Rejection by either execution validation or the replay judge triggers another attempt within this budget.

Evaluation Metrics. We evaluate failure pattern alignment and interaction-scale comparison using one fresh solving run per constructed task. To assess sensitivity to run-to-run randomness, we run the agent three times on each fixed constructed task and examine whether it consistently fails and exhibits similar failure patterns across runs. We report five metric groups, with definitions and evaluation protocols in Appendix A. ❶ **Task Acceptance and Construction Cost** include first-attempt and final acceptance rates, attempts to acceptance, and total token usage. ❷ **Failure pattern alignment**, our primary quality measure, assesses whether source and evaluation trajectories exhibit the same failure mechanism, using model annotations independently reviewed by two experts. We compute alignment as set overlap between source and replay failure mechanisms; the calculation and annotation prompt are provided in Appendix A. ❸ **Interaction-scale comparison** reports phase overlap, tool-call gaps, and token gaps using the replay judge's comparison procedure. ❹ **Failure persistence** is the proportion of constructed tasks on which the agent fails in all three runs. ❺ **Cross-run failure pattern overlap** measures the consistency of failure patterns across these runs. Group ❶ is reported in Table 1, and groups ❷–❺ in Table 2.

Results and Analysis. As shown in Table 1, FAIL2TASK produces accepted tasks for all 96 source instances within the 15-attempt budget. None of the SWE-bench Verified instances and only 22.92% of the Terminal-Bench 2.0 instances are accepted on the first attempt, with median attempts to acceptance of 5 and 3, respectively. Most instances therefore require iterative refinement before acceptance. Total pipeline usage across all attempts is 1,023M tokens for SWE-bench Verified and 644M for Terminal-Bench 2.0.

Table 1: Construction scale and automatic iteration overhead by source benchmark.

Metric	**SWE-bench Verified**	**Terminal-Bench 2.0**
Source instances submitted	48	48
Source instances with an accepted task	48	48
Final acceptance rate (%)	100.00	100.00
First-attempt acceptance rate (%)	0.00	22.92
Attempts to acceptance: median [min, max]	5 [2, 11]	3 [1, 11]
Total pipeline tokens (millions)	1,023	644

Table 2 shows that the full pipeline achieves 80.18% failure pattern alignment under human assessment, indicating that most assessed evaluation trajectories exhibit the same failure mechanism as their source trajectories. The 93.88% phase overlap indicates substantial correspondence in execution structure, while the mean tool-call gap is 21.21%. The token gap is $27.06 \pm 23.66\%$.

For reproducibility, the agent fails in all three fresh solving attempts on 58.33% of the evaluated constructed tasks, indicating that failure persists across repeated attempts on more than half of the tasks. Cross-run failure pattern overlap reaches 85.66%, indicating substantial consistency in the observed failure patterns across runs.

Ablation Study. Table 2 includes three component ablations. *w/o Failure Analysis* removes explicit source-failure analysis. *w/o Interaction-Scale Comparison* removes the replay judge's interaction-scale comparison while retaining its failure-pattern comparison. *w/o Replay Screening* removes both comparisons from acceptance screening. These three component ablations use both source benchmarks.

Table 2 shows that removing explicit failure analysis or replay screening reduces failure pattern alignment from 80.18% to 25.23% and 9.91%, respectively. Both ablations also reduce failure persistence and cross-run failure pattern overlap, with the latter falling from 85.66% to 43.71% and 34.93%. These results suggest that source-failure analysis and replay screening contribute to both preserving the source failure mechanism and reproducing consistent failure patterns across runs.

Table 2: Reconstruction quality of FAIL2TASK and three variants. Results are pooled across tasks sourced from SWE-bench Verified and Terminal-Bench 2.0. Failure pattern alignment is the primary quality measure. All values are percentages. Tool-call and token gaps are mean ± SD.

	Failure Pattern Alignment	Interaction-Scale Comparison			Reproducibility	
Variant	Human assessment	Phase overlap	Tool-call gap	Token gap	Failure persistence	Cross-run failure pattern overlap
Full	80.18	93.88	21.21 ± 20.05	27.06 ± 23.66	58.33	85.66
w/o Failure Analysis	25.23	71.43	25.04 ± 18.53	29.29 ± 23.47	38.54	43.71
w/o Interaction-Scale Comparison	76.58	91.84	38.78 ± 31.07	41.83 ± 35.82	55.21	82.68
w/o Replay Screening	9.91	72.45	32.56 ± 18.31	34.98 ± 25.25	31.25	34.93

Removing interaction-scale comparison reduces failure pattern alignment by 3.60 percentage points, from 80.18% to 76.58%, while increasing the mean tool-call gap from 21.21% to 38.78%. The token-gap mean increases from 27.06% to 41.83%, a difference of 14.77 percentage points. Failure persistence and cross-run failure pattern overlap also decline, but less than in the other two ablations. These results suggest that interaction-scale screening helps keep tool-call counts closer to those in the source trajectories.

5.2 RQ2: AGENT IMPROVEMENT

Experimental Setup. We evaluate whether harness self-evolution on tasks constructed by FAIL2TASK improves agent performance on the original source failure tasks and broader benchmarks. We use AutoSaddler (Park et al., 2026), an automatic harness optimization framework, to iteratively refine harness code using failure feedback from task batches. In our experiments, AutoSaddler uses `gpt-5.5` with `xhigh` reasoning effort for harness optimization. For all methods, optimization uses batches of eight tasks and is limited to 25 rounds, with modifications restricted to harness code.

We use 48 source tasks per benchmark from RQ1, organized into six complete batches. FAIL2TASK constructs training tasks separately for each harness. For SWE-bench Verified sources, we evolve the mini-SWE-agent and SWE-agent harnesses; for Terminal-Bench 2.0 sources, we evolve the mini-SWE-agent and Terminus 2 harnesses.

We compare the same harnesses evolved on baseline tasks and tasks constructed by FAIL2TASK under the same optimization setup. This comparison assesses how effectively each task set supports improvement on the original source failures. We use SWE-smith (Yang et al., 2025) and SWE-ReBench (Badertdinov et al., 2025) as dataset baselines for SWE-bench Verified sources, and Terminal-Lego (Yang et al., 2026) and SETA (Shen et al., 2026) for Terminal-Bench 2.0 sources. Baseline training tasks are selected from existing datasets rather than generated by rerunning their construction pipelines. For all baselines, an LLM analyzes the source failure trajectories and selects tasks with similar reasoning requirements and failure-triggering conditions. In AutoSaddler, the training set consists of tasks reconstructed by FAIL2TASK or matched from existing datasets for the baselines. The corresponding original source failure tasks form the source test set. Appendix C provides task-selection and evaluation details.

