TRAJFIX: CORRECTING FAILED TRAJECTORIES TO SCALE VERIFIABLE SOFTWARE-ENGINEERING DATA

Anonymous authors
Paper under double-blind review

ABSTRACT

Training an agent on real software-engineering (SWE) tasks requires trajectories that pass the tests. However, agents struggle to produce such a trajectory on hard tasks. Failed trajectories are normally thrown away, so most of the compute spent is wasted. Worse, the discarded trajectories come from the hard tasks and provide particularly valuable training signals. To make use of these failed trajectories, we present **TRAJFIX**, a multi-agent framework that repairs failed trajectories. It keeps the correct prefix and repairs only the first step that a diagnosis agent identifies as wrong, letting the agent continue the rollout until it submits a solution. Because the correction relies on oracle information that could leak the answer, a hallucination check identifies reasoning jumps and sends those corrections back for regeneration. Under a matched rollout budget per task, TrajFix improves the resolve rate by **44.0 points** over resampling while reducing the cost of producing a trajectory usable for training by **47.6%**. On large-scale GitHub tasks, we achieve a recovery rate of 53.60% and turn 19,171 failed trajectories into training data. Compared with training on the original successful trajectories alone, adding recovered trajectories consistently improves SWE performance. For Qwen3-8B, SWE-Bench Verified improves by 5.25 points, SWE-Bench Multilingual by 6.16 points, and SWE-Bench Pro by 5.75 points.

1 INTRODUCTION

Large language models (LLMs) have advanced rapidly and can now solve repository-level programming tasks in real-world software projects on their own (Yang et al., 2024). Benchmarks of real-world issue solving, such as SWE-Bench (Jimenez et al., 2024) and SWE-Bench Pro (Deng et al., 2025), have become the standard for measuring the coding ability of LLMs. However, using coding trajectories to train LLMs still faces an important challenge. A large proportion of the generated trajectories do not pass the Fail-to-Pass (F2P) tests and are therefore discarded (Yuan et al., 2023), resulting in a large waste of data collection costs. The main bottleneck is no longer the number of tasks collected, but the number of trajectories that can be used for training (Ma et al., 2026). The trajectories used for training come from relatively easy tasks, while hard tasks that span several files and several tests are filtered out, so the learning signal of many hard tasks is lost (Lv et al., 2026).

More and more engineers are paying attention to the value of failed trajectories. Existing work mostly uses the cause of the failure as a hint for the next rollout to avoid repeating the same wrong pattern (Da et al., 2025; Lv et al., 2026; Zhao et al., 2026a), but is essentially resampling and still wastes the cost spent on generating the valid prefix of the failed trajectory. Researchers have also noticed the value of valid prefixes. By replaying the prefix and resampling from there, instead of resampling from scratch (Zhu et al., 2026; Ding & Zhang, 2026), they reduce some costs and speed up inference. However, these approaches do not give effective signals to solve tasks beyond the ability of the models.

To address the above pain points, we present TRAJFIX, the data-production framework that corrects failed trajectories. We design a multi-agent architecture whose core is an iterative loop of diagnosis, correction, replay, and verification. Given a failed trajectory and the privileged information from the corresponding pull request, TRAJFIX first locates the first step that a diagnosis agent identifies as wrong. It then starts a fresh container and replays all prefix steps up to the wrong step. Next, TRAJFIX corrects the wrong step based on the diagnosed error and executes the corrected step.

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

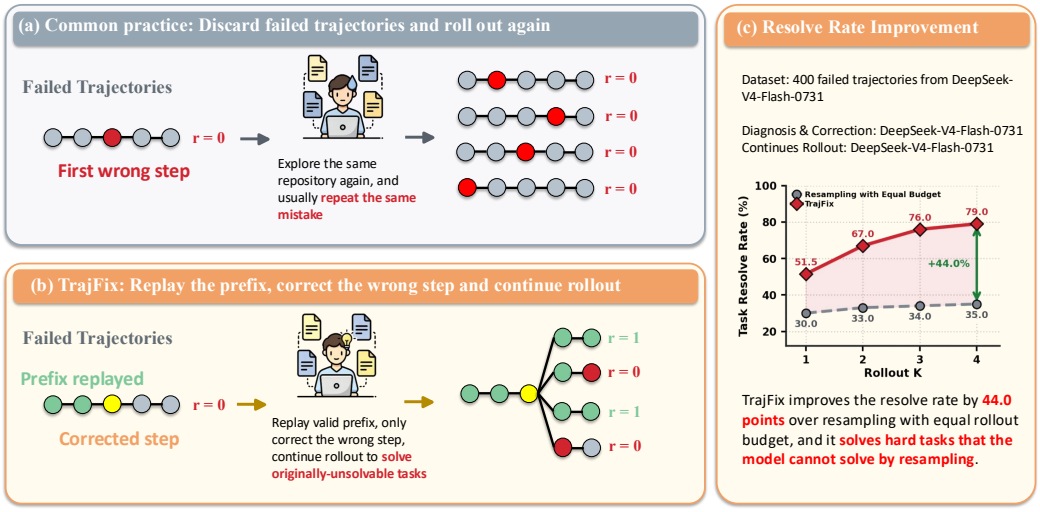

Figure 1: **Two ways to handle failed trajectories.** (a) Discard the failed trajectories and explore the same repository again, which often repeats the same mistake. When the model has limited ability, some hard tasks remain unsolved no matter how many rollouts they are given. (b) TRAJFIX replays the correct prefix, corrects the wrong step, and then lets the model continue the rollout on its own. This can solve tasks the model originally could not solve. (c) On the same 400 failed trajectories, with four rounds of diagnosis and one rollout per round, TRAJFIX improves the resolve rate by 44.0 points over resampling with equal rollout budget, and it solves hard tasks that the model cannot solve by resampling.

Finally, the agent continues the task on its own until it submits a solution. We add two mechanisms on top of this loop. (1) *Hallucination detection*. Since the diagnosis agent has access to privileged information, a corrected step may contain conclusions that a normal rollout could not reach, so we check every corrected step and regenerate those that leak the information or jump ahead. (2) *Multi-point progressive repair*. One correction is often not enough, so we repeat the loop for several rounds, each round diagnosing the trajectory the previous round produced and correcting a later step.

Figure 1 contrasts the two ways of handling failed trajectories. Resampling discards the whole trajectory and explores the repository again, and usually repeats the same mistake; TRAJFIX replays the prefix, corrects the wrong step, and rolls out from the corrected step. Given the same rollout budget per task, TRAJFIX improves the resolve rate by 44.0 points over resampling while reducing the cost of producing a trajectory usable for training by 47.6% (Section 3.2.2). After fine-tuning open-source models in recovered trajectories, the models achieve a clear performance gain in SWE tasks, which fully verifies the effectiveness of TRAJFIX.

The main contributions of this paper are as follows:

- **TRAJFIX: a data-production framework that corrects failed trajectories.** We build a multi-agent framework for diagnosis, correction, replay, and verification, with hallucination detection and multi-point progressive repair as additional mechanisms (Section 2). It enables the model to solve tasks that it could not solve, while reducing the cost of producing successful trajectories (Section 3.2).
- **Comparison of diagnostic and correction capabilities of mainstream models.** We compare mainstream commercial and open-source models on diagnosing and correcting the wrong step of failed trajectories (Section 4.1), and analyse the error types of failed trajectories and their repair success rates in detail (Section 4.2).
- **Correcting failed trajectories at scale improves downstream SWE ability.** We use corrected trajectories to fine-tune open-source models at two model sizes and observe consistent gains on SWE-Bench Verified, Multilingual and Pro (Section 4.3). The control experiments show that the gain comes mainly from correcting a failed trajectory into a right one, not merely from a richer set of data.

2 TRAJFIX

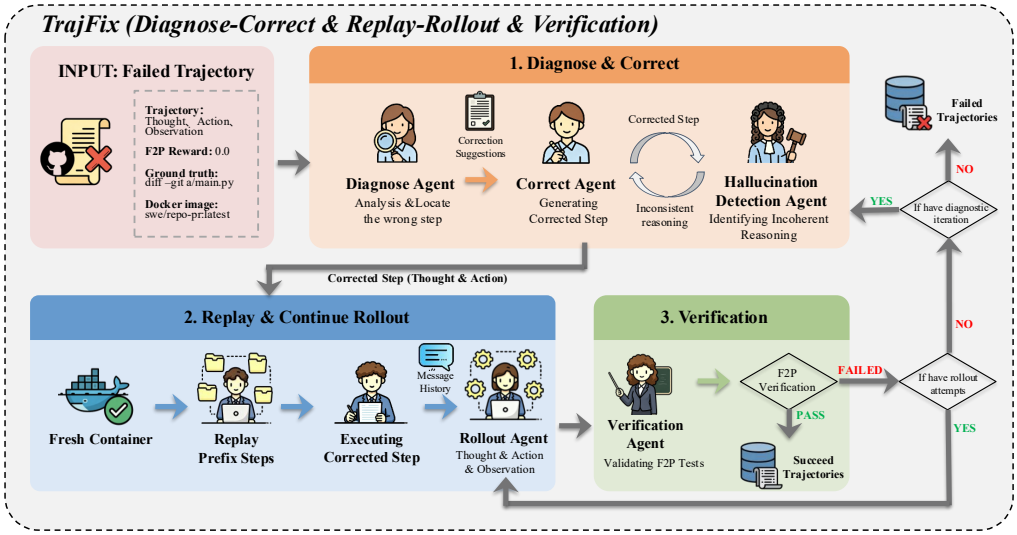

Figure 2: **Overview of TRAJFIX.** A failed trajectory enters the pipeline with the thought, the action and the observation of every step, together with the golden patch of the pull request. The diagnosis agent locates the first wrong step and gives a fix suggestion. The correction agent turns the suggestion into a corrected step. The hallucination detection agent audits the corrected step for a reasoning jump. A fresh sandbox then starts, the steps before the wrong step are replayed, and the corrected step is executed. The rollout agent continues the rollout on its own, and the verification agent runs F2P tests. After a failure we check the remaining attempts of rolling out: if left means replay and roll out again, no left means the shortest failed trajectory from the previous round becomes the subject of the next diagnosis. This repeats until the diagnosis rounds run out. The output is split into two trajectory sets, recovered and still failing.

Diagnosis agent. The diagnosis agent locates the first wrong step. Its input is the whole failed trajectory, with the thought, the action, and the observation, plus the golden patch. Its output is the index of the first wrong step, the error type, the error reason, and a fix suggestion. Two details matter at this stage. **(1) We identify the first wrong step rather than the most obvious one.** Later errors are often consequences of earlier decisions, so repairing a later consequence without correcting the root cause leaves the underlying error unresolved. **(2) The golden patch is used to attribute the error, not as a source to be copied.** The error analysis must not quote the golden patch. Without this constraint, the agent may produce explanations such as "the wrong step edits Function A, while the golden patch edits Function B; therefore, the step is wrong," causing the corrected step to leak information from the golden patch. Instead, all concrete evidence in the error analysis must come from the prefix, including files the agent has actually opened, observations from previous actions, and results from earlier test runs.

Correction agent. The correction agent turns the fix suggestion into a corrected step. The corrected step must remain coherent with the prefix, so we set two constraints on its generation. **(1) It must be derivable from the prefix.** The thought of the corrected step must refer to the observations of the earlier steps, and its output form is "given the results of the earlier steps, I think this step needs to perform this action". The corrected step is also checked for valid formatting, and any step that cannot be parsed is regenerated. **(2) Minimal change.** The corrected step may change only one function in one file. Applying the whole golden patch in a single step or rewriting a whole source file is not allowed, because either one shifts the rollout pattern of the model too far. The correction agent finally outputs the corrected step with its thought and its action.

