ASSERT: SYNTHESIZING REPLAY-VERIFIED TRAJECTORIES WITH ACCESS ISOLATION FOR SOFTWARE ENGINEERING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reference patches specify verified fixes, but do not provide the reliable interaction traces needed to train software engineering agents. Rejection sampling discards failed attempts, while patch-conditioned traces can contain unsupported reasoning and redundant steps. Therefore, we introduce ASSERT, a framework that converts reference patches into executable training trajectories across programming languages. Its central design separates privileged plan recovery from the execution evidence used to justify supervised decisions. Dependency-Constrained Trajectory Repair (DCTR) repairs failed plans within dependency regions identified by failure feedback, while Replay-Certified Decision-Sufficient Slicing (RCDS) removes redundant interactions subject to evidence, replay, and final-test constraints. A patch-blind runner supplies observations, and an isolated rationale writer explains fixed actions using preceding evidence. With GPT-5.5 and GLM-5.2 teachers, ASSERT retains up to 2.93 times as many trajectories as the largest baseline pool and improves student Pass@1 by up to 5.84 percentage points over the strongest baseline in each setting on SWE-bench Verified and Multi-SWE-bench. To compare supervision quality at matched task coverage, we also train a student for each method using one method-specific trajectory per task in each setting's four-method training intersection. ASSERT maintains a Pass@1 advantage of up to 6.07 percentage points over the strongest baseline and has the lowest unsupported-claim rate in every setting, supporting improvements in both supervision yield and training utility.

1 INTRODUCTION

Large language model (LLM) agents address software issues by inspecting repositories, editing code, and validating proposed patches (Yang et al., 2024; Wang et al., 2025b). A patch specifies the code change intended to resolve an issue, while a trajectory records the reasoning, actions, and observations used to construct and verify that patch. Supervised fine-tuning (SFT) on successful trajectories improves open software engineering agents (Pan et al., 2025; Yang et al., 2025), but obtaining enough reliable and informative trajectories remains difficult.

Two main challenges limit trajectory construction. First, ***limited trajectory availability***: rejection sampling discards unsuccessful rollouts, so tasks yield no accepted trajectory when all sampled attempts fail, despite available environments and verified reference patches (Pan et al., 2025; Ma et al., 2026). Language-specific synthesis, such as SWE-smith's Python AST mutations, also requires adaptation for reuse across languages (Yang et al., 2025). Second, ***poor trajectory quality*** stems from insufficient isolation of privileged information and redundant steps. Insufficient isolation lets patch-conditioned generators justify actions using unseen entities or future outcomes, yielding rationales based on information unavailable to the student at that decision point. Some work reduces leakage through process graphs and blinded continuations (Ma et al., 2026), but assessing whether evidence supports an action still requires semantic judgment. Even trajectories with test-passing patches can contain repeated exploration, unproductive loops, and avoidable tool errors that contribute little useful evidence but still become imitation targets (Ma et al., 2026; Tao et al., 2026). Such redundancy dilutes useful supervision and can encourage inefficient behavior. Final-test acceptance alone cannot ensure trajectory quality.

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.

Therefore, we introduce **ASSERT** (**A**ccess-**S**eparated **S**ynthesis of **E**xecutable, **R**eplay-grounded **T**rajectories) to address these challenges. ASSERT comprises four stages: *Privileged Solving*, *Action-Plan Repair*, *Execution Grounding*, and *Rationale Reconstruction*. These stages use reference patches to construct and repair action plans, replace draft results with actual execution outputs, and reconstruct the final rationales from visible evidence. ASSERT uses repository toolchains for replay and validation, enabling the same workflow across languages.

To address challenge 1, limited trajectory availability, we propose **Dependency-Constrained Trajectory Repair (DCTR)**, which recovers supervision by repairing patch-anchored action plans within dependency regions identified by failure feedback. This constraint preserves valid substructure, while structured feedback from failed replay or verification directs further repair. These mechanisms allow tasks missed by patch-blind rollouts to contribute verified trajectories.

To address challenge 2, poor trajectory quality, we combine **access isolation with Replay-Certified Decision-Sufficient Slicing (RCDS)**. Stage-specific read permissions and field-specific write ownership separate privileged planning, executed observations, and reconstructed rationales. Only the patch-blind runner writes exported replayable observations; the rationale writer uses the fixed current action and certified prefix evidence, without access to the reference patch, draft rationales, or current and future outcomes. These boundaries restrict direct answer access and hindsight, better approximating the student's inference-time information constraints. RCDS increases supervision density by removing dispensable action–observation groups while preserving evidence needed for later decisions, including informative failed explorations. ASSERT does not eliminate uncertainty in semantic judgments; execution ownership, provenance constraints, and post-deletion validation limit its effect on exported supervision.

We evaluate ASSERT with two teacher models on SWE-bench Verified (Python) and Multi-SWE-bench (Go) (Zan et al., 2025). Its students achieve up to 69.60% and 31.78% Pass@1, exceeding the strongest baseline by 3.00–5.84 percentage points across all four teacher–language settings. They also use fewer interactions on shared successful evaluation tasks. Our experiments further demonstrate that ASSERT produces more and higher-quality training trajectories. It retains 1.62–2.93 times as many trajectories as the largest baseline pool under the evaluated construction protocols. With one trajectory per method for every task in the separate four-method training intersections, ASSERT leads the strongest baseline by 1.80–6.07 percentage points on the full benchmarks.

Our contributions in this paper are as follows:

- We propose ASSERT, an automated tool that generates high-quality training trajectories for software engineering agents while enabling high task utilization across programming languages.
- We introduce stage-specific access isolation and RCDS to improve trajectory quality by better aligning rationale generation with students' inference-time information constraints and removing redundant interactions while preserving decision-relevant evidence.
- We evaluate ASSERT on SWE-bench Verified and Multi-SWE-bench using GLM-5.2 and GPT-5.5 as teacher models. Its students achieve Pass@1 gains of 3.00–5.84 percentage points over the best-performing baseline in each setting and use fewer steps on shared successful tasks.

2 Related Work

Rollout-based trajectory collection and curation. Executable repositories support trajectory collection (Pan et al., 2025; Yang et al., 2025; Jain et al., 2025). SWE-Dev and Skywork-SWE scale collection and task diversity (Wang et al., 2025a; Zeng et al., 2025); OpenSWEtraces adds multilingual traces and behavioral pruning (Ahmad et al., 2026). SWE-Lego combines loss masking, curricula, and semi-resolved traces (Tao et al., 2026), whereas strict rejection sampling retains only successful rollouts.