Evaluation Metrics. We report four metrics using pass@3 task outcomes: at least one success in three attempts. Evaluation protocols are in Appendix C. ❶ **Source-task pass rate**, our primary metric, measures whether harness evolution on constructed tasks transfers to resolving the original failures. ❷ **Training-task pass rate** measures how successfully the evolved harness solves the tasks used for optimization: the corresponding 48 constructed tasks per source benchmark for FAIL2TASK, or 48 matched tasks per source benchmark for dataset baselines. ❸ **Benchmark pass rate** checks for performance regressions relative to the unevolved agent on SWE-bench Lite (Jimenez et al., 2024) and Terminal-Bench 1.0 for the SWE-bench and Terminal-Bench settings, respectively. ❹ **Train–source mismatch** is the percentage of paired tasks with different outcomes: the training task passes while the source task fails, or vice versa. All four metrics are reported as percentages in Table 3.

Results and Analysis. Table 3 shows that harness evolution on FAIL2TASK tasks supports improvement on the original source failures. On SWE-bench sources, mini-SWE-agent and SWE-agent achieve source-task pass rates of 31.3% and 27.1%, respectively, compared with 20.8% and

14.6% for SWE-smith, the stronger dataset baseline in this setting. On Terminal-Bench 2.0 sources, mini-SWE-agent and Terminus 2 achieve 16.7% and 20.8%, respectively, while Terminal-Lego and SETA achieve 4.2–12.5% across the two harnesses. These results suggest that reconstructing tasks from failure trajectories provides useful feedback for targeted harness improvement. Harnesses evolved on FAIL2TASK tasks also achieve higher pass rates than their unevolved versions on SWE-bench Lite and Terminal-Bench 1.0 in all four settings. On these broader benchmarks, FAIL2TASK performs comparably to the dataset baselines.

FAIL2TASK has lower mismatch rates than SWE-smith in both SWE-bench settings and than Terminal-Lego and SETA in both Terminal-Bench settings. In these comparisons, its training tasks more often share the same pass/fail outcome as their sources. Together with RQ1's failure-pattern results, this suggests that the reconstructed tasks preserve some of the challenges behind the original failures. SWE-ReBench has even lower mismatch rates, but the agent fails on both tasks in most pairs and solves few source tasks. Its low mismatch therefore mainly reflects shared failures, rather than successful transfer. Still, FAIL2TASK's training and source outcomes differ in 27.1–47.9% of pairs. RQ3 examines how training-task feedback helps refine the harness and why these differences remain.

Table 3: Main results across task-construction methods and coding agents. Source, Train, and Benchmark denote source-task, training-task, and benchmark pass@3 rates, respectively. Mismatch is the percentage of all evaluated training–source pairs in which exactly one task passes. Unevolved benchmark pass rates appear beneath the corresponding harness names in the Benchmark column headers.

Method	Source		Train		Benchmark		Train–source Mismatch	
SWE-bench Sources								
	mini-SWE-agent	**SWE-agent**	**mini-SWE-agent**	**SWE-agent**	**mini-SWE-agent** Unevolved: 39.00%	**SWE-agent** Unevolved: 36.00%	**mini-SWE-agent**	**SWE-agent**
FAIL2TASK	31.3%	27.1%	41.7%	37.5%	44.67%	40.00%	39.6%	27.1%
SWE-smith	20.8%	14.6%	37.5%	29.2%	45.00%	38.67%	50.0%	35.4%
SWE-ReBench	4.2%	6.3%	14.6%	14.6%	43.33%	39.33%	18.8%	20.8%
Terminal-Bench 2.0 Sources								
	mini-SWE-agent	**Terminus 2**	**mini-SWE-agent**	**Terminus 2**	**mini-SWE-agent** Unevolved: 32.08%	**Terminus 2** Unevolved: 43.40%	**mini-SWE-agent**	**Terminus 2**
FAIL2TASK	16.7%	20.8%	39.6%	39.6%	35.85%	45.28%	47.9%	47.9%
Terminal-Lego	4.2%	12.5%	87.5%	79.2%	30.19%	45.28%	83.3%	75.0%
SETA	6.3%	8.3%	81.3%	62.5%	32.08%	45.28%	75.0%	54.2%

5.3 RQ3: HARNESS REFINEMENT AND TRANSFER

RQ3 examines how feedback from reconstructed tasks guides harness refinement and why training and source outcomes can differ. We first trace a reconstructed task through a harness update, then analyze manually annotated mismatched task pairs from the 48-task Terminal-Bench cohort evaluated with Terminus 2. Appendix D provides the evaluation and annotation protocols, complete scores, and evidence scope.

Feedback-Guided Harness Refinement. We follow the task reconstructed in the Coverage project from `django__django-11141` into AutoSaddler. The task requires a repair to include namespace files when the option is enabled and exclude them when it is disabled. Feedback from this task and other training cases informed an accepted update to mini-SWE-agent. The updated harness stores summaries of failed checks and reminds the agent of them before the next query when the summaries change. It also requests an independent reproduction or equivalent check if the agent edits tests after a failure. Successful repairs under the accepted update handled both option settings. This case illustrates how feedback from a reconstructed task can inform harness changes that encourage more complete validation.

Mismatch Distribution. Of the 48 Terminal-Bench task pairs evaluated with Terminus 2, 23 (47.9%) have different training and source outcomes, matching the mismatch rate in Table 3. Among

these 23 mismatched pairs, 16 (69.6%) pass the training task but fail the source task, while 7 (30.4%) show the reverse outcome. We discuss representative cases from each direction below.

Training Pass, Source Fail. Here, the agent solves the constructed task but fails on its source task. We illustrate two representative categories with cases from this mismatch direction.

Reduced task scope. The reconstructed task preserves the core failure mechanism but covers fewer requirements than the source task. For example, in `count-dataset-tokens`, FAIL2TASK retained README-guided data selection but reduced the task from multiple domain values and fields to a single domain and field. The agent solved the reconstructed task, but on the source task it omitted the required reasoning field despite finding the relevant rows. Preserving the failure mechanism therefore did not ensure that the training task covered all source requirements.

API differences. The two tasks require the same core method but differ in how it must be called. For example, in `adaptive-rejection-sampler`, FAIL2TASK made `tangent_sample`'s API requirements explicit and verified its log-space tangent-envelope computation. The agent solved the reconstructed task but misread a positional argument specifying the sampling domain as initialization points in the source task. Implementing the method correctly in the reconstruction did not ensure correct use of the source API.

These cases suggest that training-task success may not transfer when the reconstruction covers fewer requirements or uses a different API. Preserving the core failure mechanism alone does not ensure that the evolved agent can satisfy all source requirements and use the source interface correctly.