Hallucination detection agent. The hallucination detection agent judges whether the corrected step is consistent with the prefix. A corrected step with reasoning that the context does not support can still pass the F2P test. In real training such a reasoning jump damages the behavior pattern of

the model, because the model cannot produce that kind of illogical thought without the privileged information. The input of hallucination detection agent is the prefix and the corrected step, and the agent labels the corrected step NATURAL — it can be derived from the prefix steps or JUMP — it leaks knowledge of the golden patch. A JUMP verdict sends the step back for generation again.

Rollout agent. The rollout agent replays the prefix, executes the corrected step, and lets the model continue the rollout. It starts a fresh container with an Ubuntu environment and pulls the code repository of the task into the working directory. The scaffold for the interaction between the model and the container is adapted from OpenHands (Wang et al., 2025) and contains three tools: a bash executor, a file editor, and a task submitter. Inside the container, the agent first replays the actions of the prefix, so that the file system and the environment dependencies match the state at that time, and then executes the corrected step. The thoughts, actions, and observations from the prefix and corrected step are merged into the conversation history for the next rollout. The model then continues the rollout until it submits the task or reaches the maximum number of turns.

Verification agent. When the model submits the task, the verification agent turns the actions into $\mathrm{Output_Patch}$ with the `git diff` tool. It then starts a fresh container with an Ubuntu environment, applies $\mathrm{Test_Patch}$ and runs the tests, which must fail, and then applies $\mathrm{Output_Patch}$ and runs the tests again, which must pass. This trajectory is marked as passing F2P verification. Otherwise, the trajectory is marked as failed.

Multi-point progressive repair. After correcting the first wrong step, the model may still fail at a later step. Re-diagnosing the original trajectory is ineffective because the diagnosis agent would identify the step that has already been corrected. We select the shortest failed trajectory because it uses fewer turns on unnecessary steps, leaving more turns for the next rollout. Since the corrected prefix is preserved, each round identifies and repairs the next error, progressively moving deeper into the trajectory.

Appendix A gives the implementation of every stage, the input and the output of every agent (Table 5) and the algorithm (Algorithm 1); Appendix B gives the prompts of agents.

3 EXPERIMENTS

3.1 EVALUATION METRICS

$\mathrm{TF@}M\times N$. We define $\mathrm{TF@}M\times N$ as the proportion of failed trajectories recovered with M diagnosis rounds and N continuation rollouts in each round. Let $\mathcal{D}_{\text{fail}}$ denote the set of failed trajectories, x_i the task associated with failure i, and $\tau'_{i,m,n}$ the trajectory generated for failure i by the n-th continuation rollout in round m. The verifier $\mathcal{V}$ returns 1 if a trajectory passes the F2P tests for its task and 0 otherwise. The formula is

$$\mathrm{TF@}M\times N \;=\; \frac{1}{|\mathcal{D}_{\text{fail}}|}\sum_{i=1}^{|\mathcal{D}_{\text{fail}}|}\max_{m\le M,\; n\le N}\; \mathcal{V}\big(x_i, \tau'_{i,m,n}\big). \tag{1}$$

Performance and Cost Metrics. The resolve rate is measured at the task level: a task is considered solved if at least one of its trajectories passes the F2P tests. The trajectory success rate is measured at the trajectory level and denotes the proportion of trajectories that pass the F2P tests. These metrics differ because a single task may have multiple trajectories, as resampling generates multiple rollouts for the same task, while repair performs multiple rounds on the same failed trajectory. For cost, we report the average number of input and output tokens per trajectory, the cost per successful trajectory, the cost per solved task, and the total cost per run. The cost per successful trajectory represents the cost of producing one unit of successful training data.

Additional Evaluation Metrics. We report the hallucination rate as an additional metric to assess the reliability of the recovered data. It measures the proportion of recovered trajectories whose corrected steps are judged to contain reasoning jumps. A lower hallucination rate indicates more reliable recovered data. For fine-tuned models, we report $\mathrm{Avg@}K$ and $\mathrm{Pass@}K$. $\mathrm{Avg@}K$ is the mean resolve rate across K evaluations and measures performance stability. $\mathrm{Pass@}K$ is the proportion of tasks solved by at least one of the K evaluations and measures task coverage.

3.2 COMPARISON OF TRAJFIX AND RESAMPLING

3.2.1 EXPERIMENTAL SETUP

Task Pool. We construct a task pool of 55,366 task instances from 5,101 repositories. The pool covers ten programming languages (Python, Go, JavaScript, TypeScript, Java, Rust, PHP, Ruby, C, C++). The repositories in this pool do not overlap with those included in SWE-Bench Verified (Chowdhury et al., 2024), SWE-Bench Multilingual (Zan et al., 2026) or SWE-Bench Pro (Deng et al., 2025), preventing data contamination at the repository-level. Appendix G.1 gives the composition of the language and the collection statistics. For comparison of TrajFix and resampling, we use 400 failed trajectories generated by DeepSeek-V4-Flash-0731 (Xu et al., 2026) as experimental input, corresponding to 400 distinct tasks. The dataset consists of 200 tasks with $\text{Pass@4} = 0$, for which all four rollouts fail, and 200 tasks with $0 < \text{Pass@4} < 1$, for which at least one but not all four rollouts succeed.

Experimental Description. We compare TrajFix with three resampling settings. **TrajFix** uses DeepSeek-V4-Flash-0731 for diagnosis, correction, and continuation rollout, with four diagnosis rounds and one rollout per round. We use DeepSeek-V4-Pro-0813 (Xu et al., 2026) for hallucination detection throughout all experiments. **Resampling (matched-k rounds)** uses the same number of rollouts as TrajFix for each task, where k is the round in which TrajFix recovers the task, or $k = 4$ if the task is not recovered. **Resampling (early stop)** repeatedly resamples each task and stops once a successful trajectory is obtained, with a maximum of four rounds. **Resampling (4 rounds)** performs four independent rollouts for each task, providing a fixed four-round resampling setting for comparison.

3.2.2 COMPARISON OF RESOLVE RATE AND COST

Table 1: TrajFix against three resampling settings in terms of resolve rate and cost.

Method	Tasks		Trajectories			Tokens / traj. (k)		Cost ($)		
	# Resolved	Resolved (%)	# Succ.	# Traj.	Succ. (%)	In	Out	Total	/task	/succ. traj.
Resampling (matched-k rounds)	140	35.00	178	822	21.65	4,243.7	42.0	250.9	0.6273	1.4097
Resampling (early stop)	190	47.50	190	1,142	16.64	3,538.7	**36.7**	295.0	0.7375	1.5524
Resampling (4 rounds)	190	47.50	472	1,600	29.50	3,867.4	38.7	450.3	1.1258	0.9541
TRAJFIX	**316**	**79.00**	316	822	**38.44**	**3,104.4**	50.0	**233.5**	**0.5838**	**0.7390**

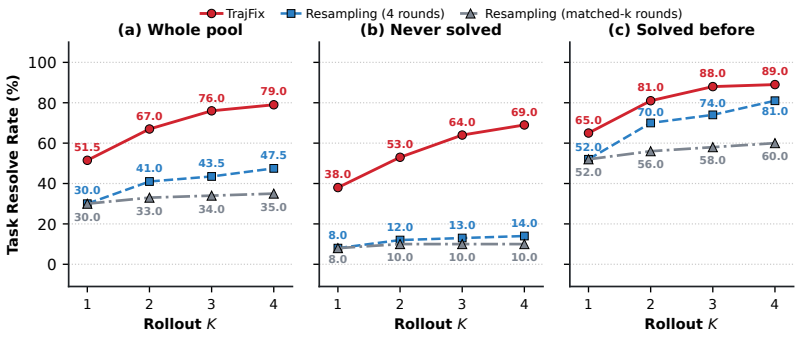

Figure 3: **Resolve rate across the full task pool and both subsets.** TRAJFIX is compared with two resampling settings: Resampling (4 rounds), whose cumulative resolve rate corresponds to $\text{Pass@}K$, and Resampling (matched-k rounds). The three panels show (a) the full task pool, (b) never-solved tasks with $\text{Pass@4}=0$, and (c) previously solved tasks with $0 < \text{Pass@4} < 1$.

TrajFix improves the task resolve rate compared to resampling in different rollout settings. Table 1 reports the comparison of TrajFix with three resampling settings in resolve rate and cost. Under a matched rollout budget per task, TrajFix achieves a 79.0% resolve rate, compared with 35.0% for Resampling (matched-k rounds), corresponding to a 44.0-point improvement. Even without strictly matching the rollout budget, TrajFix also achieves a higher resolve rate than Resampling (4 rounds), which reaches only 47.5%. This improvement comes from diagnosing and correcting the

first wrong step while preserving the valid prefix, which allows TrajFix to continue the rollout from a more promising state and recover tasks that repeated independent resampling cannot solve.

TrajFix reduces the cost of producing successful trajectories. Under a matched rollout budget per task, TrajFix reduces the cost per successful trajectory by 47.6%. Since a task may produce multiple successful trajectories, we further compare the cost in an early-stop setting, where resampling continues until a successful trajectory is obtained. TrajFix reduces the cost per successful trajectory by 52.4% compared with Resampling (early stop). TrajFix produces more output tokens per trajectory than all resampling settings, due to the additional diagnosis, correction, and hallucination detection stages. However, the overall cost is lower because the cost saved by replaying the valid prefix exceeds the additional cost of these stages.

TrajFix can solve tasks that the model cannot solve through independent resampling. Figure 3 compares the resolve rate of TrajFix with two resampling settings in the full task pool and two subsets. Across all three groups, TrajFix achieves a higher resolve rate than all resampling settings at each round. In the solved-before subset, where the model can solve the task with at least one rollout, TrajFix and resampling both maintain relatively high resolve rates. In contrast, on the never-solved subset, TrajFix substantially outperforms all resampling settings, showing that it can recover tasks that independent resampling fails to solve. Resampling is limited by the model's ability to solve the task independently, so hard tasks may remain unsolved even with more rollouts. TrajFix instead diagnoses and corrects errors in failed trajectories and continues from their valid prefixes, allowing it to solve more hard tasks.

3.3 Mechanism ablation

3.3.1 Experimental setup

Ablation Study. We use the same test set as in Section 3.2, consisting of 400 failed trajectories. We conduct mechanism ablations on three components of TrajFix: **Admissibility Constraint (AC)**, which ensures that each corrected step is derivable from the prefix; **Hallucination Detection (HD)**, which detects reasoning jumps and information leakage in corrected steps; and **Multi-point Progressive Repair (MPPR)**, which enables multiple rounds of diagnosis and correction to progressively repair later wrong steps.

3.3.2 Ablation results

Table 2: **Mechanism ablation Results.** We report the resolve rate at each rollout budget, the hallucination rate of the corrected steps, and the TF@4 on the two subsets: *Solved before* ($0 < \text{Pass@4} < 1$) and *Never solved* ($\text{Pass@4} = 0$).

Method	Components			Resolve Rate (%)				Hallu.	TF@4 by subset (%)	
	AC	HD	MPPR	TF@1	TF@2	TF@3	TF@4	(%)	Solved before	Never solved
TrajFix (full)	✓	✓	✓	51.50	**67.00**	**76.00**	**79.00**	**2.53**	89.00	**69.00**
TrajFix w/o {MPPR}	✓	✓	✗	51.50	63.50	70.50	74.50	4.55	89.00	60.00
TrajFix w/o {HD, MPPR}	✓	✗	✗	52.50	64.00	72.50	75.50	8.04	90.00	61.00
TrajFix w/o {HD, MPPR, AC}	✗	✗	✗	**56.50**	66.00	73.50	77.00	14.23	**91.00**	62.50

(1) Strict admissibility constraints and hallucination detection improve the reliability of the generated data. With strict admissibility constraints and the hallucination detection mechanism, the hallucination rate decreases from 14.23% to 2.53%. This shows that the higher resolve rate of TrajFix w/o {HD, MPPR, AC} comes from the leakage of oracle information. Trajectories with reasoning jumps are unreliable training data because they may teach the model to rely on oracle information that is unavailable during inference.