Privileged-solution guidance. Answers, patches, and demonstrations support rationale synthesis, reverse curricula, and filtering (Zelikman et al., 2022; Xie et al., 2025; Xi et al., 2024; Ma et al., 2025). Lingma SWE-GPT selects traces by localization and patch similarity (Ma et al., 2024). Hindsight Hint Distillation guides new rollouts with hints derived from failures and reference answers (Wang et al., 2026). KAT-Coder-V2.5 (KwaiKAT Team, 2026) uses hints for failure recovery, then regenerates hint-free trajectories from the original task context while holding verified patches

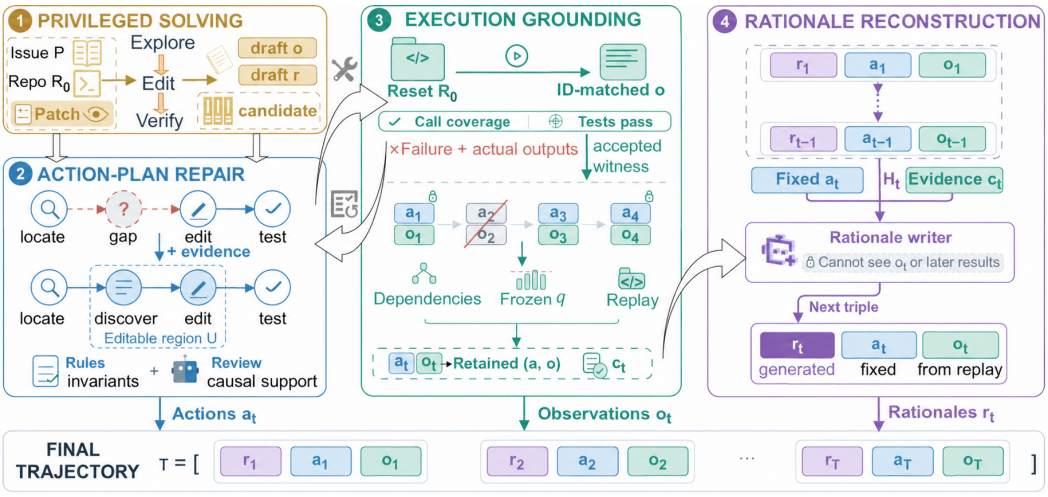

Figure 1: Overview of ASSERT. Reference patches guide plan construction and repair; a patch-blind runner records actual execution outputs, and rationales are reconstructed from fixed actions and certified prefix evidence.

fixed. Prior work combines patch-derived process graphs, executed blinded seeds, and graph-guided mutation with prefix-groundedness checks and final-test filtering (Ma et al., 2026).

ASSERT introduces access-separated trajectory synthesis that integrates dependency-local repair with replay-certified slicing. DCTR recovers failed plans while preserving valid substructure; runner-owned observations ground replayable calls in actual execution. Evidence-preserving slicing and isolated rationale reconstruction increase supervision density and align explanations with evidence available before each action, jointly improving task utilization and trajectory quality.

3 Methodology

3.1 Overview

ASSERT converts an issue and reference patch into a trajectory through four components (Figure 1; recorded case in Appendix A). ***Privileged Solving*** constructs a patch-informed candidate plan. ***Action-Plan Repair*** revises its actions using deterministic rules and reviewer assessment. ***Execution Grounding*** replaces draft outputs of replayable tools with observations obtained by executing the corresponding actions from the pre-fix repository state. For candidates that pass replay and the official tests, it applies Replay-Certified Decision-Sufficient Slicing (RCDS) and passes the retained groups to ***Rationale Reconstruction*** together with their evidence certificates. For each task, ASSERT exports at most one accepted trajectory.

ASSERT targets both eligible-task utilization and exported-trajectory quality. For utilization, reference patches let Privileged Solving and Action-Plan Repair construct candidates for tasks missed by patch-blind rollouts. Execution Grounding then replays each repaired plan from a clean repository, replaces draft results with runner-owned observations, and retains it only after complete replayable-call coverage and official-test acceptance. Failures return to Action-Plan Repair rather than immediately discarding the task. For quality, ASSERT uses two mechanisms. First, write authority is separated by field: patch-aware components may propose actions and draft rationales, only the runner writes replayable observations, and the rationale writer has no direct access to the raw patch or current and future outcomes, which reduces direct answer leakage (Appendix B). Second, RCDS aims to increase usable supervision density by removing dispensable action–observation groups while preserving action predictability, dependencies, replay validity, and evaluator acceptance.

3.2 NOTATION

We represent a task as $x = (P, R_0, p^\star, \mathcal{E})$, where P is the problem statement, R_0 the repository snapshot at the pre-fix commit, $p^\star$ the reference patch, and $\mathcal{E} = (\mathcal{E}_{\mathrm{F2P}}, \mathcal{E}_{\mathrm{P2P}})$ the official fail-to-pass and pass-to-pass test sets.

For T native assistant turns, we write $\tau = (r_t, a_t, o_t)_{t=1}^{T}$, where r_t is the rationale preceding action a_t, a_t contains every tool call emitted in that turn, and o_t contains identifier-matched result slots. The action–observation group $v_t = (a_t, o_t)$ cannot be split. We write $A = (a_1, \ldots, a_T)$, $O = (o_1, \ldots, o_T)$, $V(A, O) = \{v_1, \ldots, v_T\}$; for any trajectory ξ, $V(\xi)$ denotes its action–observation groups. We index all individual tool calls in turn order as $u_1, \ldots, u_n$. A model-authored *draft result* occupies a result slot before replay. The accepted replay witness is $\bar{\tau}$; τ denotes the export after RCDS and Rationale Reconstruction.

The *strict-prefix context* preceding turn t is $H_t = (P, \tau_{<t})$, where $\tau_{<t}$ denotes the trajectory prefix before turn t with prior supervised rationales already reconstructed. For every retained supervised action, an evidence certificate c_t identifies spans attributed as support in P or earlier runner-produced observations by stable source identifiers and content hashes.