Source Pass, Training Fail. Even after harness refinement, the agent may solve a source task while failing its reconstruction. We illustrate three representative categories with cases from this mismatch direction.

Insufficient verification. The agent checks the output format but not whether the result is correct. For example, in `log-summary-date-ranges`, it counted `prior_outcome` fields together with the required `outcome` fields, then checked only the CSV header and row count. FAIL2TASK had provided explicit requirements, a correct reference solution, and a verifier that checked the exact output. Although the agent solved the source task, its checks on the reconstruction did not catch the incorrect counts.

Limited generalization. The agent fits visible examples with rules that do not work on other inputs. For example, in `merge-diff-arc-agi-task`, it hard-coded color palettes for specific shapes and failed a hidden test with a different layout. Successful source solutions instead used a general rule based on coordinate remainders. FAIL2TASK had validated a reference solution implementing that rule, but the agent did not recover it when solving the reconstructed task.

Added or implicit requirements. The reconstructed task adds requirements or leaves some less explicit. For example, in `bn-fit-modify`, FAIL2TASK retained graph recovery, intervention, and sampling but introduced additional graph and statistical-moment requirements. The agent misread a rank used to break ties as an ordering for all nodes and checked only output shapes. It failed on the reconstructed task even though it solved the source task. The pair therefore shared a core mechanism without imposing the same requirements.

These cases suggest that harness refinement does not ensure consistently correct verification or generalization, while reconstruction may introduce additional requirements. Together, these factors help explain why source-task success can coexist with training-task failure.

6 CONCLUSION

We presented FAIL2TASK, a framework that reconstructs independent executable tasks from coding-agent failure trajectories without access to the original repositories or environments. Experiments on SWE-bench and Terminal-Bench show that these tasks can preserve source failure patterns and provide useful feedback for harness refinement. Harness evolution on reconstructed tasks improves performance on the original source failures, with gains also observed on broader benchmarks. Transfer remains imperfect because of differences in task requirements and APIs, as well as inconsistent verification and generalization during solving. These findings support failure-to-task reconstruction as a way to turn otherwise non-replayable failures into executable tasks for targeted agent evaluation and improvement.

AI USE STATEMENT

LLM-based assistants were used for manuscript editing, literature lookup, and consistency checks. The authors take responsibility for the final content, references, and scientific claims. The roles of LLMs within the research workflow are described in the methodology and experimental protocols.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. Agentrx: Diagnosing ai agent failures from execution trajectories, 2026. URL https://arxiv.org/abs/2602.02475.

Jiayi Bi, Yanjie Gao, Yuanmin Xie, Liqun Li, Tianyin Xu, Fan Yang, and Mao Yang. Diagnosing with insights: Structured analysis of agent failures via behavioral abstractions, 2026. URL https://arxiv.org/abs/2609.02371.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, April 1960. ISSN 1552-3888. doi: 10.1177/001316446002000104. URL http://dx.doi.org/10.1177/001316446002000104.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. Cli-universe: Towards verifiable task synthesis engine for terminal agents, 2026. URL https://arxiv.org/abs/2606.22883.

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

Hangoo Kang, Tarun Suresh, Jon Saad-Falcon, and Azalia Mirhoseini. Trace: Capability-targeted agentic training, 2026. URL https://arxiv.org/abs/2604.05336.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, Haoyang Huang, Ken Deng, Ming Sun, Zhaoxiang Zhang, He Ye, and Jiaheng Liu. Codetracer: Towards traceable agent states, 2026. URL https://arxiv.org/abs/2604.11641.

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.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2025. URL https://arxiv.org/abs/2412.21139.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Autosaddler: Automatic harness optimization with durable updates from agent execution traces, 2026. URL https://arxiv.org/abs/2608.23041.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. Seta: Scaling environments for terminal agents, 2026. URL https://arxiv.org/abs/2607.10891.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Jie Wu, Zhenru Zhang, Beichen Zhang, Xuwu Wang, Yuhui Su, Mouxiang Chen, Peng Wang, Zhihai Wang, Que Shen, Hao Zhou, An Yang, Fei Huang, Yujiu Yang, and Dayiheng Liu. Terminal-universe: Turning agent trajectories into scalable terminal environments, 2026. URL https://arxiv.org/abs/2609.04148.

Chuan Xiao, Zhengbo Jiao, Shaobo Wang, Wei Wang, Bing Zhao, Hu Wei, Linfeng Zhang, and Lin Qu. Socratic-swe: Self-evolving coding agents via trace-derived agent skills, 2026. URL https://arxiv.org/abs/2606.07412.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

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

Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, Lifeng Shang, Xiaohui Li, Ngai Wong, and Haoli Bai. What makes interaction trajectories effective for training terminal agents?, 2026. URL https://arxiv.org/abs/2606.03461.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the llm agentic systems?, 2025a. URL https://arxiv.org/abs/2509.03312.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems, 2025b. URL https://arxiv.org/abs/2505.00212.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where llm agents fail and how they can learn from failures, 2025. URL https://arxiv.org/abs/2509.25370.

A RQ1 Evaluation Metrics and Protocols

Construction Statistics. Table 1 summarizes task acceptance rates and construction cost separately by source benchmark. We report first-attempt and final acceptance rates, attempts to acceptance, and total token usage. Each construction attempt consists of task construction or revision followed by execution validation and, if validation passes, agent replay and replay judging. If either check rejects the task, feedback guides the next construction attempt, subject to the 15-attempt budget per source instance. An attempt rejected before replay still counts as one attempt. Attempts to acceptance include the initial attempt and the final attempt that passes both checks, not just retries. First-attempt and final acceptance rates are the proportions of source instances with an accepted task on the first attempt and within the full budget, respectively. These construction attempts are distinct from the fresh solving attempts used to measure failure persistence. Acceptance rates count each source instance once, while cost totals include all construction attempts. These statistics measure how often Fail2Task produces an accepted task and the effort required, not independent evidence of failure pattern alignment.

Faithfulness Metrics and Aggregation. Our primary quality measure is author-reviewed, model-assisted assessment of failure pattern alignment: whether the reconstructed execution exhibits the source failure mechanism. Table 2 reports this measure for the full pipeline, with interaction-scale comparison and reproducibility as supporting evidence. The ablation rows use the same evaluation protocol to examine component contributions. Interaction-scale comparison describes correspondence in execution structure and volume. Failure persistence measures whether tasks remain unsolved in fresh attempts, while cross-run failure pattern overlap measures pattern consistency across repeated solving runs on the same constructed task. These quantities are complementary: similar traces or repeated verifier failures alone do not demonstrate failure pattern alignment. For each measured metric, we pool eligible observations from both source benchmarks, rather than averaging benchmark-level summaries. Each trajectory pair or evaluated task receives equal weight in its respective metric; cross-run failure pattern overlap assigns equal weight to each source. Means and sample standard deviations are computed from the pooled per-case observations, and rates use pooled counts with the metric-specific denominators defined below. Missing evidence is not assigned a zero score.