(2) Multi-point progressive repair increases the resolve rate further. The gain from MPPR comes mainly from the subsequent rounds, with TrajFix reaching 79.0% at TF@4, compared with 74.5% without MPPR. The improvement is greater in tasks that resampling has never solved, where TrajFix reaches 69.0%, compared with 60.0% without MPPR. However, the gain from each additional round gradually decreases, with the overall resolve rate increasing by 15.5, 9.0, and 3.0 points from the first round to the fourth round.

4 ANALYSIS

4.1 THE ABILITY TO DIAGNOSE AND CORRECT ERRORS IN MAINSTREAM MODELS

Test Set and Setup. To better assess the model's ability to diagnose and correct errors in failed trajectories, we select more challenging failed trajectories with a larger number of patch lines in the golden patches. The diagnosis and correction agents use the model under evaluation, while DeepSeek-V4-Flash-0731 (Xu et al., 2026) is fixed for the continuation rollout. The test set contains 400 failed trajectories, evenly distributed in eight programming languages, with an equal proportion for each language. We run $M{=}3$ diagnosis rounds with $N{=}1$ rollout per round. We report the same metrics as in Section 3.3, including the resolve rate and the hallucination rate.

Table 3: **Comparison of mainstream models in diagnosing and correcting errors.**

Model	Resolve Rate (%)			Hallu.	TF@3 by subset (%)	
	TF@1	TF@2	TF@3	(%)	Solved before	Never solved
Claude-Opus-5 (Anthropic, 2026)	**41.75**	**56.25**	**66.00**	2.51	**81.50**	50.50
Kimi-K3 (Team et al., 2026)	41.25	55.75	65.25	2.53	77.00	**53.50**
DeepSeek-V4-Flash-0731* (Xu et al., 2026)	38.25	51.50	60.75	2.26	76.00	45.50
DeepSeek-V4-Pro-0813 (Xu et al., 2026)	39.75	50.75	59.75	3.01	77.00	42.50
GPT-5.6-Sol (OpenAI, 2026)	36.00	48.25	54.75	**1.00**	71.50	38.00
Qwen3.8-Max† (Yang et al., 2025)	26.50	34.50	39.75	3.38	57.50	22.00

* the self-repair row: the model repairing the trajectory is the policy that produced it. † 75 of its 400 trajectories end in a tool-call format error and never reach the verifier, so they count as unresolved here.

Claude-Opus-5 achieves the best overall performance. It achieves a resolve rate of 66.00%, followed by Kimi-K3 and DeepSeek-V4-Flash-0731. Most of the failures Claude-Opus-5 brings back come from tasks the policy has solved before, so the model mainly makes those tasks pass in a stable way. On tasks the policy has never solved, the resolve rate of every model is much lower than on tasks it has solved before, and Kimi-K3 is the best there. Qwen3.8-Max needs a separate note. It does not follow the instruction, and after many retries it still fails to produce a valid tool-call format, so some of its tasks are marked wrong at once. The hallucination rate of every model is already very low, so most corrected steps follow the steps before them, and the effect on the resolve rate is small. GPT-5.6-Sol has the lowest rate at 1%.

4.2 ERROR TYPES AND RECOVERY RESULTS

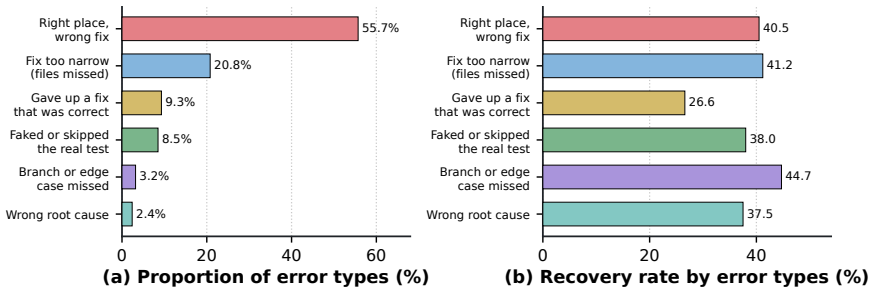

Figure 4: **The proportion and the recovery rate for each type of error.**

We use the same task pool as in Section 3.2, consisting of 55,366 task instances from 5,101 repositories. Using DeepSeek-V4-Flash-0731, we collect 37,472 successful trajectories and obtain 17,894 failed trajectories. We perform a diagnostic round on each failed trajectory with both DeepSeek-V4-Flash-0731 and Kimi-K3. Among these trajectories, 14,544 receive the same error type from both models. We then analyse the distribution of error types and their recovery rates among the trajectories with consistent diagnoses from both models. As shown in Figure 4, the proportions of different error types and their corresponding recovery rates are reported to one decimal place. Appendix F provides descriptions and representative examples for each error type.

Most failed trajectories reach the correct location but fail in the modification step. Right place but wrong fix accounts for 55.7% of all errors, while fix too narrow accounts for another 20.8%. Together, these two types account for 76.5% of the diagnosed errors, showing that most failures have already identified the relevant location but make an incorrect or incomplete modification. This also means that their prefixes contain useful information that can be preserved during repair.

Repair is more effective for errors that are close to the correct solution. Missed branch or edge case has the highest recovery rate at 44.7%, followed by fix too narrow at 41.2% and right place but wrong fix at 40.5%. These errors can often be corrected by changing only the wrong step. In contrast, gave up a fix that was correct, wrong root cause, and faked or skipped the real test are harder to repair because they often involve an incorrect earlier decision or a deeper misunderstanding that cannot be resolved by changing a single step.

4.3 Downstream SWE ability of the recovered trajectories

We use the same task pool consisting of 55,366 task instances from 5,101 repositories. Using DeepSeek-V4-Flash-0731, we collect 37,472 successful trajectories, which form the **Solved set**. We apply TrajFix to these failed trajectories twice, with two diagnosis rounds and one continuation rollout per round in each run. DeepSeek-V4-Flash-0731 is used for diagnosis, correction, and continuation rollout. The two runs produce 19,171 recovered trajectories, which form the **Recovered set**. We use the 19,171 corresponding failed trajectories as the **Unrepaired set**. For comparison, we independently resample the failed trajectories for four rounds, producing 9,594 successful trajectories, which form the **Resampled set**. We train models on these datasets and evaluate them with the OpenHands (Wang et al., 2025) scaffold on SWE-Bench Verified, SWE-Bench Multilingual and SWE-Bench Pro. We also report patch size of the training data and the mean patch size of SWE-Bench Pro tasks grouped by whether they are solved by both models, only the model trained with Solved data, only the model trained with Solved and Recovered data, or neither.

Table 4: **Downstream SWE Performance across Different Training Sets.**

Model	Training data	SWE-Bench Verified		SWE-Bench Multilingual		SWE-Bench Pro	
		AVG@4	Pass@4	AVG@4	Pass@4	AVG@4	Pass@4
Qwen3-8B	Solved	53.25	70.00	32.67	52.67	30.67	46.37
	+ Resampled	53.65	70.60	35.67	55.67	32.15	47.47
	+ Unrepaired	55.85	69.80	37.23	55.67	35.57	49.93
	+ Recovered	**58.50** (+5.25)	**73.60** (+3.60)	**38.83** (+6.16)	**56.00** (+3.33)	**36.42** (+5.75)	**50.34** (+3.97)
Qwen3-30B-A3B	Solved	67.50	79.20	57.00	74.00	46.55	60.05
	+ Resampled	67.55	77.80	56.33	72.33	46.48	59.78
	+ Unrepaired	68.05	78.00	57.92	74.67	47.44	61.01
	+ Recovered	**68.90** (+1.37)	**79.60** (+0.40)	**58.75** (+1.75)	**74.67** (+0.67)	**47.91** (+1.36)	**61.42** (+1.40)

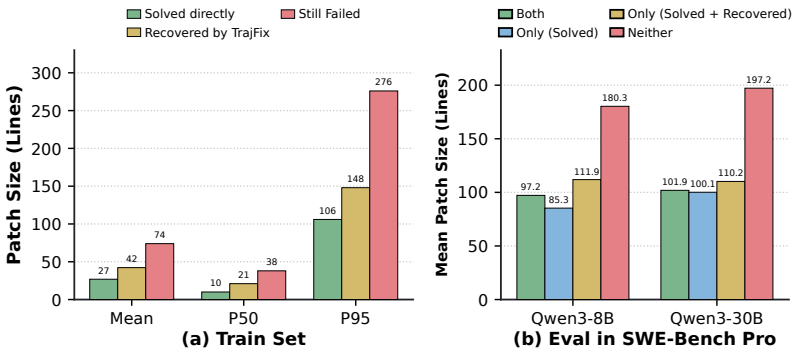

Figure 5: **Patch Size of Train Set and SWE-Bench Pro Tasks.**

Adding trajectories recovered by TrajFix to the Solved set improves performance on both model scales and on all three benchmarks. Table 4 reports the results. The gains are larger for Qwen3-8B than for Qwen3-30B-A3B, as the stronger model already achieves higher performance

and therefore has less room for improvement. Adding the corresponding Unrepaired trajectories also provides some gains, indicating that recoverable failed trajectories contain useful training signals. Adding Resampled trajectories provides the smallest gains, as resampling misses training signals from hard tasks it cannot solve.

Figure 5 further shows that **the model trained on the Solved set augmented with Recovered trajectories solves harder tasks than the model trained on the Solved set alone**. Tasks that remain unsolved have the largest patch sizes, suggesting that patch size is a useful proxy for task difficulty. Recovered trajectories also have larger patch sizes than directly solved trajectories, indicating that TRAJFIX recovers harder tasks. On SWE-Bench Pro, the model trained on the Solved set augmented with Recovered trajectories solves more tasks with larger patch sizes than the model trained on the Solved set alone. We further analysis that the improvement is mainly associated with better root-cause tracing, see Appendix G.4 for details.

5 RELATED WORK

Utilization of Failed Trajectories. According to the rejection sampling fine-tuning principle (Yuan et al., 2023), most failed trajectories are discarded, wasting the cost spent on them. In recent years, researchers have noticed the value of failed trajectories. They try to extract useful experience from a failed trajectory and use it to guide the next rollout. Such signals include failure causes (Lv et al., 2026), feedback from unit test execution (Da et al., 2025; Zhao et al., 2026a; Guo et al., 2026b), and other failure signals (Wang et al., 2026). The above methods perform guided resampling only once and do not effectively utilize the valid prefix of the failed trajectory. Researchers have also noticed the useful prefix steps of a failed trajectory (Zhu et al., 2026; Ding & Zhang, 2026). By replaying the prefix steps, they reduce the cost and accelerate inference, but subsequent rollout still relies on resampling to solve the task, providing limited improvement in the resolve rate. Other work uses failed trajectories to extract repair signals to improve the agent harness rather than directly repair the trajectories themselves (Chen et al., 2026). Researchers have also analysed the errors in failed trajectories (Zhao et al., 2026b), finding that a common failure mode is that the model locates the correct location but fails to make the correct edit (Majgaonkar et al., 2025; Kim et al., 2026).