3.3 TRAJECTORY CONSTRUCTION

3.3.1 PRIVILEGED SOLVING

Privileged Solving addresses the utilization limit of patch-blind rollout: even when a teacher cannot discover a repair unaided, $p^\star$ provides a verified solution anchor. The patch, however, specifies only the final code change; it does not supply the executable investigation, evidence dependencies, or verification path needed for process supervision. ASSERT must therefore first externalize a complete candidate process before later stages can inspect, repair, and replay it.

The solver explores and edits R_0 while an append helper records an executable action trace at decision time, preserving each call's exact arguments and turn boundary together with a draft rationale and identifier-matched result slot. These draft fields are necessary: the rationale preserves the intended role of each action so Action-Plan Repair can detect missing causal bridges, while the result slot preserves the action–observation structure needed to align later execution. Action-Plan Repair may revise the plan, Execution Grounding replaces every replayable draft result, and Rationale Reconstruction generates a new rationale from certified evidence rather than the draft. This separation lets Privileged Solving supply the otherwise missing process structure without allowing its patch-aware claims to survive merely because they sound plausible.

3.3.2 ACTION-PLAN REPAIR

Privileged access makes a solution reachable, but not necessarily imitable: the candidate may jump to patch-specific entities, omit the discovery that justifies an edit, or violate executable-trace invariants. Action-Plan Repair therefore combines two authorities that neither can replace. Deterministic rules veto mechanically decidable defects such as malformed identifier pairing, invalid edit targets, and missing post-edit verification; a tool-using reviewer judges forward coherence, evidentiary support, and robustness to alternative implementations. Proposal and review may use $p^\star$ and private repository observations, but that information cannot populate result fields: any repository-state premise needed by an exported action must be established by an observable step. The combination prevents fluent review from overriding hard invariants while using semantic judgment where static rules are insufficient (Appendix C).

Dependency-Constrained Trajectory Repair (DCTR). Algorithm 1 summarizes the loop. Lines 1–6 initialize the state and dependency graph; Lines 7–19 propose a dependency-local trial and apply the rule, review, replay, coverage, and evaluator gates. Lines 20–23 feed a rejection report into the next state or stop within the fixed budgets.

An unconstrained reviewer can repair one defect by rewriting unrelated, already valid behavior. DCTR instead treats trajectory repair as bounded program repair over the atomic groups $V(A, O)$ (Algorithm 1). A dependency graph G records state flow, observed-value provenance, and verification coverage; each failure localizes an editable region U. Candidate Insert, Delete, Replace, and

Compose operations must remain within U, preserve dependency closure, and avoid tried signatures before being ranked by changed-unit count and cost $J(A)$. Locality preserves verified substructure, while dependency closure prevents a repair from silently invalidating its consumers.

DCTR further treats rejection as supervision for the next repair rather than as a reason to discard the task. Deterministic, reviewer, replay, coverage, and evaluator failures update memory M, the current (A, O), graph G, and region U; real replay results replace speculation in subsequent rounds. This closed loop concentrates model effort on the failed dependency neighborhood and can recover tasks that rejection sampling would lose. The finite budgets B and tried signatures bound the search. On acceptance, DCTR returns the replay witness $\bar{\tau}$ and rebuilds $G^\star$ for RCDS.

Algorithm 1 Bounded dependency-constrained trajectory repair (DCTR).

Input: problem P; pre-fix repository R_0; reference patch $p^\star$; official tests $\mathcal{E}$; initial action/result skeleton (A_0, O_0); proposal cap K; ranking cost function J; budgets $B = (B_{\text{total}}, B_{\text{rule}}, B_{\text{replay}}, B_{\text{eval}})$.
Output: first accepted replay witness $\bar{\tau}$, or None.

```
priv_ctx, exec_ctx = (P, R0, pstar), (R0, tests)
state = initialize(A0, verified_or_partial(O0))
for round_id in range(B.total):
    // Build the dependency graph
    mechanical = deterministic_edges(state.plan, state.results)
    G = augment_graph(priv_ctx, state, mechanical)
    // Propose and admit a local repair
    if round_id == 0: trial = state.plan
    else:
        proposals = propose(priv_ctx, state, G, K)
        trial = admit(proposals, state, G, J)
    if trial is None:
        report = no_admissible_report(state.region)
        run, obs = None, None
    else:
        state.tried.add(signature(trial))
        // Apply rules, review, replay, coverage, and evaluation
        report, run, obs = apply_gates(trial, G, priv_ctx, exec_ctx)
        if report.accepted: return Accepted(trial, obs)
    // Feed rejection evidence into the next round
    state, stop = reject(state, trial, run, report, G, B)
    if stop: break
return None
```

3.3.3 Execution Grounding

Unlike pipelines that use execution only as a terminal filter, ASSERT makes replay both an *ownership boundary* and a *counterfactual oracle*: it determines what the environment actually returned, and whether a shorter trajectory remains executable and passes the official tests.

Clean replay and the witness. Patch-aware components freeze actions, but never author exported replayable observations. A patch-blind runner restores R_0, executes those actions, and replaces every replayable draft result with the matched environment output. Only a complete, unambiguous replay whose patch passes the official evaluator becomes the witness $\bar{\tau}$; failures return as structured evidence to Action-Plan Repair. While candidate generation depends on the teacher's coding ability, replayable observations come from actual execution in the repository environment. This replay-and-verification procedure can be reused with language-specific toolchains and tests (Appendix D).

Replay-Certified Decision-Sufficient Slicing (RCDS). Shortening a test-passing trace may remove the evidence that makes a later edit imitable. RCDS therefore targets *decision sufficiency*, not length alone, through five coupled mechanisms. First, *proxy-model decision distortion*, measured by a frozen scorer, asks whether removing context makes the teacher's next protected action harder to predict. Second, every retained supervised action carries a *strict-prefix evidence certificate* c_t: addressable, content-hashed spans from P or earlier runner-produced observations attributed as support for the decision. Third, deletion is an executable intervention—the sliced trace is actually replayed and re-evaluated rather than declared redundant by an LLM. Fourth, RCDS preserves *ma-*

terial pivots, including negative observations that change the subsequent plan: for example, a failed symbol search that motivates searching callers is evidence, whereas a malformed command is an action error to delete or, when needed to explain recovery, retain as context while masking the entire erroneous assistant turn. Fifth, Rationale Reconstruction conditions on c_t, not on the patch-aware draft rationale. Together, these mechanisms separate candidate ranking, evidence attribution, and executable certification.