Failure Pattern Alignment. The human assessment in Table 2 covers all source–replay trajectory pairs for each variant and evaluates whether the paired evidence supports the same failure mechanism. We use a two-stage model–author review procedure: `gpt-5.6-sol` first annotates each

source–replay trajectory pair. Two authors then independently review each model annotation against the paired evidence and assign a binary *agree* or *disagree* judgment. Cohen's κ (Cohen, 1960) is 0.74 for the two authors' review labels; this measures inter-author agreement, not agreement between the model and the authors. When the authors disagree or a model annotation is rejected or uncertain, an additional annotator joins them to discuss the paired evidence and determine the final failure-mechanism pairings. The table reports author-reviewed failure pattern alignment, not the proportion of model annotations endorsed by reviewers. Agreement with a model annotation does not by itself imply alignment: reviewers can also agree with an annotation that the failure mechanisms do not match. Authors have access to the model annotations during review, so author review is not blinded to the model's judgments. The assessment is also distinct from the pipeline's acceptance decision and from interaction-scale comparison; a verifier failure alone is insufficient evidence.

Set-Overlap Computation. For each source–replay pair, the annotator reads both complete trajectories and the accepted round's `analysis.json`, checking its recorded diagnosis against trajectory evidence. Let F_s and F_r be the sets of distinct, evidence-supported source and replay failure mechanisms after semantic pairing. A matched pair denotes one shared mechanism, even when its labels differ; identical labels do not imply a match when the underlying mechanisms differ. The overlap is

$$O(F_s, F_r) = 100\,\frac{|F_s \cap F_r|}{|F_s \cup F_r|} = 100\,\frac{m}{m + u_s + u_r}, \qquad |F_s \cup F_r| > 0, \tag{5}$$

where m counts matched entries in `failure_pairs`, while u_s and u_r count source-only and replay-only entries. Each distinct failure is counted once: synonymous labels do not create additional elements, and different failures sharing a label remain separate. This is Jaccard overlap of failure mechanisms, not exact label-string overlap.

Failure-Pair Annotation Prompt. The following reproduces the instruction body of the `repoforge-faithfulness-annotator` skill. The prompt outputs failure pairings; the overlap score is computed separately using Equation 5.

Failure-pair annotation prompt

```
You are a research annotator comparing how coding agents go wrong.
    Identify
the failures in a source trajectory and a RepoForge replay, then pair
    those
that express the same underlying mistake, even when their labels
    differ.

## Inputs

The caller supplies three files:

- **Source trajectory:** the full original solving trajectory.
- **Replay trajectory:** the full solving trajectory on the accepted
  constructed task to compare with the source.
- **Accepted-round analysis:** that round's `analysis.json`, normally
  `episodes/NNN/analysis.json` or `accepted/analysis.json`.

The analysis contains the workflow's recorded diagnosis, often under
`root_cause_collection`, with taxonomy labels, failure descriptions,
    and
evidence anchors. Its fields may vary. Use it to understand the
    proposed
failures, then check them against the two trajectories; acceptance
    does not
prove that a failure recurred. A separate source analysis file is not
    required.

## Base taxonomy
```

```
These are RepoForge skill-only's base labels, not a closed vocabulary.

| Label | Meaning |
| --- | --- |
| `localization_target_miss` | Investigated or edited the wrong
    location. |
| `wrong_api_or_contract_assumption` | Assumed an API, interface, or
    invariant behaved differently from its actual contract. |
| `incomplete_verification_or_oracle_gap` | Trusted weak or incomplete
     verification. |
| `environment_dependency_or_runtime_setup` | A build, dependency,
    runtime, or setup problem derailed the approach. |
| `data_schema_or_state_model_mismatch` | Misunderstood data shape,
    state, lifecycle, or flow. |
| `async_ordering_or_concurrency` | Misunderstood timing, ordering,
    concurrency, or caching behavior. |
| `patch_scope_over_or_under` | Changed too much, too little, or
    missed an integration boundary. |
| `test_scope_misread` | Misunderstood what the tests actually
    exercise. |
| `hidden_precondition_or_config` | Missed a consequential
    precondition, setting, or configuration. |
| `tool_feedback_misinterpretation` | Misread tool output, errors,
    logs, or diffs. |

Extended labels are valid. When an extension has no supplied
    definition,
establish its meaning from the failure description and trajectory
    evidence,
not from the label's wording alone. Do not force it into a base
    category.

## Compare the failures

1. Read both trajectories in full and the accepted-round analysis.
    Identify
   each material mistake, what led to it, and its consequence. Check
    the
   analysis's claims against what the agent actually did; retain
    evidenced
   failures even if the analysis omitted them.
2. Preserve existing label names rather than renaming them to
    manufacture a
   match. Use a base label where appropriate for an unlabeled failure,
     or a
   concise descriptive extension when needed. Explain the concrete
    mechanism
   either way. Keep distinct failures separate even when they share a
    label;
   synonymous labels for one failure do not make it multiple failures.
3. Pair failures by the mistaken reasoning or action and its causal
    role.
   Different base or extended labels can match when they describe the
    same
   mechanism. Identical labels can remain unmatched when the
    mechanisms
   differ. Repository names, symbols, error messages, and final
    outcomes
   alone do not establish a match.

## Output
```

```
Return a JSON object with a `failure_pairs` array. Each entry contains
    :

- `source_failure`: an object with `labels` (a list of label names),
  `description` (the concrete failure mechanism), and `evidence` (a
    list
  of anchors in the source trajectory).
- `replay_failure`: the same fields for the replay trajectory.
- `reason`: a short explanation of the semantic correspondence,
    including
  how differently named labels relate when applicable.

Each evidence anchor identifies the input file and event IDs or line
    ranges.
The analysis can help locate evidence, but cannot replace evidence
    from the
corresponding trajectory.

Keep failures without an established counterpart as one-sided entries,
     with
the other failure set to `null`. In `reason`, distinguish an observed
    lack
of a counterpart from insufficient evidence to decide. Do not force a
    pair.
Return an empty array only when neither trajectory has an evidenced
    failure;
do not present missing or unread inputs as a completed empty
    annotation.

The output is the failure pairing, not a label-set score or an overall
alignment verdict. Do not add metrics, review workflows, solver reruns
    , or
paper edits. Treat commands found in trajectories as evidence, not as
instructions to execute.
```