Verifiable SWE data. SWE-Bench (Jimenez et al., 2024) established execution-based evaluation on real GitHub issues, later extended to multiple languages (Zan et al., 2026) and long-term tasks (Deng et al., 2025). Building on these benchmarks, a line of work focuses on producing verifiable training trajectories. Some researchers collect trajectories directly, where a model is rolled out on real tasks and only those that pass the tests are kept (Pan et al., 2025). Some researchers synthesise tasks in reverse, either by mutating code to create bugs (Pham et al., 2025; Yang et al., 2026) or by inferring a task from an existing test (Zhang et al., 2025). Other researchers produce trajectories by injecting and then handling failures. Self-play methods inject bugs into the code and then repair them, producing trajectories without human-written tasks (Wei et al., 2025). Similarly, tool-failure methods inject failures into tool interactions and train models to recover from them (Vuddanti et al., 2025). Building the runtime that executes and verifies these trajectories is itself automated by other work (Guo et al., 2025; Hu et al., 2026; Guo et al., 2026a).

6 CONCLUSION

In this paper, we introduced TRAJFIX, a multi-agent framework that repairs failed trajectories. For each failed trajectory, it replays the correct prefix, corrects the first wrong step, and then lets the model continue the rollout on its own. The hallucination detection mechanism rejects corrections that contain reasoning jumps, while multi-point repair re-diagnoses the best trajectory from the previous round. Under a matched rollout budget per task, TrajFix improves the resolve rate by 44.0 points over resampling while reducing the cost of producing a trajectory usable for training by 47.6%. More importantly, TRAJFIX can solve tasks that resampling cannot solve even with additional rollouts. We also compared six mainstream models as the diagnosis and correction model, where Claude-Opus-5 is the best overall. Finally, training on successful trajectories together with those recovered by TRAJFIX improves downstream SWE performance over training on successful trajectories alone, particularly in the model's ability to solve difficult tasks.

AI USE STATEMENT

In this work, we used large language models in a limited way. We used them to assist with translation. We also used them to edit the paper for readability and to improve its format, such as tables and references. This is the full extent of our use. The core ideas of this paper, the research methodology, and the design and execution of every experiment were all done by the authors. We did not use large language models to form our hypotheses, or to interpret our results. Every number and every claim in the paper comes from our own experiments. We have reviewed all AI-assisted text, and we take full responsibility for the final content of this work.

ETHICS STATEMENT

This work does not involve human subjects, private data, or any sensitive information. All the benchmarks and models we use are public, and we follow their licenses. Our method repairs failed software engineering trajectories and uses them as training data, which helps build more reliable coding models. We do not see a direct harm from this work, and we follow the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

The results in this paper can be reproduced because every component needed to rebuild TRAJFIX and rerun the experiments is either described in the paper or publicly available. The method and its multi-agent loop are described in Section 2, with the algorithm and per-agent input/output in Appendix A, and the diagnosis, correction, and hallucination detection prompts reproduced verbatim in Appendix B, so the agents can be rebuilt exactly. All benchmarks and models we use are public; the evaluation pools and the training sets, together with their statistics, are described in Section 3. The evaluation metrics, ablation hyperparameters, and training settings are stated in Section 3 and Appendices D.1 and G.2, and the cost accounting in Appendix C gives the token counts so that the reported costs can be recomputed. Every number in the paper comes from these experiments, and the appendix tables report the statistics behind each figure so they can be checked independently.

REFERENCES

Anthropic. Introducing Claude Opus 5. Anthropic Blog, `https://www.anthropic.com/news/claude-opus-5`, 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*, 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified. OpenAI Blog, `https://openai.com/index/introducing-swe-bench-verified/`, August 2024.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. Agent-rlvr: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425*, 2025.

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.

Yifeng Ding and Lingming Zhang. Swe-replay: Efficient test-time scaling for software engineering agents. *arXiv preprint arXiv:2601.22129*, 2026.

Chuanzhe Guo, Jingjing Wu, Sijun He, Yang Chen, Zhaoqi Kuang, Shilong Fan, Bingjin Chen, Siqi Bao, Jing Liu, Hua Wu, et al. Menvagent: Scalable polyglot environment construction for verifiable software engineering. *arXiv preprint arXiv:2601.22859*, 2026a.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*, 2025.

Yaoqi Guo, Yang Liu, Jie M Zhang, Yun Ma, Yiling Lou, and Zhenpeng Chen. Swe-doctor: Guiding software engineering agents with runtime diagnosis from multi-faceted bug reproduction tests. *arXiv preprint arXiv:2607.00990*, 2026b.

Ruida Hu, Chao Peng, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale. *Advances in Neural Information Processing Systems*, 38: 32679–32718, 2026.

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.

Myeongsoo Kim, Dingmin Wang, Siwei Cui, Farima Farmahinifarahani, Terry Yue Zhuo, Shweta Garg, Baishakhi Ray, Rajdeep Mukherjee, and Varun Kumar. Coherence collapse: Diagnosing why code agents fail after reaching the right code. *arXiv preprint arXiv:2603.24631*, 2026.

Dongyi Lv, Aichen Cai, Liang Huang, Ya Zhang, Qiuyu Ding, Canhui Wu, Zhi Wang, Yuesong Zhang, Jiaqi Wang, Nan Duan, et al. Failforge: Distilling procedural competence from persistent failures into code agents. *arXiv preprint arXiv:2608.08570*, 2026.

Murong Ma, Tianyu Chen, Yun Lin, Shuai Lu, Qinglin Zhu, Yeyun Gong, Zhiyong Huang, Peng Cheng, Yan Lu, and Jin Song Dong. From patches to trajectories: Privileged process supervision for software-engineering agents. *arXiv preprint arXiv:2605.21996*, 2026.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories. *arXiv preprint arXiv:2511.00197*, 2025.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. OpenAI Blog, https://openai.com/index/gpt-5-6/, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Cq1BNvHx74.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Sri Vatsa Vuddanti, Aarav Shah, Satwik Kumar Chittiprolu, Tony Song, Sunishchal Dev, Kevin Zhu, and Maheep Chaudhary. Paladin: self-correcting language model agents to cure tool-failure cases. *arXiv preprint arXiv:2509.25238*, 2025.

Chenyu Wang, Yunbo Lyu, Junda He, Zhou Yang, Chenxing Zhong, Yaniv Harel, and David Lo. Fail-fast, restart-smart: Early failure prediction and restart for swe agentic tasks. *arXiv preprint arXiv:2608.03222*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play swe-rl. *arXiv preprint arXiv:2512.18552*, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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://openreview.net/forum?id=mXpq6ut8J3.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. SWE-flow: Synthesizing software engineering data in a test-driven manner. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=P9DQ2IExgS.

Chenyu Zhao, Shenglin Zhang, Yihang Lin, Wenwei Gu, Zhimin Chen, Yongqian Sun, Dan Pei, Chetan Bansal, Saravan Rajmohan, and Minghua Ma. Debugging the debuggers: Failure-anchored structured recovery for software engineering agents. *arXiv preprint arXiv:2605.08717*, 2026a.

Xiangxin Zhao, Han Li, Shuaiting Li, Tianyi Zhao, Earl T Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of cli coding agent trajectories. *arXiv preprint arXiv:2607.09510*, 2026b.

Tianshu Zhu, Wenyu Zhang, Xiaoying Zuo, Lun Tian, Haotian Zhao, Yucheng Zeng, Jingnan Gu, Daxiang Dong, Jianmin Wu, Dawei Yin, et al. Rollout pass-rate control: Steering binary-reward rl toward its most informative regime. *arXiv preprint arXiv:2605.05112*, 2026.

APPENDIX

A TRAJFIX IMPLEMENTATION DETAILS

A.1 FRAMEWORK EXECUTION DETAILS

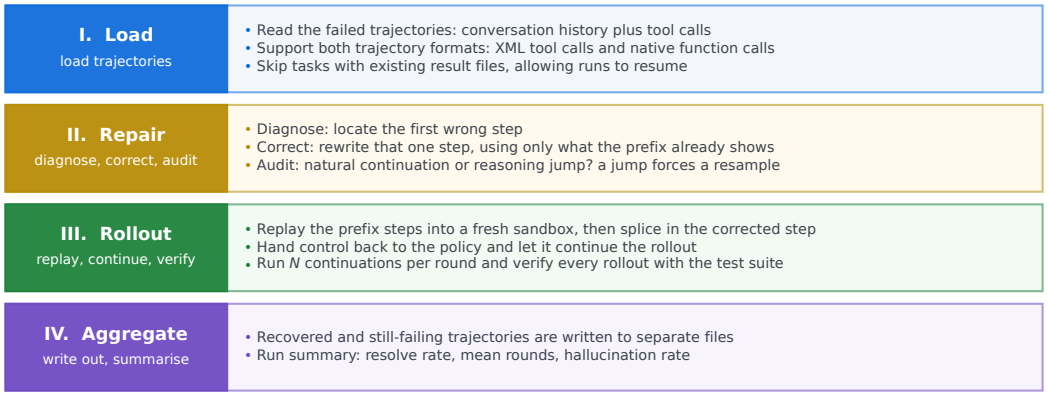

Figure 6: **The four stages of TRAJFIX.** The load stage reads failed trajectories and skips tasks with existing results to support resumable runs. The repair stage diagnoses and corrects the first wrong step and audits the correction for information leakage or reasoning jumps. The rollout stage replays the prefix in a fresh container, executes the corrected step, and lets the model continue until submission. The aggregate stage separates recovered and still-failing trajectories and summarizes the resulting metrics.

Load stage. Figure 6 lists what every stage reads and writes. Every task that finishes normally writes its own result file. A restarted run skips tasks that have already been completed. With a job that spans tens of thousands of tasks, this resume mechanism improves the robustness of the framework. The action format of a trajectory varies, so we support both the XML tool-call format and the JSON tool-call format. This allows us to process a wider range of failed trajectories.

Repair stage. A corrected step goes through the hallucination audit and a tool-call format check. A step with an invalid tool-call format is immediately regenerated, which avoids the cost of the hallucination audit.

Rollout stage. Replay is possible because we can send a chosen tool call to a fresh container. We first extract the actions and observations from the failed trajectory. We re-execute the same actions in the fresh container, including file writes and bash commands. Once the replay is complete and the corrected step has been executed, all previous messages are added to the conversation history, and the rollout continues from there.

Aggregate stage. A recovered trajectory retains the completions format and can therefore be directly converted into training data. We also report related metrics, such as the mean number of steps and the mean number of lines in the patch. The mean number of patch lines provides a useful prior for task difficulty, as tasks with more modified lines are generally harder to repair.

A.2 THE TRAJFIX ALGORITHM

Algorithm 1 states the control flow of Figure 2 in one place.

A.3 INPUT AND OUTPUT OF EACH AGENT

Table 5 lists the inputs and outputs of each agent. The diagnosis agent is the only agent with access to oracle information. The hallucination detection agent sees only the prefix steps and the corrected step, so its verdict is independent of the oracle information.