Concretely, let $\emptyset \neq \mathcal{Q} \subseteq V(\bar{\tau})$ be the protected groups, $\mathcal{D} \subseteq V(\bar{\tau}) \setminus \mathcal{Q}$ the deleted groups, and $s \in \{\mathrm{pre}, \mathrm{replay}\}$. For a protected group $v_t = (a_t, o_t) \in \mathcal{Q}$, let $(y_{t,1}, \ldots, y_{t,L_t})$ be the L_t-token serialization of a_t, with $y_{t,<j}$ denoting the tokens before position j. Then $\pi_t^{\mathcal{D},s}$ contains only the tool schema, P, and the retained runner-grounded prefix; it excludes the patch, rationales, current result, and future groups. The frozen scorer q_ψ measures the increase from the witness prefix π_t^0:

$$\ell_\psi(a_t \mid \pi) = -\frac{1}{L_t} \sum_{j=1}^{L_t} \log q_\psi(y_{t,j} \mid \pi, y_{t,<j}), \qquad d_t^s(\mathcal{D}) = \Big[\ell_\psi(a_t \mid \pi_t^{\mathcal{D},s}) - \ell_\psi(a_t \mid \pi_t^0)\Big]_+ .$$

A positive $d_t^s(\mathcal{D})$ means the deletion made a_t harder to predict under q_ψ. RCDS uses this signal to screen and rank candidates, then recomputes it on replayed observations so stale outputs cannot justify a changed slice. To prevent small losses from accumulating—or one crucial action from using the full allowance—RCDS constrains aggregate and worst-action distortion over $\mathcal{Q}$:

$$D_{\mathrm{agg}}^s(\mathcal{D}) = \exp\left(\frac{\sum_{v_t \in \mathcal{Q}} L_t d_t^s(\mathcal{D})}{\sum_{v_t \in \mathcal{Q}} L_t}\right) - 1, \qquad D_{\max}^s(\mathcal{D}) = \max_{v_t \in \mathcal{Q}} \{\exp(d_t^s(\mathcal{D})) - 1\}.$$

A replayed slice is admissible only if $D_{\mathrm{agg}}^{\mathrm{replay}}(\mathcal{D}) \leq \epsilon_{\mathrm{agg}}$ and $D_{\max}^{\mathrm{replay}}(\mathcal{D}) \leq \epsilon_{\max}$, in addition to satisfying the dependency, replay, and evaluator constraints above. Among slices evaluated within its replay budget, RCDS retains the admissible one with the largest lexicographic score

$$S(\mathcal{D}) = \big(\Delta N_{\text{AO-tokens}}(\mathcal{D}),\ \Delta N_{\mathrm{calls}}(\mathcal{D})\big),$$

where both savings are measured against $\bar{\tau}$ and token savings rank first, prioritizing supervision-density gains over raw call reduction. The bounded search excludes rejected deletion assignments and preserves replay-and-test acceptance without claiming global optimality (Appendix E).

3.3.4 Rationale Reconstruction

At inference time, the student must choose its rationale and action from P and the realized history $\tau_{<t}$; it cannot use the action's outcome or the eventual repair. A rationale written with either source of hindsight therefore teaches reasoning under an information state the student will never have. After RCDS freezes the executable trajectory and its evidence certificates, ASSERT reconstructs the rationale for each supervised turn through

$$r_t \sim q_\phi(\cdot \mid H_t, a_t, c_t).$$

The asymmetry of this interface is deliberate: a_t specifies the *semantic target*—what verified behavior must be explained—whereas c_t selects the admissible evidence already present in $H_t = (P, \tau_{<t})$. This lets us preserve the accepted action rather than ask the writer to invent an easier-to-explain substitute, while grounding its rationale in evidence available to the student.

Isolation by construction. Each supervised turn is rendered sequentially from an explicit allowlist: the problem, the already reconstructed prefix, the fixed current action, and its evidence certificate. The raw patch, draft rationale, current and sibling results, future groups, evaluator and reviewer feedback, and repository access are absent from the request; action and observation signatures are immutable. This interface blocks direct access to the raw patch and current or future outcomes, while retaining the fixed action as the explanation target. Action-conditioned reconstruction does not establish that the student can independently infer that action from the prefix.

The runner establishes execution provenance for replayable observations; mechanical checks validate evidence sources and strict-prefix availability in the complete export. Action support and explanation quality remain empirical properties. Rationales, fixed actions, and replay results form τ; subsequent SFT truncation can remove supporting evidence (Appendix F).

4 EXPERIMENTS

We evaluate downstream performance and efficiency across construction teachers, trajectory yield and quality, and the contributions of slicing and rationale reconstruction.

4.1 EXPERIMENTAL SETUP

Datasets. We construct Python trajectories from 2,353 eligible SWE-Gym tasks and evaluate on SWE-bench Verified (500 tasks) (Jimenez et al., 2024); for Go, we use 863 eligible Multi-SWE-RL tasks and the 428-task Go subset of Multi-SWE-bench. The splits are repository-disjoint, and evaluation patches are never used for construction or selection.

Baselines. We compare ASSERT with **STaR** (Zelikman et al., 2022), which regenerates failed rollouts in one patch-conditioned response; **RS** (Pan et al., 2025), which discards failed patch-blind rollouts; and **P2T** (Ma et al., 2026), which guides patch-blind search with a patch-derived process graph. **Vanilla** is the student before SFT.

Implementation Details. We use `gpt-5.5-2026-04-24` and `GLM-5.2` as construction teachers with the OpenHands scaffold. Students start from `Qwen3.5-35B-A3B` and are trained separately for Python and Go. RCDS uses frozen `Qwen2.5-Coder-7B-Instruct` as a proxy scorer. All student models are evaluated using a shared custom ReAct-style tool-use scaffold (Yao et al., 2023). Each trained checkpoint is evaluated five times; data and SFT are fixed across runs. Each evaluation run uses one greedy rollout per task with a 32,768-token context window and a 200-assistant-turn cap; Pass@1 is the primary metric. Further settings are in Appendix F.

4.2 DOWNSTREAM PERFORMANCE AND EFFICIENCY

Downstream effectiveness. Table 1 shows that ASSERT achieves the highest Pass@1[1] in all four settings. For example, GPT-5.5/Python improves from P2T's 66.20% to 69.60%, while the largest gain over the strongest baseline is 5.84 points on Go. The advantage across both teachers and languages supports reuse of the construction workflow beyond a single setting. Failure recovery expands task coverage, while execution grounding, slicing, and reconstruction shape the supervision learned from each task. These full-pool results reflect both factors; Section 4.3 tests whether the advantage persists when training tasks are matched.