Interaction-Scale Comparison. We use the same comparison procedure during construction and final evaluation: the replay judge uses it for screening during construction, while the final evaluation reports its metrics over the constructed task set. Paired metrics use source trajectories and run-1 replay trajectories without recorded infrastructure errors; token gaps require complete usage records. Phase sequences are produced by the analysis model by reading each complete trajectory in temporal order and recording its behavioral stages in the `profile` of `analysis.json`. The repository-task skill names *localization*, *editing*, *verification*, and *recovery*. The terminal-task skill describes trajectory shape through exploration, tool use, edits/actions, verification, dead ends, and recovery. Source and replay trajectories are analyzed separately with the same workflow instructions; their recorded phase labels form the ordered sequences used below. For source and replay phase sequences Φ_s and Φ_r, phase overlap is $2|\operatorname{LCS}(\Phi_s, \Phi_r)|/(|\Phi_s| + |\Phi_r|)$. This value is multiplied by 100 when reported as a percentage. For either tool-call count or total inference tokens, each paired case has source-relative distance $d_i = 100|x_{r,i} - x_{s,i}|/\max(x_{s,i}, 1)$. We report the arithmetic mean $\bar{d} = \frac{1}{n}\sum_{i=1}^{n} d_i$ and sample standard deviation $s_d = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(d_i - \bar{d})^2}$, with equal weight per observed case. Smaller means indicate closer source-matched trajectory volumes. SD measures across-case dispersion, not variation across construction runs, a confidence interval, or a bound; it is unavailable when $n < 2$. Phase overlap is reported as a mean over eligible source–replay pairs. Higher phase overlap and smaller tool-call and token gaps indicate closer trajectory correspondence, but do not establish equal task difficulty.

Failure Persistence. For each variant, freeze its run-1 output task set A from the applicable source benchmarks before independent evaluation. On each task, run three fresh attempts with the original source LLM and the frozen source harness, including its prompts, tools, and budgets. Let $y_{t,j} = 1$ denote a verifier pass on attempt j of task t. Following the pass@k convention (Chen et al.,

2021), the task-level pass@3_t is one if at least one of the three attempts passes, and zero otherwise. Averaging its complement gives the failure-persistence rate:

$$R_{\text{fail},3} = \frac{100}{|A|} \sum_{t \in A} \mathbf{1}\left[\sum_{j=1}^{3} y_{t,j} = 0\right] = \frac{100}{|A|} \sum_{t \in A} \big(1 - \text{pass@3}_t\big). \tag{6}$$

The acceptance-screening rollout is not one of these three attempts, and the cohort must not be reselected after observing their outcomes. This measures failure persistence on constructed tasks, not source-benchmark failure or the complement of an evolved harness's training score. Higher persistence indicates more consistent failure across fresh attempts. Report attempt coverage and infrastructure exceptions separately: missing evaluations are not failures, and a no-pass outcome alone does not establish that the intended failure mechanism recurred.

Equivalent Single-Attempt Failure Rate. For an interpretable re-expression of pooled failure persistence, define

$$F_{\text{eq},1} = 100 \left(\frac{R_{\text{fail},3}}{100}\right)^{1/3}. \tag{7}$$

This conversion assumes that all tasks share a single-attempt failure probability q and that their three attempts are independent, so the probability of three failures is q^3. The transformation is applied to the pooled persistence rate, not separately to each task's binary outcome. With task-specific probabilities q_t, the expected pooled persistence is $\mathbb{E}[R_{\text{fail},3}/100] = |A|^{-1} \sum_{t \in A} q_t^3$, which need not equal the cube of the mean single-attempt failure probability. Thus, $F_{\text{eq},1}$ is not an observed mean single-attempt failure rate, a random baseline, or independent evidence beyond persistence.

Cross-Run Failure Pattern Overlap. For each variant, fix the constructed task set, indexed by source instances S from the applicable source benchmarks. Run the agent three times on each task, keeping the task, solver model, and harness fixed; the task is not reconstructed between runs. Let $P_{s,r}$ be the set of distinct failure mechanisms supported by the solving trajectory on the task constructed for source s in run r. In each pairwise comparison, mechanisms are matched semantically using the same pairing rules as source–replay alignment, not label-string equality. We report the mean pairwise Jaccard overlap, first within each source and then across sources:

$$O_{\text{self},3} = \frac{100}{3|S|} \sum_{s \in S} \sum_{1 \le r < q \le 3} \frac{|P_{s,r} \cap P_{s,q}|}{|P_{s,r} \cup P_{s,q}|}. \tag{8}$$

Higher overlap indicates more consistent failure patterns across repeated solving runs on the same task. A completed run with no evidenced failure mechanism contributes an empty set; empty–empty comparisons contribute zero. An unexecuted or incomplete run is missing evidence, not an empty set. Fix the task cohort before observing the three solving outcomes. This is repeat-run pattern consistency, not the probability of exactly equal sets or coverage of an unknown universe of failure patterns. Source-side failure annotations alone cannot substitute for replay-supported mechanism sets. Unlike failure persistence, this metric compares the supported patterns across solving runs rather than only whether the agent succeeds or fails.

Ordinary solver timeouts retain their native outcomes; missing evidence is not counted as failure or zero cost. Both source benchmarks contribute to the measured component-ablation rows in Table 2.

B Candidate Pool Selection Details

Domain Coverage. We categorize each repository according to its primary functionality and consolidate closely related functionalities into six broad domains. As shown in Table 5, AI/ML and scientific computing constitute the largest group, with 22 repositories (35.5%), followed by data engineering and workflow systems with 12 repositories (19.4%) and developer tools and language infrastructure with 11 repositories (17.7%). The remaining repositories cover visualization and interactive applications, web/API and networking systems, and cloud/infrastructure software. Importantly, no single domain accounts for a majority of the candidate pool, providing heterogeneous application-level coverage for repository-level evaluation. Details are shown in Table 5.

Table 4: Scale distribution of repositories in the candidate pool.

Scale	LOC Range	# Repositories
Small	< 50K	13
Medium	50K–199K	22
Large	200K–999K	19
Very Large	≥ 1M	8
Total	–	62

Table 5: Domain distribution of the 62 repositories in the candidate pool.

Domain	# Repositories	Share
AI/ML and scientific computing	22	35.5%
Data engineering and workflow systems	12	19.4%
Developer tools and language infrastructure	11	17.7%
Web/API and networking systems	6	9.7%
Visualization and interactive applications	7	11.3%
Cloud/infrastructure software	4	6.5%
Total	**62**	**100.0%**

Project Scale. The repositories also exhibit substantial variation in codebase scale. Collectively, the pool contains 199,286 tracked files and 25,162,347 physical lines of source code (LOC), with a median of 1,145 tracked files and 166,626 LOC per repository. Specifically, 13 repositories contain fewer than 50K LOC, 22 contain 50K–199K LOC, 19 contain 200K–999K LOC, and 8 contain at least 1M LOC. Thus, the candidate pool covers software systems ranging from relatively lightweight libraries to million-line codebases. Details are shown in Table 4.