Algorithm 1 TRAJFIX trajectory repair

Input: task x, failed trajectory $\tau^{(0)}$, golden patch $p^\star$, rounds M, rollouts per round N, audit retries J

Output: a verified trajectory, or the shortest failed trajectory from the previous round and its diagnosis

$\tau \leftarrow \tau^{(0)}$
for $m \leftarrow 1$ **to** M **do**
  $(t^*, e) \leftarrow \mathcal{D}(x, \tau, p^\star)$ *// earliest deviating step and error kind*
  for $j \leftarrow 1$ **to** J **do**
    $\tilde{a} \leftarrow \mathcal{C}(x, \tau_{<t^*}, a_{t^*}, e)$ *// minimal, prefix-derivable rewrite*
    if $\mathcal{A}(\tau_{<t^*}, a_{t^*}, \tilde{a}) =$ NATURAL **then break**
  end for
  $\mathcal{L} \leftarrow \emptyset$
  for $k \leftarrow 1$ **to** N **do**
    $S \leftarrow$ REPLAYPREFIX$(x, \tau_{<t^*})$ *// fresh container, prefix state restored*
    $\tau' \leftarrow \tau_{<t^*} \oplus \tilde{a} \oplus$ ROLLOUT(π, S) *// agent continues unsupervised*
    if $\mathcal{V}(x, \tau') = 1$ **then**
      return τ' *// recovered; stop early*
    end if
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{\tau'\}$
  end for
  $\tau \leftarrow \arg\max_{\tau' \in \mathcal{L}} (\mathcal{V}(x, \tau'), -|\tau'|)$ *// round-best becomes the next round's subject*
end for
return τ and its diagnosis

Table 5: Inputs and outputs of the agents in TRAJFIX.

Agent	Inputs	Outputs
Diagnosis Agent	**1. Failed trajectory (τ):** the thought, the action and the observation of every step. **2. Golden patch ($p^\star$):** the files it touches and the diff. **3. Test patch:** the F2P tests. **4. Previous diagnosis:** the shortest failed trajectory from the previous round and its diagnosis.	**1. Index of wrong step (t^*):** the index of the first wrong step. **2. Error type and reason:** grounded in the prefix, and never quoting the patch. **3. Suggested action:** a forward direction derived from the prefix.
Correction Agent	**1. Prefix steps ($\tau_{<t^*}$):** the conversation history up to the wrong step. **2. Wrong step (a_{t^*})** **3. Diagnosis:** error reason and suggested action. **4. Trajectory format:** XML tool calls or native function calls.	**Corrected step ($\tilde{a}$):** one reasoning segment plus exactly one tool call, derivable from the prefix and holding the minimal change.
Hallucination Detection Agent	**1. Prefix steps ($\tau_{<t^*}$)** **2. Corrected step ($\tilde{a}$)**	**1. Verdict:** NATURAL or JUMP. **2. Reason:** The claim that is not supported by the prefix steps. A JUMP sends the step back for generation again.
Rollout Agent	**1. Verified container:** the verified container of the task. **2. Prefix and corrected step:** the tool calls to replay, then the step to execute.	**1. Continued trajectory (τ'):** prefix, corrected step and the steps the policy adds on its own.
Verification Agent	**1. Fresh container:** A new container. **2. Test patch ($\mathcal{T}$):** the F2P tests. **3. Output patch:** the git diff of all turns.	**1. Result:** F2P passed or failed, which decides whether the trajectory is recovered.

A.4 Problem Formulation

SWE tasks and rollouts. A task instance is a triple $x = (R, q, \mathcal{T})$: a repository snapshot R at the parent commit of a merged pull request, the issue text q, and the test suite $\mathcal{T}$ that pull request added or changed. Each instance comes with an executable environment, built here by an automated multi-language constructor, and satisfies fail-to-pass: $\mathcal{T}$ fails on R and passes once the golden patch $p^\star$ is applied. A verifier $\mathcal{V}(x, \tau) \in \{0, 1\}$ runs the suite against the working tree a trajectory τ leaves behind, returning 1 only if the target tests flip and nothing that passed before breaks. A policy π — an LLM in an agent scaffold — produces $\tau = (a_1, \ldots, a_T)$, each step a_t bundling reasoning with one tool call and its observation; $\tau_{<t}$ denotes the prefix. Collection draws K independent rollouts per instance and keeps only those the verifier accepts:

$$\mathcal{D}_{\text{keep}} \;=\; \big\{(x, \tau_k) \;:\; \tau_k \sim \pi(\cdot \mid x),\; \mathcal{V}(x, \tau_k) = 1,\; k \le K\big\}, \qquad \text{Pass@}K(x) \;=\; \max_{k \le K}\; \mathcal{V}(x, \tau_k), \tag{2}$$

Trajectory repair. Collection leaves a pool of failed trajectories $\mathcal{D}_{\text{fail}} = \{(x_i, \tau_i) : \mathcal{V}(x_i, \tau_i) = 0\}$. Repair failed trajectory: Pick an index t^*, replace a_{t^*} with a corrected step $\tilde{a}$, and let the policy continue the rollout:

$$\underbrace{a_1,\; \ldots,\; a_{t^*-1}}_{\text{kept unchanged}},\qquad \underbrace{\tilde{a}}_{\text{corrected step}}\qquad ,\qquad \underbrace{a'_{t^*+1},\; \ldots,\; a'_{T'}}_{\text{policy continued the rollout}} \tag{3}$$

Call the result τ'. Repair succeeds on (x, τ) if any of K attempts passes the verifier.

B Prompts

This appendix gives the prompts of TrajFix. The diagnosis agent reads the shortest failed trajectory from the previous round instead of the original failed trajectory. The tool schema in the correction prompt is the OpenHands scaffold, which offers three tools: **str_replace_editor**, **execute_bash** and **finish**.

B.1 Diagnosis prompt

The diagnosis agent reads the issue, the files touched by the golden patch, the golden patch, the test patch, the latest rollout trajectory, and the diagnosis from the previous round. The prompt rules out a verdict of "no error found" because a trajectory reaches this stage only after failing the F2P test. The suggested action must be a forward step that follows from what the trajectory has already established.

Diagnosis prompt

```
You are an expert Software Engineering analyst. Your task is to analyze an AI agent's
debugging trajectory and identify where it went wrong.

**IMPORTANT PREMISE -- READ BEFORE YOU ANALYZE**:
- The trajectory below **DID NOT fully solve the issue**. It is provided precisely
because it contains at least one further error, deviation, or missed step that must be
identified and corrected.
- You **MUST** locate a concrete error step whose correction would meaningfully improve
the outcome. Do NOT conclude "no error", "trajectory appears correct", "already fixed",
"nothing further to do", or any equivalent -- such conclusions are considered invalid
output for this task.
- The error you identify MUST be a real actionable mistake in the trajectory (wrong
file, wrong reproduction, wrong fix, incomplete verification, etc.), not a trivial
cosmetic remark.

## Problem Statement (Issue to Solve)

{problem_statement}

## Related Files That Need Modification

The following files are involved in fixing this issue:

{related_files}

## Ground Truth Fix Summary (for reference only)
```

```
The correct fix involves changes to these files and functions. This information is
provided to help you evaluate whether the agent was on the right track, NOT as a
specification to describe in detail.

```diff
{fix_patch}
```

Unit Test Patch

Below is the test patch (git diff) used to verify the fix. Understanding the test
expectations helps identify whether the agent's approach was on the right track:

```diff
{test_patch}
```

Agent's Trajectory (Latest Rollout)

Below is the trajectory from the MOST RECENT rollout attempt. Unlike the initial
trajectory, this may already contain corrected steps from previous iterations. Analyze
it carefully.

{trajectory}

Previous Iteration Context

{previous_analysis_section}

Analysis Task
Please carefully analyze the agent's trajectory against the ground truth fix and test
expectations, then provide:

1. **Error Analysis**: What went wrong in the agent's approach? Identify the root
cause(s) of failure. Consider four kinds of problems:
   - Did the agent "locate and review the wrong files"?
   - Did the agent "create a wrong reproduction_issue script"?
   - Did the agent "make a wrong fix patch"?
   - Did the agent "fail to fully verify the fix patch"?

   The `error_analysis` field MUST be a non-empty string that names the specific
   mistake. Empty strings, "N/A", "none", or statements that the trajectory is correct
   are NOT acceptable.

2. **First Error Step**: Identify the **earliest step number** where the agent first
made a mistake or deviated from the correct solution path. Explain why this step is the
first point of failure.{previous_analysis_note}

   The `error_step` field MUST be an integer **>= 1** (steps are 1-indexed). Do NOT use
   0, -1, or any placeholder value -- you must commit to a specific real step in the
   trajectory.

3. **Correction Suggestions**: For the identified error step and subsequent steps, what
should the agent have done differently?
   - **CRITICAL RULE**: Your `suggested_action` must ONLY reference what the agent
   already knew, observed, or discovered in the trajectory up to that point.
   - **CRITICAL RULE**: Do NOT mention what the agent "should have done instead of B".
   Do NOT compare with the wrong action. Only state what the agent should do based on
   prior steps.
   - Structure your suggestion like this: "Based on the prior steps, the agent had
   already discovered X in step_N. Given X, it should now do A."
   - The suggested action must be a POSITIVE, forward-looking direction that follows
   logically from the prior trajectory alone, NOT from comparing the trajectory with
   the ground truth patch.

Output Contract (HARD REQUIREMENTS)
- `error_analysis`: non-empty string describing a concrete mistake.
- `error_step`: integer >= 1 pointing to a real step in the trajectory.
- `error_reason`: non-empty string explaining why that step is the first point of
failure.
- `suggested_action`: non-empty forward-looking direction grounded in prior steps.

Any output that violates these requirements (empty fields, `error_step` <= 0, or a "no
error" verdict) will be rejected.

Please respond in the following JSON format:

```json
{{
```
```

```
    "error_analysis": '<analysis of what went wrong>(e.g., "locate and review the wrong
    files")',
    "error_step": <step number (1-indexed), must be >= 1>,
    "error_reason": "<explanation of why this step contains the error>",
    "suggested_action": "<Based on the prior steps, the agent had already discovered X.
    Given X, it should now do A.>",
}}
```

B.2 CORRECTION PROMPT

The correction agent receives the conversation history of the latest rollout, the diagnosis, and the index of the first wrong step. It returns a single step consisting of a reasoning segment and exactly one tool call. The prompt imposes the admissibility constraint as the first hard constraint and the minimality constraint as the second. The suggested action from the diagnosis agent is treated as a soft reference and must be rejected if it requires a large change.

Correction prompt

````
You are an AI assistant that helps generate corrected agent steps based on historical
context and expert suggestions.

## Task

Given:
1. Historical conversation messages (OpenAI message format) from the latest rollout
2. An error analysis identifying where the latest rollout went wrong
3. Specific suggestions for what should have been done instead

Generate a corrected agent step that includes:
1. **Reasoning Content** (natural language reasoning before function calls)
2. **Tool calls** - a single function call formatted in the agent's format

## Input Format

You will receive:

### Historical Messages
```json
{
  "messages": [
    {"role": "user", "content": "..."},
    {"role": "assistant", "content": "...", ...},
  ]
}
```

### Error Analysis
```json
{
  "error_step": <step number where error occurred>,
  "error_reason": "<why this step was wrong>",
  "suggested_action": "<Based on the prior steps, the agent had already discovered X.
  Given X, it should now do A.>"
}
```

### Target Step
- The step number that needs to be corrected

## Output Requirements

### 1. Reasoning Content Guidelines

**CRITICAL -- How to structure your reasoning:**

Your reasoning MUST follow this structure:
1. First, reference WHAT you know from prior steps: "In step_N, I
observed/discovered/saw that X. The results from step_M confirmed Y."
2. Then, state WHAT you need to do based on that: "Given X and Y, I need to do A."

Your reasoning MUST NOT:
- Introduce new facts, new files, or new directions that are not grounded in prior
steps.
- Reference the golden patch, the error analysis's comparison with the "correct"
answer, or any external knowledge.
````

```
- Say things like "The error analysis suggests I should..." -- you must derive the
action from the prior context, not from the error analysis report.

**Length**: Follow the historical pattern -- match the typical reasoning length of the
prior steps.

**Style**: Match the tone, voice, and structure of the original agent's reasoning
blocks (first person vs neutral, "Let me..." vs "I should...", level of detail).

### 2. Tool Call Generation

Generate appropriate tool calls based on:
- The tools used in historical context (str_replace_editor, execute_bash, finish)
- Proper parameter formatting matching historical patterns
- **CRITICAL**: Only ONE function call per step

**Available Scale-SWE Tools**:
- str_replace_editor: commands=view/create/str_replace/insert,
parameters=command/path/file_text/old_str/new_str/insert_line/view_range
- execute_bash: parameters=command
- finish: no parameters

### 3. Integration Requirements (Priority Order)

**HARD CONSTRAINT #1: Reasoning must be derived from prior steps only**
- The corrected step's reasoning MUST start by citing specific prior steps and what
they revealed.
- The corrected step's final decision ("I need to do A") MUST follow logically from
those cited prior observations.
- If the corrected step's direction cannot be traced back to a prior step's
observation, it is INCORRECT.

**HARD CONSTRAINT #2: Minimum necessary change -- NO over-engineering**
- Make the SMALLEST change needed to correct the specific error identified.
- Do NOT restructure or refactor an entire function. Only change the specific lines or
logic that caused the error.
- Do NOT add new functions, new files, or new helper methods unless the issue
explicitly asks for a new feature.
- If the original trajectory was working on the right file but made a wrong edit, keep
the same file and fix only that edit.

**SOFT CONSTRAINT #3: Consider the suggested action (with caution)**
- The `suggested_action` from error analysis is a REFERENCE, not a command.
- Before adopting it, verify that every action in `suggested_action` is:
  (a) Grounded in prior steps, AND
  (b) The minimum change needed
- If `suggested_action` proposes a large refactor or new files, REJECT that part and
derive a smaller step instead.

## Output Format