Construction cost. Cost is the mean LLM API-query cost (USD) per retained training trajectory (Appendix F). ASSERT costs less than P2T in every setting; for GPT-5.5/Python, it lowers the cost from $28.56 to $19.80 (30.7%) while improving student Pass@1. This combines better downstream performance with cheaper retained supervision relative to P2T. RS and STaR remain much cheaper, so ASSERT trades additional construction effort for stronger training signals. STaR's low cost reflects single-response synthesis without intermediate execution; its weaker students illustrate that inexpensive patch-conditioned traces alone do not ensure equally effective repair supervision.

Student interaction efficiency. Figure 2 compares mean tool calls on a fixed intersection of tasks solved by all four methods in all five runs within each setting. ASSERT uses the fewest calls throughout; for GPT-5.5/Python, it reduces P2T's 52.56 calls to 49.12, a 6.5% reduction. Holding task identity and successful completion fixed makes this a comparison of interaction efficiency on the same solved problems. Together with the higher overall Pass@1, the pattern suggests that ASSERT's supervision supports both successful repair and more economical tool use.

4.3 TRAJECTORY QUALITY

Usable trajectory yield. ASSERT retains the largest pool in every setting (Figure 3), reaching 2.93× the largest baseline pool with GPT-5.5 on SWE-Gym. With one trajectory per successful task, this increase represents broader task coverage. The construction designs help explain the ordering: RS discards failed patch-blind rollouts, whereas ASSERT uses privileged solving and targeted repair to recover candidates that can pass replay and tests. STaR also uses the reference patch, but

[1]Table 1 reports mean Pass@1 over five evaluations of each fixed checkpoint, rounded to two decimal places. Percentage-point gains are computed from unrounded means.

Table 1: **Main evaluation results on benchmarks.** Pass@1 (%) measures student resolution over 500 Python or 428 Go tasks; Cost (USD) is the mean LLM API-query cost per retained training trajectory. Bold denotes the best value among the four methods for each metric. Vanilla is the student before SFT and has no trajectory-construction cost.

Benchmark	Teacher	Metric	Vanilla	STaR	RS	P2T	ASSERT
SWE-bench Verified (Python)	GLM-5.2	Pass@1 ↑	59.60	60.40	63.20	65.80	**68.80**
		Cost ↓	—	**0.024**	0.44	2.34	2.13
	GPT-5.5	Pass@1 ↑	59.60	61.20	61.00	66.20	**69.60**
		Cost ↓	—	**0.065**	3.91	28.56	19.80
Multi-SWE-bench (Go)	GLM-5.2	Pass@1 ↑	20.33	22.90	23.60	25.23	**30.84**
		Cost ↓	—	**0.024**	0.46	2.57	2.38
	GPT-5.5	Pass@1 ↑	20.33	23.83	25.93	25.23	**31.78**
		Cost ↓	—	**0.068**	4.30	32.12	28.71

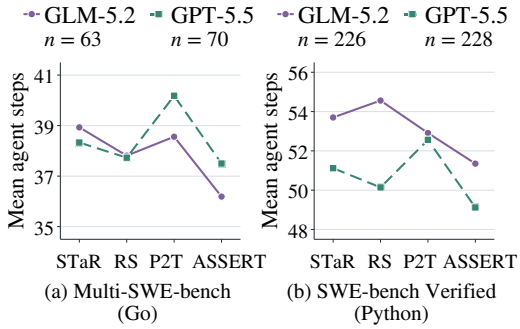

Figure 2: **Student interaction efficiency on shared successful tasks.** Mean tool calls on tasks solved by all four methods in all five runs; n is the fixed intersection size.

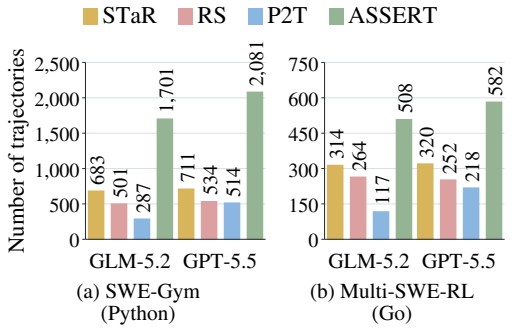

Figure 3: **Retained trajectory counts by construction teacher and dataset.** Panels: SWE-Gym (Python) and Multi-SWE-RL (Go), on separate y-scales.

synthesizes intermediate interactions without executing them; P2T constrains patch-guided search through process graphs and groundedness filtering (Ma et al., 2026). Such filtering can reject otherwise test-passing candidates, explaining why patch access need not increase yield over RS (Appendix F.2). ASSERT combines recovery with execution validation, expanding usable supervision while retaining actual observations and final-test acceptance. These counts characterize each construction protocol; their budgets do not establish equal-compute efficiency.

Per-trajectory supervision quality. To assess whether broader coverage alone explains the gains, we train all methods on their shared task-ID intersection in each setting, using one trajectory per task and the same student and training recipe. Evaluation still covers the full benchmark. ASSERT remains best in all four settings (Figure 4); on the 292 shared GPT-5.5/Python tasks, it reaches 66.20% Pass@1 versus P2T's 64.40%. With task identities and sample counts fixed, this gain supports greater average training utility per trajectory. Full-pool training further raises ASSERT to 69.60% (Table 1), suggesting that recovered tasks supply additional useful supervision. Together, these results support complementary benefits from trajectory utility and task coverage.