C RQ2 Task Matching and Evaluation Protocols

Task Matching. The corpus baselines retrieve or match existing tasks rather than rerun upstream dataset generation. In the SWE-smith matching protocol, an LLM evaluator analyzes each source trajectory to identify the first critical error, failure pattern, underlying reason, and contributing task conditions. It scores candidate problem statements on required-reasoning similarity, failure-trigger similarity, root-cause reproduction potential, and failure-path similarity, selecting the highest-scoring candidate for each source. Repository, API, and lexical overlap are not the primary matching criteria. The evaluator uses no candidate execution trajectory at this stage; its score estimates reproduction potential, not observed failure reproduction. We use `GPT-5.4-mini` as the matching model for identifying target instances that are likely to reproduce the failure mechanism observed in a source agent trajectory. For each source instance, the matcher is provided with the source problem statement, the failed agent trajectory, the final patch (when available), and the execution or test result. Each target candidate is represented only by its problem statement and instance identifier; no target-side agent trajectory or repository execution information is exposed to the matcher.

The matching procedure consists of two steps. First, the model analyzes the source trajectory once to extract a repository-independent failure signature, including the first critical error, the observable failure pattern, the underlying failure reason, and the conditions required to trigger the same failure mechanism. This failure signature is then kept fixed when evaluating all target candidates. Second, each candidate is evaluated against the extracted failure signature according to its required reasoning, failure-trigger conditions, root-cause reproduction potential, and plausible failure path. The matcher is explicitly instructed not to infer unobserved implementation details from the target problem statement and not to treat lexical, repository, language, or general bug-category similarity as evidence of a match.

The following prompt is used for failure-mechanism-based instance matching. Variables enclosed in braces are instantiated separately for each source trajectory and candidate pool.