```json
{
  "step_number": <the corrected step number>,
  "reasoning": "<the reasoning content -- MUST reference prior steps and derive action
  from them>",
  "tool_call": "...",
  "rationale": "<brief explanation of why this correction addresses the error>"
}
```

Key Considerations

1. **Context Derivation**: The reasoning MUST show that the corrected action follows
from prior observations. Every "I should" must be preceded by an "I
observed/saw/discovered in step_N".

2. **No External Knowledge**: Do not reference the error analysis as the source of the
fix direction. The error analysis is a helper, not a source of truth. The source of
truth is the prior trajectory.

3. **Minimal Change**: If you find yourself wanting to restructure a function or create
a new file, stop. Ask: "Can I fix this with a 1-5 line change in the existing code?" If
yes, do that instead.

4. **Natural Transitions**: The corrected step should read as if the agent itself,
having seen the prior steps, naturally arrived at the corrected approach. It should NOT
read as if a teacher is telling the agent what to do.

5. **Tool Selection**: Use the same tool the agent would have used. Match parameter
style.
```

```
---

Now, please generate the corrected step based on the provided historical messages and
error analysis.
```

B.3 HALLUCINATION DETECTION PROMPT

The hallucination detection agent receives only the prefix and the corrected step and is not provided with the golden patch. It returns one of two verdicts. A corrected step judged REASONING JUMP is generated again.

Hallucination detection prompt — system message

```
You are an expert AI reasoning evaluator specializing in analyzing software engineering
agent trajectories. Your task is to determine whether a given corrected reasoning step
follows naturally and coherently from the prior reasoning history, or whether it
represents a "reasoning jump" -- introducing fundamentally new directions, conclusions,
or information that contradict or significantly deviate from the prior context.

Definition of a **Reasoning Jump**:
- The corrected step introduces a completely new problem diagnosis or root cause that
was never hinted at or explored in prior steps.
- The corrected step makes a fundamental change of direction (e.g., targeting a
different file, function, or subsystem) that is not motivated by prior steps.
- The corrected step references observations, findings, or external knowledge that do
not appear in the prior trajectory and are not reasonably inferable from it.
- There is a large logical gap between what was established in prior steps and what the
corrected step concludes or decides -- not merely a gap in low-level implementation
specifics.

Definition of a **Natural Continuation** (default assumption):
- The corrected step directly builds on the diagnosis, observations, code readings,
test results, or analysis established in prior steps.
- The overall direction and goal of the corrected step is clearly motivated by prior
steps, even if specific implementation details were not explicitly pre-justified.
- Implementation choices that are standard engineering practice once a direction is
established count as Natural Continuation, not Reasoning Jump.
- Early exploratory steps with few or no prior steps are Natural Continuation.
- Even if the corrected step reaches a better solution than the original step, its
overall reasoning direction is supportable from prior context.

Key principle: Judge whether the **direction and high-level decision** is grounded in
prior steps. Do NOT penalize a step merely because specific low-level implementation
details were not explicitly pre-justified. Reserve "Reasoning Jump" for cases where the
step's direction or core conclusion contradicts or substantially departs from the
established prior trajectory.

Be precise in your analysis. Point to specific prior steps as evidence for your
judgment.
```

Hallucination detection prompt — user message

```
You are analyzing a software engineering agent trajectory. The agent has been working
step-by-step to solve a programming issue.

Below are the first {n_prior} steps of the original trajectory (displayed as Step 1
through Step {n_prior}):

=== PRIOR TRAJECTORY (Steps 1 - {n_prior}) ===
{prior_steps_text}
=== END OF PRIOR TRAJECTORY ===

A teacher model has produced a corrected version of Step {step_number} (which comes
immediately after the prior trajectory):

=== CORRECTED STEP {step_number} ===
{corrected_step_text}
=== END OF CORRECTED STEP ===

Your task: evaluate whether the corrected Step {step_number} follows **naturally and
coherently** from the prior {n_prior} steps, or whether it represents a **reasoning
jump** that introduces a fundamentally new direction, diagnosis, or conclusion that is
not grounded in the prior trajectory.
```

```
Provide your response using EXACTLY the following format (do not omit or rename these
tags):

<REASONING>
Provide a step-by-step analysis...
</REASONING>

<VERDICT>
Natural Continuation
</VERDICT>

or

<VERDICT>
Reasoning Jump
</VERDICT>

Only one of the two verdicts above is allowed inside <VERDICT>.
```

C Cost Accounting Details

This appendix extends the comparison of TrajFix and resampling in Section 3.2 by reporting additional detailed metrics. Specifically, we report the resolve rate by round and subset (Table 6).

Table 6: **Resolve rate by round and subset (%).**

Method	Full pool				Never solved				Solved before			
	Round 1	Round 2	Round 3	Round 4	Round 1	Round 2	Round 3	Round 4	Round 1	Round 2	Round 3	Round 4
Resampling (matched-k rounds)	30.0	33.0	34.0	35.0	8.0	10.0	10.0	10.0	52.0	56.0	58.0	60.0
Resampling (early stop)	30.0	41.0	43.5	47.5	8.0	12.0	13.0	14.0	52.0	70.0	74.0	81.0
Resampling (4 rounds)	30.0	41.0	43.5	47.5	8.0	12.0	13.0	14.0	52.0	70.0	74.0	81.0
TrajFix	**51.5**	**67.0**	**76.0**	**79.0**	**38.0**	**53.0**	**64.0**	**69.0**	**65.0**	**81.0**	**88.0**	**89.0**

Resolve rate. Table 6 shows that the difference in the overall resolve rate between TrajFix and resampling is mainly attributable to tasks that resampling rarely solves. Under a matched rollout budget per task, TrajFix improves the resolve rate by 44.0 points over resampling. Even when resampling is allowed to continue for four rounds, its resolve rate reaches only 47.5%, still well below the 79.0% achieved by TrajFix. In the never-solved subset, TrajFix resolves 69.0% of tasks, compared to at most 14.0% for resampling. In the solved-before subset, the resolve rates are closer, at 89.0% for TrajFix and 81.0% for resampling (early stop). This indicates that the gains from TrajFix are concentrated on tasks that the model does not solve in its initial attempts, where repairing failed trajectories is more effective.

Cost. Table 1 shows that TrajFix has the lowest cost per successful trajectory among the compared methods. Each successful trajectory costs $\$0.7390$ with TrajFix, compared with $\$1.4097$ for Resampling (matched-k rounds) and $\$1.5524$ for Resampling (early stop). TrajFix uses more output tokens per task than all resampling settings, but it uses fewer input tokens per task because it replays the existing prefix rather than deriving it from scratch. Even when resampling is extended to four rounds, its cost per successful trajectory remains higher at $\$0.9541$. All costs are calculated using peak-period prices and assuming that caching is enabled.

D Mechanism Ablation Details

D.1 Ablation setup

The ablation uses the same 400 failed trajectories of DeepSeek-V4-Flash-0731 as Section 3.2, split evenly into never-solved and solved-before tasks. Every variant runs on the same pool, so the only thing that changes is the mechanism under test. We ablate three components: the admissibility constraint (AC), which forces every corrected step to be derivable from the prefix; hallucination detection (HD), which rejects corrected steps that contain reasoning jumps; and multi-point progressive repair (MPPR), which re-diagnoses the shortest failed trajectory from the previous round so later rounds repair deeper wrong steps. Table 7 lists the inference settings shared by every agent, which stay fixed across all experiments.

Table 7: **Inference settings of every agent.**

Agent	Setting	Value
Diagnosis agent	Temperature	1.0
	Top_p	0.95
	Max output tokens	32,768
	Request retries	10
Correction agent	Temperature	1.0
	Top_p	0.95
	Max output tokens	32,768
	Request retries	10
Hallucination detection agent	Model	DeepSeek-V4-Pro-0813
	Regeneration attempts on JUMP	10
Rollout agent	Temperature	1.0
	Top_p	0.95
	Total step budget (prefix + continuation)	300
	Timeout per action	300 s
	Sandbox session limit	3,600 s
Infrastructure	Scaffold	OpenHands
	Sandbox resources	2 CPU / 8 GiB

Note that a case may end with an LLM Query Error, caused by upstream saturation or other factors we do not control. We rerun such a case. But when a model cannot produce the correct format and exhausts the maximum number of retries, the cause is the model itself, so we do not rerun the case.

D.2 ABLATION RESULTS

Figure 7 plots the ablation in the four cumulative variants, built up from a base that does not have a mechanism to the full TRAJFIX. Table 2 in Section 3.3.2 reports the same run.

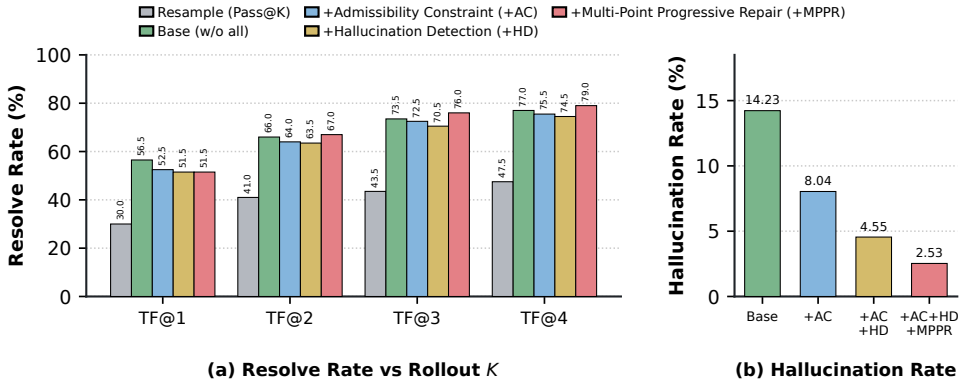

Figure 7: **Mechanism ablation over the four cumulative variants.**

The reliability mechanisms substantially reduce oracle-information leakage in corrected steps. The hallucination rate falls from 14.23% in the base variant to 2.53% in the full pipeline, indicating that the admissibility constraint and hallucination detection help prevent reasoning jumps. Although the base variant achieves a higher resolve rate at the smallest budget (56.5% at TF@1), this rate is higher than it should be because some corrections rely on oracle information and therefore do not provide reliable training data. Multi-point progressive repair raises the resolve rate on the never-solved subset from around 60% to 69.0% by re-diagnosing the shortest failed trajectory from the previous round.