We also measure Unsupported Claim Rate (UCR) in the rationale immediately before the first source edit: the fraction of assessable factual claims lacking support in preceding evidence (Min et al., 2023; Es et al., 2024). ASSERT has the lowest UCR in every setting; for GPT-5.5/Python, it achieves 4.58%, compared with P2T's 6.88% and STaR's 24.70%. These results distinguish final-patch correctness from evidence-grounded explanations. The pattern is consistent with prefix-grounded reconstruction limiting unsupported assertions. UCR probes local explanation grounding, comple-

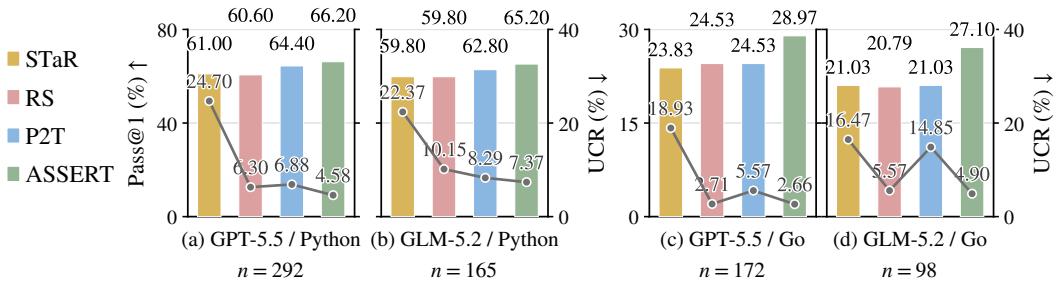

Figure 4: **Matched supervision quality.** Each panel uses n shared training tasks, with one trajectory per method per task. Bars: full-benchmark Pass@1; lines: UCR in the rationale immediately before the first source edit.

Table 2: GPT-5.5/Python ablations. Steps average tool calls over all 500 evaluation tasks, including failures; Cost is the mean LLM API-query cost (USD) per retained trajectory.

Variant	Pass@1(%)	Steps	Cost
Full ASSERT	69.6	49.12	19.80
Without RCDS slicing	68.4	49.68	19.91
Without Rationale Reconstruction	67.6	49.72	17.94
Without RCDS+Rationale Reconstruction	66.4	50.38	16.63

menting the downstream learning evidence from matched-task Pass@1. Scores average per-task ratios over each full intersection; the judging protocol is in Appendix F.3.

4.4 Ablation Study

Table 2 ablates RCDS and Rationale Reconstruction with GPT-5.5/Python (protocol in Appendix F.4). Removing slicing lowers Pass@1 by 1.20 points, while replacing reconstructed rationales with privileged drafts lowers it by 2.00 points; both also increase evaluation tool calls. All variants retain replay-grounded observations and final-test acceptance, so successful replay alone does not deliver the full training benefit. The results support removing unnecessary context and replacing privileged explanations with rationales grounded in prior evidence.

Reconstruction also exposes a quality–cost trade-off: omitting it lowers retained-trajectory cost from $19.80 to $17.94, but sacrifices downstream accuracy. Removing both components produces the largest performance loss, supporting their combined use. Additional DCTR ablations show that unrestricted repair and independent resampling both underperform local, failure-guided repair (Appendix F.7), supporting targeted refinement of privileged candidates.

5 Conclusion

We presented ASSERT, a framework for converting reference patches into executable training trajectories for software engineering agents. DCTR recovers supervision from failed plans, while access isolation and RCDS improve evidence grounding and reduce redundant interactions. Across two teachers and two languages, ASSERT retains 1.62–2.93 times as many trajectories as the largest baseline pool and improves student Pass@1 by 3.00–5.84 percentage points over the strongest baseline in each setting, with fewer interaction steps on shared successful evaluation tasks. ASSERT-trained students retain a 1.80–6.07 percentage-point advantage over the strongest baseline when all methods use one trajectory per task from each setting's four-method training-task intersection. These results suggest that ASSERT can leverage existing repair knowledge to increase supervision availability and improve downstream student performance.

AI USE STATEMENT

In this work, we used generative AI models to generate synthetic training trajectories and perform the model-based review, rationale reconstruction, and unsupported-claim auditing described in the method and experimental protocols. We also used generative AI tools to provide feedback on experimental design, suggest parameter settings, and assist with inspecting and reporting experimental results. Additionally, these tools assisted with drafting, translation, editing, literature discovery and comparison, reference checking, and preparing LaTeX and plotting code. ASSERT trajectories undergo the replay, evidence checks, and official-test validation described in the paper; these checks do not establish the correctness of every model-generated explanation or judgment. All AI-assisted content has been manually reviewed by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Wasi Uddin Ahmad, Nikolai Ludwig, Somshubra Majumdar, and Boris Ginsburg. Open-SWE-Traces: Advancing dual-mode multilingual distillation for software engineering agents. *arXiv preprint arXiv:2606.16038*, 2026.

Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. RAGAs: Automated evaluation of retrieval augmented generation. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, pp. 150–158. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.eacl-demo.16. URL https://aclanthology.org/2024.eacl-demo.16/.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Conference on Language Modeling (COLM)*, 2025. URL https://openreview.net/forum?id=7evvwwdo3z.

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, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

KwaiKAT Team. KAT-Coder-V2.5 technical report. *arXiv preprint arXiv:2607.05471*, 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.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma SWE-GPT: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. SoRFT: Issue resolving with subtask-oriented reinforced fine-tuning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 11427–11441. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.ACL-LONG.559. URL https://doi.org/10.18653/v1/2025.acl-long.559.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12076–12100. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.741. URL https://aclanthology.org/2023.emnlp-main.741/.

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 Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*, 2026. URL https://arxiv.org/abs/2601.01426.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-Dev: Building software engineering agents with training and inference scaling. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pp. 3742–3761. Association for Computational Linguistics, 2025a. doi: 10.18653/V1/2025.FINDINGS-ACL.193. URL https://doi.org/10.18653/v1/2025.findings-acl.193.

Shengjie Wang, Guanghe Li, Zonghan Yang, and Yang Gao. Hindsight hint distillation: Scaffolded reasoning for SWE agents from CoT-free answers. *arXiv preprint arXiv:2605.11556*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025b. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zhiheng Xi, Wenxiang Chen, Boyang Hong, Senjie Jin, Rui Zheng, Wei He, Yiwen Ding, Shichun Liu, Xin Guo, Junzhe Wang, Honglin Guo, Wei Shen, Xiaoran Fan, Yuhao Zhou, Shihan Dou, Xiao Wang, Xinbo Zhang, Peng Sun, Tao Gui, Qi Zhang, and Xuanjing Huang. Training large language models for reasoning through reverse curriculum reinforcement learning. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 54030–54048. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/xi24a.html.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. SWE-Fixer: Training open-source LLMs for effective and efficient GitHub issue resolution. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pp. 1123–1139. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-ACL.62. URL https://doi.org/10.18653/v1/2025.findings-acl.62.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In Amir Globerson, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software

engineering agents. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/5afa9cb1e917b898ad418216dc726fbd-Abstract-Datasets_and_Benchmarks_Track.html.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/639a9a172c044fbb64175b5fad42e9a5-Abstract-Conference.html.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and Yahui Zhou. Skywork-SWE: Unveiling data scaling laws for software engineering in LLMs. *arXiv preprint arXiv:2506.19290*, 2025.