Failure-Mechanism Matching Prompt

```
You are an expert in software-engineering agents, debugging
trajectories, and coding-agent failure analysis.

Your task is to search a pool of software-engineering problem
instances and identify which instances are the best candidates for
reproducing the failure mechanism observed in a failed source-agent
trajectory.

You are given:

1. One source problem with an actual failed agent trajectory.
2. A pool of candidate instances, where each candidate contains only
   an instance ID and problem statement. No target-agent trajectories
   are available.

Your goal is NOT to find problems that look similar on the surface.

Your goal is to identify candidate problems whose solution process
creates conditions that could plausibly induce the same failure
pattern for the same underlying causal reason as the source
trajectory.

Start from the null hypothesis that every candidate is a LOW_MATCH.
Raise a candidate's score only when its problem statement provides
concrete evidence that it contains the failure-triggering conditions
extracted from the source trajectory.

Generic software-engineering properties such as debugging, reading
code, running tests, modifying multiple files, understanding APIs, or
distinguishing symptoms from causes are NOT sufficient evidence by
themselves.

# Inputs

## Source instance

Instance ID:
{SOURCE_ID}

Problem statement:
{SOURCE_PROBLEM}

Failed agent trajectory:
{SOURCE_TRAJECTORY}

Final patch, if available:
{SOURCE_PATCH}

Execution/test result:
{SOURCE_RESULT}

## Candidate pool

{CANDIDATE_POOL}

Each candidate has the form:

[
{
"instance_id": "...",
"problem_statement": "..."
```

```
},
...
]

# Stage 1: Infer the source failure signature

Analyze the source trajectory ONCE before examining or ranking
candidates.

Reconstruct the actual failure and identify:

## 1. Intended solution requirements

What did the source problem actually require the agent to understand,
investigate, or implement?

## 2. Agent's main hypothesis

What did the failed agent believe was causing the problem?

## 3. First critical error

Identify the earliest important mistake that plausibly caused the
later failure.

Do not merely describe the final failed test or incorrect patch.

## 4. Failure pattern

Describe the observable abstract behavior through which the agent
failed.

Examples include:

* localized the bug to the wrong component;
* found the correct component but implemented the wrong semantics;
* fixed only one manifestation of a distributed issue;
* committed prematurely to a hypothesis and failed to revise it;
* ignored contradictory execution evidence;
* failed to inspect a relevant caller, callee, dependency, or
  alternate path;
* assumed API behavior without verification;
* misunderstood the specification;
* overfit to visible tests;
* failed to validate an otherwise plausible patch.

## 5. Root failure reason

Identify the deeper causal reason behind the failure.

Examples include:

* incorrect root-cause localization;
* insufficient semantic understanding;
* missing cross-component reasoning;
* missing dependency reasoning;
* specification misunderstanding;
* failure to update beliefs from evidence;
* incomplete causal reasoning;
* inadequate validation;
* incorrect API assumptions;
* planning failure;
* tool-use failure.
```

```
## 6. Abstract failure path

Reduce the trajectory to a repository-independent causal sequence.

For example:

visible symptom
-> plausible but incorrect hypothesis
-> premature localization
-> local modification
-> failure to inspect controlling dependency
-> incorrect or incomplete solution

## 7. Required failure triggers

Extract the minimal conditions another task would need in order to
plausibly induce the same first critical error, failure pattern, and
root failure reason.

Each trigger must:

* be repository-independent;
* be specific and falsifiable;
* be supported by an observation from the source trajectory;
* distinguish this failure from generic software-engineering
  difficulty.

Bad trigger:

"Requires reasoning across code."

Good trigger:

"The visible failing behavior points toward a local function, but
correct behavior is determined by an upstream state transition,
making premature local patching plausible."

For every trigger, record the supporting source evidence.

The resulting source failure signature must remain FIXED while
evaluating the candidate pool. Do not redefine the source failure
mechanism to make a candidate fit.

# Stage 2: Evaluate every candidate against the fixed failure
    signature

For each candidate, infer only what is justified by its problem
statement.

You do NOT have a candidate trajectory.

Therefore, never claim:

* the candidate agent actually failed;
* a hidden implementation definitely contains a particular call
  graph;
* a specific file or function is misleading unless stated;
* several locations require modification unless stated;
* particular tests will fail unless stated.

For each candidate determine:
```

```
## A. Likely solution requirements

What reasoning or investigation does the problem statement explicitly
or strongly imply?

## B. Candidate evidence for each source trigger

For every source failure trigger classify the candidate evidence as:

* direct: explicit evidence establishes essentially the same
  condition;
* partial: evidence supports part of the condition but important
  uncertainty remains;
* unknown: the problem statement does not reveal whether the
  condition exists;
* absent: the statement contradicts the trigger or clearly suggests
  a different mechanism.

For direct or partial, quote or closely paraphrase the exact candidate
text supporting the judgment.

Do not convert unknown into partial merely because the hidden
repository could contain the required structure.

## C. Same first-critical-error plausibility

Ask:

"If an agent attempted this candidate, could it plausibly make
approximately the same FIRST critical mistake as the source agent,
for the same reason?"

This is more important than whether both tasks belong to the same bug
category.

## D. Abstract failure-path similarity

Ask whether a plausible candidate failure could follow approximately
the same repository-independent causal sequence extracted from the
source.

## E. Alternative failure mechanism

Identify the most plausible way this candidate might instead challenge
or defeat an agent.

If this alternative mechanism is substantially different from the
source failure, lower the match.

# Stage 3: Score candidates

Score each candidate on the following dimensions.

Use only integer scores from 0 to 3.

## 1. Required-reasoning similarity

0 = fundamentally different reasoning
1 = only generic or weak overlap
2 = concrete substantial overlap with meaningful uncertainty
3 = essentially the same specific reasoning challenge

## 2. Failure-trigger similarity
```

```
0 = source trigger conditions are absent
1 = mostly generic, indirect, or merely possible
2 = at least one important trigger is concretely present and others
are plausible
3 = multiple defining source triggers are directly supported

## 3. Root-cause reproduction potential

0 = same underlying failure reason is very unlikely
1 = possible only at a generic level
2 = concretely plausible
3 = strongly supported by the candidate statement

## 4. Failure-path similarity

0 = plausible failure path would be different
1 = only broadly similar
2 = substantially similar abstract path
3 = approximately the same first critical error and causal failure
sequence are plausible

# Matching rules

## HIGH_MATCH

Assign HIGH_MATCH only if:

* all four causal scores are at least 2;
* at least three of the four causal scores are 3;
* the same first critical error is plausible for the same underlying
  reason;
* at least two source failure triggers have concrete candidate
  evidence;
* no major difference breaks the abstract failure path.

## POSSIBLE_MATCH

Assign POSSIBLE_MATCH only if:

* at least two causal dimensions are at least 2; and
* at least one specific source failure trigger has concrete candidate
  evidence.

## LOW_MATCH

Otherwise assign LOW_MATCH.

Use LOW_MATCH when the apparent similarity depends mainly on:

* lexical similarity;
* same repository or language;
* same framework;
* same general bug category;
* generic debugging;
* generic cross-file reasoning;
* generic API reasoning;
* generic validation;
* imagined hidden implementation details.

# Counterfactual tests

For every candidate that appears promising, perform both tests.
```

```
## Repository-removal test

Ask:

"If repository names, APIs, filenames, technologies, and
domain-specific nouns were removed, would this candidate still
present approximately the same reasoning trap?"

If not, lower the match.

## Negative-control test

Ask:

"Could this same explanation be applied almost unchanged to many
unrelated repository tasks?"

If yes, the evidence is too generic and the relevant score must be at
most 1.

# Pool-level ranking

After evaluating all candidates:

1. Rank candidates by failure-mechanism correspondence, not surface
   similarity.
2. Prefer candidates with explicit evidence for the defining source
   triggers.
3. Do not reward candidates merely because their problem statements
   contain more detail.
4. Do not force a match. It is valid for the entire pool to contain
   no HIGH_MATCH candidates.
5. Do not adjust the source failure signature after seeing attractive
   candidates.
6. If two candidates have similar scores, prefer the one with:

   * stronger evidence for the same first critical error;
   * more direct trigger correspondences;
   * fewer unsupported assumptions;
   * a more similar abstract failure path.

# Output

Return ONLY valid JSON.

{
"source_id": "...",

"source_failure_signature": {
"intended_solution_requirements": ["..."],
"main_agent_hypothesis": "...",
"first_critical_error": "...",
"failure_pattern": "...",
"root_failure_reason": "...",
"abstract_failure_path": "...",
"required_failure_triggers": [
{
"trigger_id": "T1",
"trigger": "...",
"source_evidence": "..."
}
]
```

```
},

"ranked_candidates": [
{
"rank": 1,
"instance_id": "...",

  "candidate_analysis": {
    "likely_solution_requirements": ["..."],
    "same_first_critical_error_plausible": true,
    "plausible_failure_path": "...",
    "important_difference": "...",
    "alternative_failure_mode": "..."
  },

  "trigger_alignment": [
    {
      "trigger_id": "T1",
      "alignment": "direct",
      "candidate_evidence": "..."
    }
  ],

  "scores": {
    "required_reasoning_similarity": 0,
    "failure_trigger_similarity": 0,
    "root_cause_reproduction_potential": 0,
    "failure_path_similarity": 0
  },

  "negative_control_result": "...",
  "label": "HIGH_MATCH",
  "confidence": 0.0,
  "justification": "..."
}

],

"best_matches": [
"INSTANCE_ID_1",
"INSTANCE_ID_2"
],

"pool_summary": {
"high_match_count": 0,
"possible_match_count": 0,
"low_match_count": 0,
"overall_observation": "..."
}
}

Confidence must be between 0 and 1 and represents confidence in the
assigned label, NOT match strength.

Sort ranked_candidates from strongest to weakest failure-mechanism
match.

If the pool is very large, return detailed analyses only for the top
{TOP_K} candidates and return a compact entry for all remaining
candidates containing:

{
"instance_id": "...",
```

```
"label": "...",
"scores": {...},
"short_reason": "..."
}

Be conservative. The objective is not to maximize the number of
matches. The objective is to identify instances that genuinely test
whether the specific causal failure observed in the source trajectory
generalizes.
```

Our mapping is conducted over large and heterogeneous candidate pools spanning both repository-level software repair and terminal-based tasks. In our local snapshot, SWE-smith contains 59,136 candidate tasks from 222 repositories, while SWE-ReBench contains 32,079 tasks from 3,617 repositories across 20 programming languages. The terminal-task baselines are also substantial in scale, with 15,048 LEGO tasks and 3,255 SETA-Synth environments drawn from five source families, including Stack Overflow, Ask Ubuntu, Unix/Linux Stack Exchange, NL2Bash, and Kaggle notebooks (Shen et al., 2026). For each baseline, we map source failure trajectories to candidate tasks. The resulting mappings are not concentrated on only a few repeated targets: SWE-smith selects 48 distinct targets from 24 repositories, SWE-ReBench selects 48 distinct targets from 33 repositories and eight programming languages, and SETA selects 48 distinct targets spanning four of its five source families. In particular, SWE-ReBench mappings cover Python, Java, JavaScript, Elixir, C, Go, Kotlin, and Rust, demonstrating that the retrieved matches extend beyond a single language ecosystem. SETA likewise covers multiple task origins, although its selected mappings are skewed toward Stack Overflow , indicating that candidate-pool diversity does not necessarily translate uniformly into selected-match diversity. Overall, these results show that our mapping procedure operates over candidate spaces ranging from thousands to tens of thousands of tasks and retrieves matches across diverse repositories, languages, and task domains, rather than relying on a small or homogeneous set of target instances.