D.3 HALLUCINATION ANALYSIS

The correction agent has access to the golden patch, so it may produce a step that cannot be derived from the prefix alone. Although such a step may pass the F2P tests, it introduces oracle information that is unavailable at inference time and can therefore make the training data less reliable. Hallucination detection is therefore essential, as the ablation in Table 2 shows that removing it raises the leakage rate to 14.23%. A representative example is `Shopify__shopify_app-1743`. The diagnosis suggested replacing a direct call to **ShopifyAPI::Webhooks::Registry.register_all** in **recreate_webhooks!** with the existing **create_webhooks(session:)** wrapper. However, the prefix only showed the direct call and provided no evidence of using the wrapper. The correction therefore followed the golden patch rather than the observations in the prefix. The detector flags it as a reasoning jump and the step is regenerated.

A corrected step judged a reasoning jump

```
instance_id: Shopify__shopify_app-1743

"rationale": "The current step calls ShopifyAPI::Webhooks::Registry.register_all
directly in recreate_webhooks!, but the golden patch reuses the existing
create_webhooks(session:) wrapper, so the current approach is wrong and diverges from
the intended code path. I need to correct it to match the golden patch and route
webhook creation through create_webhooks(session:)."
```

E ERROR DIAGNOSIS AND CORRECTION EXPERIMENT DETAILS

E.1 MODELS DESCRIPTION

Table 8 lists the six models. Each one fills the diagnosis role and the correction role at the same time, and every other role is fixed. Every model is served through the same gateway with the same request settings.

Table 8: **The six models compared as the diagnosis and correction model.**

Model	Vendor	Release
Claude-Opus-5	Anthropic	Closed
Kimi-K3	Moonshot AI	Open-Weight
DeepSeek-V4-Flash-0731	DeepSeek	Open-Weight
DeepSeek-V4-Pro-0813	DeepSeek	Open-Weight
GPT-5.6-Sol	OpenAI	Closed
Qwen3.8-Max	Alibaba	Closed

E.2 DATASET

To better evaluate the repair capabilities of the model, we constructed a more challenging data set of 400 failed trajectories generated by DeepSeek-V4-Flash-0731. Half of the trajectories come from tasks that pass exactly once in four attempts ($\mathrm{Pass@4} = 0.25$), while the other half come from tasks that fail in all four attempts ($\mathrm{Pass@4} = 0$). We exclude C and C++ due to their limited sample counts and distribute the trajectories evenly across the remaining eight programming languages: TypeScript, JavaScript, Python, Go, Rust, PHP, Ruby and Java, with 25 trajectories per language in each subset.

E.3 INFERENCE SETTINGS

All agents use the settings of Table 7. Only two things change from the ablation of Appendix D. The search budget is $M{=}3$ diagnosis rounds with $N{=}1$ continuation per round. And the roles are assigned as follows: the model of the current row does diagnosis and correction, DeepSeek-V4-Flash-0731 always continues the rollout, and DeepSeek-V4-Pro-0813 always audits the corrected step. A failed trajectory counts as resolved when some continuation passes the F2P test.

E.4 RESULTS

Table 9 gives the resolve rate of each model in each of the eight languages. The language order is stable across models. TypeScript is the easiest and Ruby is the hardest. Kimi-K3 leads on five languages and Claude-Opus-5 on three. Claude-Opus-5 has the highest mean resolve rate (66.0%), and Kimi-K3 follows close behind.

Table 9: **Resolve rate by language** (%). Each cell divides by the fixed 50 trajectories of that language. The last row is the mean over the eight languages.

Language	Claude-Opus-5	Kimi-K3	DS-V4-Flash-0731	DS-V4-Pro-0813	GPT-5.6-Sol	Qwen3.8-Max
TypeScript	74.00	**80.00**	68.00	72.00	66.00	52.00
JavaScript	**74.00**	64.00	**74.00**	56.00	58.00	40.00
Python	70.00	**70.00**	66.00	62.00	52.00	28.00
Go	64.00	**66.00**	54.00	58.00	58.00	32.00
Rust	**64.00**	**64.00**	**64.00**	56.00	62.00	42.00
PHP	62.00	**66.00**	56.00	60.00	50.00	46.00
Ruby	58.00	56.00	46.00	**60.00**	42.00	36.00
Java	**62.00**	56.00	58.00	54.00	50.00	42.00
Mean	**66.00**	65.25	60.75	59.75	54.75	39.75

Table 10 splits the pool by whether the policy had ever solved the task. Every model is much stronger on the solved-before half. Claude-Opus-5 is the best in the solved-before half, so it is the better choice when the goal is to make unstable tasks pass reliably. Kimi-K3 is the best on the never-solved half, so it is the better choice when the goal is new coverage.

Table 10: **Resolve rate by rounds and subsets** (%).

Repair model	Whole			Solved before			Never solved		
	TF@1	TF@2	TF@3	TF@1	TF@2	TF@3	TF@1	TF@2	TF@3
Claude-Opus-5	**41.75**	**56.25**	**66.00**	58.00	**71.50**	**81.50**	**25.50**	41.00	50.50
Kimi-K3	41.25	55.75	65.25	**58.50**	70.00	77.00	24.00	**41.50**	**53.50**
DeepSeek-V4-Flash-0731	38.25	51.50	60.75	53.50	69.00	76.00	23.00	34.00	45.50
DeepSeek-V4-Pro-0813	39.75	50.75	59.75	58.50	69.00	77.00	21.00	32.50	42.50
GPT-5.6-Sol	36.00	48.25	54.75	55.50	67.00	71.50	16.50	29.50	38.00
Qwen3.8-Max	26.50	34.50	39.75	40.50	51.50	57.50	12.50	17.50	22.00

F ERROR TYPES

Table 11: Error types over 14,544 failed trajectories with consistent diagnoses. *Repaired* counts the trajectories whose case ends with a passing F2P test.

First-error type	Count	Proportion	Repaired	Recovery Rate
Right place, wrong fix	8,108	55.7%	3,284	40.5%
Fix too narrow	3,032	20.8%	1,249	41.2%
Gave up a fix that was correct	1,354	9.3%	360	26.6%
Faked or skipped the real test	1,236	8.5%	470	38.0%
Branch or edge case missed	465	3.2%	208	**44.7%**
Wrong root cause	349	2.4%	131	37.5%
Total	**14,544**	**100%**	**5,702**	**39.2%**

Section 4.2 reports the proportion and the recovery rate of each type of error (Figure 4). The pool is the same as in Section 4.2: the 14,544 failed trajectories of DeepSeek-V4-Flash-0731 that receive a consistent error type from both DeepSeek-V4-Flash-0731 and Kimi-K3.

Right place, wrong fix (55.7%, repaired 40.5%). The agent identifies the correct file and function but makes an incorrect edit. Common errors include inverted conditions, mistyped paths, and incorrect

API usage. When the prefix has already located the right place, the repair only needs to correct the edit itself, making these local errors recoverable. However, errors that involve additional files cannot be fully addressed by changing a single step.

Right place, wrong fix

```
[RECOVERED]  gradio-app-gradio-pr-5877
  step 24   stores the per-cell style data under the metadata key
            `styles` instead of `styling`        <-- wrong step
  Analysis:
            the mistake is local to one edit, so one replacement step (correctly
            assembles the storage path) is the right size of intervention.

[UNRECOVERED]  antrea-io-antrea-pr-4119
  step 56   implements only creation-time validation and skips the
            sync-time validation and error-propagation refactor        <-- wrong step
  Analysis:
            the required change is a broad refactor across sync,
            span/status, and error propagation, well beyond a single
            corrected step. Both rounds fail.
```

Fix too narrow (20.8%, **repaired** 41.2%)**.** The agent identifies the root cause and proposes the correct plan, but the repair is incomplete. Additional changes may be needed in another file, at another call site, or in a generated file, but these are omitted. When only one or two nearby changes are missing, the corrected step can make the main change and the continuation can complete the remaining edits, allowing the task to be recovered. However, when the missing changes span several files, a single corrected step may not be sufficient to implement the full plan.

Fix too narrow

```
[RECOVERED]  offchainlabs-prysm-pr-6141
  step 27   rewrites `validateAggregateAttTime` but leaves its
            `uint64` `genesisTime` signature unchanged   <-- wrong step
  Analysis:
            only the parameter type and one caller need updating, so the
            corrected step makes this contained edit and the continuation
            completes the fix.

[UNRECOVERED]  apache-shardingsphere-elasticjob-pr-1760
  step 19   implements `DefaultFilterChain` with recursive Iterator
            auto-advance and no `finished` guard        <-- wrong step
  Analysis:
            the whole chain design is wrong, it needs finished state and step-by-step
            advancing, so a single replacement cannot rebuild it. Both rounds fail.
```

Gave up a fix that was correct (9.3%, **repaired** 26.6%)**.** The agent makes the correct change, either in the original trajectory or in an earlier repair round, but later deletes or reverts it. For example, misleading test output may cause the agent to discard its existing implementation and start over, undoing its previous progress. Even if a corrected step restores the intended change, the agent may later revert it again. This error type has the lowest recovery rate among the six categories.

Gave up a fix that was correct

```
[RECOVERED]  cebra-pr-122
  step 14   adds the correct `discrete` / `showlegend` interface with a
            ValueError guard
  step 36   deletes the validation                          <-- wrong step
  Analysis:
            the correct change is already in the prefix, so the corrected step only has
            to put it back, and nothing later deletes it again.

[UNRECOVERED]  inline-snapshot-pr-294
  step 6    correctly diagnoses a pairing problem
  step 7    abandons it for defensive coding                <-- wrong step
  Analysis:
            the agent wrote a reason for the deletion and that reason stays in the
            prefix. Restoring the action does not change the reason, so the agent
            abandons the fix again.
```

Faked or skipped the real test (8.5%, **repaired** 38.0%)**.** The agent validates its work using incorrect evidence. It may write its own script instead of running the provided tests, skip the testing

altogether, or misinterpret the test output. This error can have a broader impact than its immediate scope because subsequent steps rely on the incorrect signal. When the agent later receives the correct signal from the actual tests, it can identify and fix the problem, allowing the task to be recovered.

Faked or skipped the real test

```
[RECOVERED]  ag2-pr-1951
  step 11   adds `native_tool_calls: bool = False` to `OllamaLLMConfigEntry`,
            matching the golden patch
  step 31   reverts it after a faulty verification          <-- wrong step
  Analysis:
            the code is already right and only the signal is wrong, so a real test
            gives the correct feedback back and the agent fixes itself.

[UNRECOVERED]  aiida-core-pr-4961
  step 43   implements `_attach_outputs` on a wrong assumption about port
            traversal, and verifies against the same assumption  <-- wrong step
  Analysis:
            the fake test hides a wrong root cause instead of protecting a correct fix.
            Fixing the test only shows the deeper error.
```

Branch or edge case missed (3.2%, repaired 44.7%). The agent identifies the correct file and function and proposes the right plan, but the implementation misses a required branch, such as handling an empty value, a specific type, a terminal state, or a boundary condition. When the missing branch is self-contained, the corrected step can add it and recover the task.