A CASE STUDY: STRINGARRAY SERIALIZATION

In `bokeh__bokeh-12841`, plotting a DataFrame with a pandas string-dtype column raises a `StringArray` serialization error. The GPT-5.5/Python trace excerpt and task metadata are stored in `case_evidence/`; message indices below are zero-based.

Evidence-guided repair. After reading the serializer, normalization helper, and DataFrame extraction code, the agent probes the column type and conversion behavior. These probes reveal a private `PandasArray` import failure, distinct from the reported serialization error, and support using the public `ExtensionArray` interface. Three edits update the serializer and helper while preserving the ndarray encoding route. The focused checks below are distinct from the complete official test suite used for trajectory acceptance.

RCDS slicing. RCDS removed nonessential action–observation groups from the constructed trajectory while retaining the groups in Table 3. Each message pair contains an action and its observation; ranges spanning several pairs contain multiple groups.

Table 3: Retained action–observation groups in the StringArray serialization case.

Messages retained	Recorded action and result	Role in the retained evidence chain
12–13; 16–17	Type and conversion probes	Identify the private-import failure and justify the public `ExtensionArray` check and `to_numpy()` conversion.
24–25	Focused test: `1 failed`	Establish the pre-edit failure; this negative result provides diagnostic evidence.
30–35	Three edits in two source files	Implement the supported type check and normalization changes.
38–39; 44–45	Plotting check: `saved`; focused tests: `2 passed`	Validate the issue-like path and the serializer/helper behavior after editing.

RCDS deleted the redundant rereading group (messages 42–43), whose inspection of the edited serializer and helper duplicated information already available in the edit feedback and diff (messages 30–35 and 40–41), while retaining the diagnostic probes and validation steps.

B COMPONENT PERMISSIONS

Table 4 specifies read access and write ownership. Fixed actions may carry patch-derived information; access isolation does not imply statistical independence. Non-evidentiary passthrough results remain in exports but are masked for scoring and reconstruction. Grouped calls share one rationale, with all sibling results withheld. Output checks enforce length and reject references to omitted fields, without certifying semantic support or recovering the original decision process.

Table 4: Component permissions. "Private" fields are not exported. Passthrough results are tagged non-evidentiary and are outside replay-provenance guarantees.

Component	May read directly	May write
Privileged Solving	$P, p^\star, R_0$, solver tool outputs	actions, draft rationales, draft results
Action-Plan Repair	$P, p^\star, R_0$, candidate fields, private tool outputs, gate reports	candidate actions and drafts; private review report; draft-result invalidation
Execution Grounding	P, fixed actions and R_0; $\mathcal{E}$ only for the official evaluator; accepted witness and frozen scorer for RCDS	replayable result fields, evaluator report; RCDS deletion set, replayed incumbent, and certificates
Rationale Reconstruction	P, strict-prefix context, current action, evidence certificate	current reconstructed rationale only

C Repair Checks and Feedback

Checks. Rules check message structure, call/result pairing, edit targets, regeneration commands, and post-edit verification; error-level violations veto reviewer approval. Trace-artifact bookkeeping and its result slots are removed before repair. All gates identify failures by stable source-unit identifiers.

Dependency-local repair. Mechanical edges connect file writes to dependent calls, observed values to their consumers, and edits to verification. Reviewer-proposed semantic edges may supplement but not remove these edges. Missing-premise failures expand the repair region toward prerequisite evidence; execution or verification failures expand it toward affected successors and checks. Changed failure signatures reseed the region. Edits use existing source IDs or insertion anchors and preserve atomic groups and dependency closure; an intermediate state can be removed only if no retained group needs it. Untried admissible proposals are ranked by changed-unit count, then $J(A)$ (Appendix F).

Acceptance. Candidates pass rules, review, clean replay, alignment/coverage checks, and official tests in that order. A failed verification remains blocking until a later check passes with the same command family, working directory, targets, configuration, material flags, and environment fingerprint; missing fields or narrower scope cannot discharge it. DCTR returns the first accepted witness within budget, or no trajectory.

D Replay Alignment

Each replay restores the pre-fix commit in an isolated sandbox. Fixed calls are matched to recorded executions in order, using sandbox-relative paths, tokenized shell commands, newline-normalized edit strings, and matching runtime call IDs when available. A maximum order-preserving matching must cover every replayable call and fill its result slot, including shell stdout, stderr, and exit status. Ambiguous matches between repeated calls cause rejection.

Unmatched executions are allowed only for explicitly whitelisted read-only bookkeeping; state-changing or unclassifiable executions block acceptance. Passthrough calls are excluded from matching and carry no execution-provenance guarantee. The official evaluator runs only after alignment, coverage, and other blocking checks pass.

E RCDS Search and Scoring Details

The protected set $\mathcal{Q}$ is fixed before search: patch-producing edits, covering checks, and material pivots with their dependent responses. Required predecessors, including uncertain dependencies, are preserved. Each unprotected group seeds a deletion expanded to its surviving dependents; proposals touching protected groups or required evidence are rejected, and duplicates merged. Pre-replay scoring uses incumbent observations. Candidates satisfying both distortion bounds rank by token savings, then call savings relative to $\bar{\tau}$; lower aggregate distortion and source order break ties.

For each retained supervised action, $c_t = \{(\mathrm{src}_k, b_k, e_k, h_k)\}_{k=1}^{m_t}$ identifies supporting spans $[b_k, e_k)$ and content hashes in P or earlier runner observations. The reviewer selects premises; the validator rejects missing, post-action, out-of-range, or hash-mismatched spans. Deletion cannot remove cited evidence. These checks establish provenance and temporal availability, while semantic support remains a reviewer judgment. Operational-error turns retained as recovery context receive zero SFT loss. Patch-aware draft rationales are excluded from both exports and reconstruction prefixes.