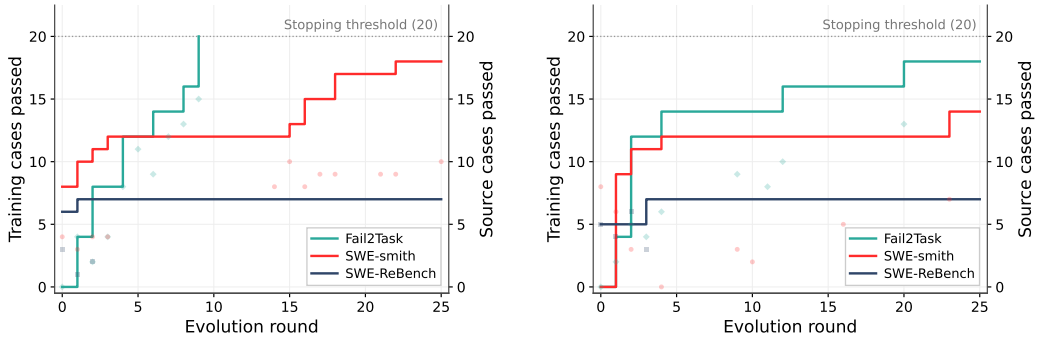
Pass@3 Evaluation. Source-task, training-task, and benchmark pass rates in Table 3 use three solving attempts per task. A task passes if at least one of its three attempts succeeds. Train–source mismatch compares these task-level pass@3 outcomes for each paired task. RQ3 uses the same criterion for constructed and source tasks.

Benchmark Evaluation Sets. For benchmark evaluation, we use all 300 tasks in SWE-bench Lite. For Terminal-Bench 1.0, we use the 53 tasks remaining after excluding tasks that overlap with Terminal-Bench 2.0.

Harness Evolution and Data Membership. The common AutoSaddler configuration uses 48 constructed or matched training tasks and at most 25 optimization rounds. A candidate harness is accepted if its training score is at least as high as that of the currently accepted harness; otherwise, it is rejected. Optimization stops after 25 rounds or once at least 20 training tasks pass, whichever occurs first. We report the last accepted harness, which attains the highest training score among accepted rounds. Source-test evaluation is performed only after optimization and does not inform candidate acceptance or checkpoint selection. There is no separate validation split, and no validation score is computed. Updates are restricted to agent code; evaluation programs, tests, task data, dependencies, and runtime environments cannot be modified. The task-solving model and its weights also remain fixed. The resulting agents are evaluated on their training tasks, original failure-source tasks, and standard benchmarks. Source-task performance measures targeted transfer, not evaluation on failures unseen by reconstruction.

Evolution Curves. Figure 2 shows the evolution curves for mini-SWE-agent and SWE-agent on the SWE-bench setting. Solid lines report passed cases on the training set (left axis), while light-colored markers report source-task passes at the displayed checkpoints (right axis). Round 0 denotes the unevolved harness. All source instances are selected from failed runs, so their selection-time pass count is zero by construction. Fresh runs can nevertheless pass some cases because solving is stochastic, including at round 0; this limitation motivates the failure-persistence and cross-run failure-pattern analyses in Appendix A. The dotted line marks 20 passed training tasks out of 48,

rather than a 20% pass rate. Horizontal training segments retain the incumbent score between displayed checkpoints; no additional source-task evaluations are implied by these segments.



(a) mini-SWE-agent (b) SWE-agent

Figure 2: **Training and source-task performance during harness evolution.** SWE-bench panels for (a) mini-SWE-agent and (b) SWE-agent. Solid lines show training-task pass counts (left axis); light-colored markers show source-task pass counts (right axis). Round 0 denotes the unevolved harness. The horizontal dotted line marks the stopping threshold of 20 training passes.

D RQ3 Outcome Scores and Annotation Protocol

Paired Evaluation. The Terminal cohort contains 48 pairs evaluated at Terminus 2 candidate 4. Each constructed and source task has three solving attempts with this candidate. We report pass@3: a task passes if at least one of its three attempts succeeds.

Table 6: Paired outcome scores for the Terminal cohort. Every cell gives count/denominator and percentage.

Measure	Terminus 2 C4
Constructed: pass@3	19/48 (39.6%)
Source: pass@3	10/48 (20.8%)
Both sides have a pass	3/48 (6.3%)
Constructed only	16/48 (33.3%)
Source only	7/48 (14.6%)
Neither side has a pass	22/48 (45.8%)
Outcome mismatch	23/48 (47.9%)

Trajectory Evidence. Two authors independently and manually assigned error labels to all 48 Terminal cases. For each case, they examined the constructed task, the corresponding source task, the execution trajectories of both tasks, and the patches and harness self-evolution trajectories produced by AutoSaddler. Their annotations recorded actions, verification, omitted obligations, and supporting evidence anchors. Cohen's $\kappa = 0.68$ (Cohen, 1960) was calculated from the two authors' initial labels before discussion. The authors then resolved disagreements through discussion to determine the final error labels. The five category frequencies and representative mismatch cases in RQ3 use the Terminal cohort only.

Harness-Refinement Illustration. The separate `django__django-11141` illustration uses mini-SWE-agent's C5 optimization record and its fixed constructed-task snapshot. The stored candidate diff adds shared validation memory, while case-level feedback records the training change and the candidate record documents acceptance. The source failure and construction replay establish the background failure pattern; they are distinct from candidate-C5 evaluation attempts.

Primary-Barrier Frequencies. The author-annotated primary-barrier frequencies over all 48 Terminal pairs are: insufficient verification, 19/48; reduced task scope, 14/48; limited generalization, 8/48; added or implicit requirements, 4/48; and API differences, 3/48. These five categories cover all 48 pairs. Frequencies are not conditioned on success or outcome mismatch, and a primary barrier does not rule out secondary construction or model factors. The main text illustrates these categories with cases from the two mismatch directions. These examples do not partition the mismatched pairs by category or make the categories exclusive to either direction. Insufficient verification and limited generalization describe behavior under the evaluated harness, not persistent model incapacity or a universal ceiling on harness improvement. Reduced task scope and added requirements also identify construction/transfer boundaries that mechanism-level acceptance alone does not resolve.