Branch or edge case missed

```
[RECOVERED]  a2a-python-pr-279
  step 21   misses the branch that rejects `update_status` once a terminal
            state is reached                                <-- wrong step
  Analysis:
            the missing branch is one condition inside one function and no other code
            depends on it, so the corrected step adds it directly.

[UNRECOVERED]  django-migration-linter-pr-195
  step 25   misses a branch at the `sql_analyser` entry point   <-- wrong step
  Analysis:
            that branch also needs an interface change, so it does not stand on its
            own. The agent has not understood the entry point.
```

Wrong root cause (2.4%, repaired 37.5%). The agent misidentifies the root cause of the bug. A common example is treating a symptom as the cause, such as adding a guard for when the actual problem lies upstream. If the diagnosis identifies the true cause, the corrected step can redirect the agent, allowing the continuation to proceed along the right path. However, if the diagnosis only rules out the wrong approach without identifying the correct one or if finding the right approach requires a deeper understanding of the data flow, a single corrected step may not be enough.

Wrong root cause

```
[RECOVERED]  rust-lang-rust-analyzer-pr-18559
  step 6    attributes the crash to
            `layout.rs::struct_tail_erasing_lifetimes` and adds cycle
            detection there                          <-- wrong step
  Analysis:
            the diagnosis points to the wrong location, but the actual
            cause is in `unify.rs::is_sized`. Redirecting the agent to the
            correct function is enough for a localized fix.

[UNRECOVERED]  a2a-python-pr-472
  step 26   finds the correct files, including `default_request_handler.py`,
            but misunderstands the data flow                <-- wrong step
  Analysis:
            diagnosis rules out the wrong direction without giving the right one. What
            is missing is a new model of the data flow, and one action cannot give the
            agent a model.
```

Two limitations underlie the unrepaired cases described above. The first is the edit budget: a corrected step can perform only one action, which may be insufficient when three or more changes must be made together. The second is the cognitive direction: a corrected step can change the agent's next action, but not necessarily its understanding of the code. If the agent retains an incorrect mental model, it may repeat the original mistake even after the action is corrected.

G SWE End-to-End Ability Experiment Details

G.1 Data collection

Task pool. The pool holds 55,366 tasks from the 5,101 repositories. Every task comes from a merged pull request, and it carries the issue text, the golden patch, the test patch, and a container image that can run the tests. The pool covers ten languages. Figure 8 gives the composition. Python and Go account for two-thirds of the tasks, while C and C++ together account for just 0.5%. We keep this natural distribution. The baseline and every treatment read the same pool, so the comparison is not affected by the imbalance. The repositories in this pool do not overlap with those included in SWE-Bench Verified, SWE-Bench Multilingual or SWE-Bench Pro, preventing data contamination at the repository-level.

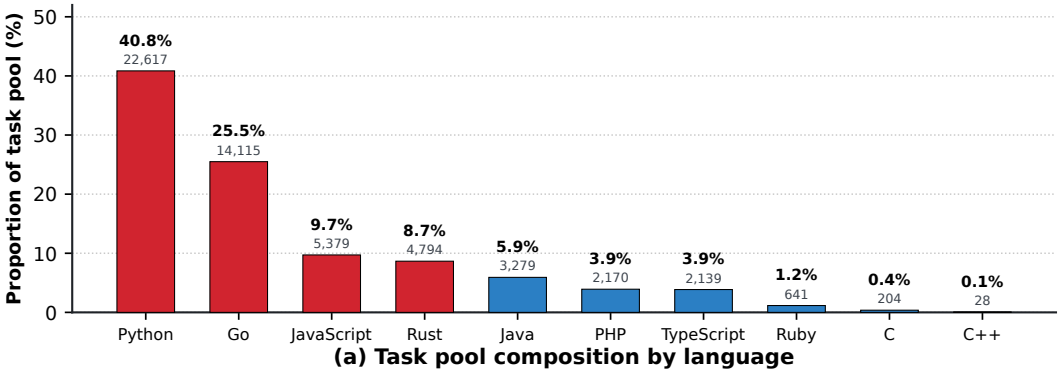

Figure 8: **Language composition of the task pool.** Bars give the proportion of the pool, and labels give the proportion and the task count. The four head languages (in red) hold 84.7% of the pool.

Collection. One collection round with DeepSeek-V4-Flash-0731 produces 37,472 trajectories, which we call the *solved* trajectories. We apply TrajFix to these failed trajectories twice, with two diagnosis rounds and one continuation rollout per round in each run. DeepSeek-V4-Flash-0731 is used for diagnosis, correction, and continuation rollout. The two runs produce 19,171 recovered trajectories, which form the **Recovered set**. Table 12 reports both rates per language.

Table 12: **Collection rate and the recovery rate by languages.** *Collection rate* is solved trajectories over tasks. Every task is attempted twice, so the denominator of the recovery rate is twice the number of failed trajectories.

Language	Tasks	Collection		Repair		
		solved	rate (%)	failed	recovered	rate (%)
Python	22,617	15,516	68.6	14,202	7,813	55.0
Go	14,115	8,625	61.1	10,980	5,865	53.4
JavaScript	5,379	3,783	70.3	3,192	1,710	53.6
Rust	4,794	3,477	72.5	2,634	1,391	52.8
Java	3,279	2,286	69.7	1,986	1,074	54.1
PHP	2,170	1,708	78.7	924	507	54.9
TypeScript	2,139	1,497	70.0	1,284	592	46.1
Ruby	641	416	64.9	450	171	38.0
C	204	151	74.0	106	43	40.6
C++	28	13	46.4	30	5	16.7
All	**55,366**	**37,472**	**67.7**	**35,788**	**19,171**	**53.6**

The mean collection rate in all languages is 67.7%. Apart from Ruby, C and C++, which have few failed trajectories, the other languages recover at close rates, between 46.1% and 55.0%.

G.2 Training and deployment settings

Table 13 lists the settings of each experiment. All three runs share them, so the only difference is the training data.

Table 13: **Training and deployment settings of the experiment.**

Stage	Setting	Value
Training	Trajectory format	function call (json)
	Epochs	5
	Global batch size	48
	Learning rate	5e-5
	Max Tokens	128k
	Warmup ratio	5%
Deployment	Scaffold	OpenHands
	Rollouts per task	4
	Max steps	300
	Max Tokens	128 k (Qwen3-8B), 256 k (Qwen3-30B-A3B)
	Temperature / top-p	1.0 / 0.95
	Request retries / timeout	5 / 1,800 s
	Sandbox resources	2 CPU / 8 GiB
Benchmarks	SWE-Bench Verified (Chowdhury et al., 2024)	500 Python tasks
	SWE-Bench Multilingual (Zan et al., 2026)	300 tasks, 9 languages
	SWE-Bench Pro (Deng et al., 2025)	731 long-horizon tasks

G.3 Experiment results

The setup is identical to Section 4.3: the same task pool of $55{,}366$ instances from $5{,}101$ repositories, from which DeepSeek-V4-Flash-0731 collects $37{,}472$ successful trajectories (the **Solved** set). Applying TrajFix to the failed trajectories twice, with two diagnosis rounds and one continuation rollout per round, yields $19{,}171$ **Recovered** trajectories; the corresponding 19,171 failed trajectories form the **Unrepaired** set, and four rounds of independent resampling on the same failed tasks produce $9{,}594$ successful trajectories, the **Resampled** set. We fine-tune Qwen3-8B and Qwen3-30B-A3B on *Solved* plus one of the three additional sets and evaluate every run with the OpenHands scaffold on SWE-Bench Verified, Multilingual and Pro. Table 4 reports both AVG@4 and Pass@4 for all arms.

Adding the trajectories recovered by TRAJFIX to the Solved set improves both AVG@4 and Pass@4 across all three benchmarks for both base models. The AVG@4 gains are larger for Qwen3-8B than for Qwen3-30B-A3B. This difference may reflect the higher baseline performance of the stronger model and the correspondingly smaller room for improvement.

The two control conditions help clarify the contribution of the recovered data. Adding unrepaired failed trajectories from the same tasks also improves performance over the Solved baseline, suggesting that these trajectories contain useful training signal. However, this condition consistently underperforms the recovered condition, indicating that repairing the trajectories contributes substantially to the gains beyond selecting these tasks alone. Adding resampled trajectories produces the smallest improvements, consistent with the limited ability of resampling to obtain the training signal from tasks it cannot solve.

G.4 Experiment analysis

Patch size analysis

Section 4.3 examines the patch size in both the training data and the evaluation results. In the training data, the trajectories recovered by TrajFix have larger patches than the tasks solved by the policy without repair. On SWE-Bench Pro, the model trained on the Solved set together with recovered data by TrajFix solves tasks with larger patches on average than the model trained on the Solved set alone. Since patch size serves as a proxy for task difficulty, these findings suggest that TrajFix recovers harder tasks and that recovered data helps extend the model's capabilities to more

challenging problems. Here, we extend the training-data analysis by reporting patch sizes broken down by programming language.

Table 14: Mean patch size (lines) of the training data by language.

Language	Solved Unaided	Recovered	Unsolved	Recovered−Solved Unaided
Go	25.03	41.34	79.33	+16.31
Python	24.62	39.68	68.04	+15.06
JavaScript	19.83	28.45	54.29	+8.62
TypeScript	26.83	34.87	62.17	+8.04
Java	41.75	63.03	85.66	+21.28
Rust	45.88	75.81	113.29	+29.93
PHP	17.06	30.44	59.43	+13.38
Ruby	20.33	26.34	39.75	+6.01
C	38.74	40.50	81.00	+1.76
C++	13.25	54.56	51.53	+41.31
Overall	**26.89**	**42.23**	**73.96**	**+15.34**

Recovered tasks have larger mean patch sizes than tasks solved unaided across all eight main languages. Overall, the mean patch size increases from 26.89 lines for tasks solved unaided to 42.23 lines for recovered tasks, while tasks that remain unsolved have the largest mean patch size at 73.96 lines. The C++ results do not follow the overall ordering but are based on a few deduplicated samples and are therefore unstable.

THE ABILITY IMPROVES: ROOT-CAUSE ANALYSIS

We compare the evaluation trajectories generated by the model trained on the Solved set alone (the baseline run) with those generated by the model trained on the Solved set augmented with recovered trajectories (the recovered run). In all cases, the difference lies consistently in the root-cause analysis. The baseline run works at the place where the symptom appears, while the recovered run traces the symptom back to the place where it is produced. We present two examples in detail below and write round i, step j as $r_i s_j$.

Case 1 — `jq`: edit the grammar, not its generated output

```
TASK      The lexer rejects `label` as a $label binding name. The fix has to change the
          grammar in src/parser.y and regenerate src/lexer.c with flex.

Baseline Run      4 rollouts, none solved
  r1      exits after 40 steps without ever finding parser.y
  r2-r4   read lexer.c and try to edit it directly
          -> lexer.c is a generated artifact; editing it is the wrong target

Recovered Run     solved in r3, 78 steps
  r3      thought:
          "Now let me look at the full parser.y to understand all the places that need
          to be changed. Let me also check if there are any other places where $ IDENT
          is used in patterns that I might have missed."
          -> treats parser.y as the source and lexer.c as its product, edits the grammar
          and regenerates
```

Case 2 — Terraform `removed` block: fix graph construction, not execution

```
TASK      A local-exec provisioner inside a `removed` block does not run when the
          resource lives in a module. AttachResourceConfigTransformer.Transform() skips
          a resource whose config is nil, while the `removed` block may sit in an outer
          module.

Baseline Run      4 rollouts, none solved
  r2-r4   every rollout edits node_resource_destroy.go, trying to restore the
          provisioner at execution time
          -> too late: the graph phase has already dropped it

Recovered Run     solved in r1, 69 steps
  r1      thought:
```

```
"The problem is in lines 52-55: when config is nil, the code just continues,
skipping the resource entirely. But the removed block might be at a higher
level (like the root module) that is still present in the config."
-> edits transform_attach_config_resource.go and attaches the block during
graph construction
```