The highest-ranked untried candidate is cleanly replayed, and all distortions and savings are recomputed. It replaces the incumbent only if all dependency, evidence, distortion, replay, and test constraints pass and $S(\mathcal{D})$ strictly improves. Rejection excludes the exact deletion set, not its supersets. Search stops after four candidate replays or when no proposal remains, returning the best certified incumbent, initially the original witness.

F SUPPLEMENTARY EXPERIMENTAL DETAILS

F.1 EXPERIMENTAL SETTINGS AND COUNTING RULES

Cost and task identity. Cost is the mean LLM API-query cost (USD) per retained training trajectory, as in Section 4.2. Missing GLM token usage is estimated from other calls in the same trajectory. Tasks are deduplicated by repository and issue/pull-request ID; repair and slicing candidates do not create new tasks. ASSERT retains at most one trajectory per task and setting. Go training excludes evaluation repositories before construction; the Python splits are also repository-disjoint.

Construction. All methods allow up to five complete attempts per task, independently of internal repair budgets, with limits of 200 assistant turns and 20 tool calls per turn. All generative roles use the setting's teacher and share ASSERT's context and output limits, including STaR's single-response generation. Temperatures are 0.6 for rollouts/candidates, 0.4 for repair/graph construction, and 0.2 for review/reconstruction; top-$p = 0.95$. Malformed structured outputs allow two retries. Other P2T settings follow the original paper (Ma et al., 2026). Shared limits do not imply equal compute because construction roles differ.

DCTR uses $B_{\text{total}} = B_{\text{rule}} = 8$, $K = 3$, and $B_{\text{replay}} = B_{\text{eval}} = 3$. The total budget includes the initial candidate; Invocation budgets include failed calls. After the changed-unit count, ranking uses $J(A) = N_{\text{calls}}(A) + 10^{-3} N_{\text{action-tokens}}(A)$, excluding rationales and observations. RCDS uses frozen `Qwen/Qwen2.5-Coder-7B-Instruct`, $(\epsilon_{\text{agg}}, \epsilon_{\max}) = (0.02, 0.10)$, and four additional replay attempts per witness. These bounds constrain proxy action predictability, not semantic correctness.

SFT. The student is Qwen3.5-35B-A3B: two epochs, seed 42, LoRA rank/alpha/dropout of 32/64/0.05, learning rate 10^{-4}, and effective batch size 16 (micro-batch one per replica, adjusted accumulation). AdamW uses $(\beta_1, \beta_2) = (0.9, 0.999)$, $\epsilon = 10^{-8}$, and zero weight decay; 3% linear warmup precedes linear decay, with gradient clipping at 1.0. Loss covers assistant tokens only, with operational-error turns additionally masked in ASSERT.

Truncation. The $L = 32{,}768$-token limit includes chat-template markers. Overlength records keep the complete system/problem prefix (p tokens), then allocate $\lfloor (L-p)/2 \rfloor$ tokens to the head and $\lceil (L-p)/2 \rceil$ to the tail. The longest fitting prefix and suffix of complete assistant-turn/tool-result groups are retained; groups are not split and unused boundary space remains unused. Token IDs and loss labels use identical retained-position indices; system, problem, tool, padding, and operational-error tokens retain zero loss. Replay and certificates apply before truncation, not automatically to shortened prefixes. Matched-task results therefore include differences in context retained during SFT; Table 7 gives pre-truncation lengths.

Evaluation. Each configuration fixes one training dataset and one checkpoint (SFT seed 42); only evaluation is repeated five times. Greedy rollouts use 200 assistant turns, a 32,768-token context, and at most 8,192 output tokens per response. Reported metrics average these five evaluations, not independent construction or training runs. For solved-task counts s_r, $\overline{\text{Pass@1}} = 100 \sum_{r=1}^{5} s_r/(5N)$, with $N = 500$ (Python) or 428 (Go). Gains use unrounded means and the strongest baseline in each setting. For Figure 2, the comparison set is $S = \bigcap_m \bigcap_{r=1}^{5} S_{m,r}$, where $S_{m,r}$ contains tasks solved by method $m \in \{\text{STaR}, \text{RS}, \text{P2T}, \text{ASSERT}\}$ in run r. Tool calls are averaged over S and then runs; $|S|$ is 226/228 for Python and 63/70 for Go (GLM-5.2/GPT-5.5).

F.2 INTERPRETING P2T TRAJECTORY YIELD

P2T converts a reference patch into a process graph and constructs trajectories through patch-blind continuations. Curator-modified candidates face groundedness checks, segment selection balances graph progress against length, and final trajectories must pass tests (Ma et al., 2026). Patch guidance supplies intermediate targets but does not ensure acceptance within the attempt budget. Figure 3 counts survivors of this complete protocol under our teachers and budgets, explaining why patch access need not yield more trajectories than RS.

F.3 UNSUPPORTED-CLAIM AUDIT

Automated audit. The window is the rationale before the first production-source edit; test-only edits do not qualify. The method-blind `gpt-5.5-2026-04-24` judge (medium reasoning effort) checks atomic factual claims against the problem and preceding observations, excluding the reference patch and current/future outputs. Plans and explicit hypotheses are excluded, but their factual premises remain assessable. Judgments record evidence citations or missing premises. UCR averages per-window unsupported/assessable claim ratios; zero-fact windows are undefined, not zero. The fixed pools contain 292/172/165/98 tasks (GPT-5.5/Python, GPT-5.5/Go, GLM-5.2/Python, GLM-5.2/Go), independently of evaluation repeats.

Human audit. Ten tasks per setting (seed 42) are sampled with all four methods, giving 40 windows per method. Two annotators independently label a manually checked claim list under the same evidence rules, blinded to method, teacher, certificates, and automated labels. Current actions are hidden until factual labels are fixed; disagreements are adjudicated while blinded. Table 5 averages assessable-window ratios within settings, then weights settings equally. This local audit assesses rationale grounding, not action correctness or observation authenticity.

Table 5: Human UCR on paired first-source-edit windows, averaged equally over four settings.

Method	Sampled windows	Human UCR (%) ↓
STaR	40	20.54
RS	40	6.48
P2T	40	8.82
ASSERT	40	**4.66**

F.4 ABLATION PROTOCOL

Table 2 uses the same GPT-5.5/Python pre-slicing witnesses, task IDs, sample counts, and SFT settings. Without RCDS, full witnesses undergo reconstruction; without reconstruction, action-aligned drafts archived during repair replace reconstructed rationales. Removing both uses full witnesses and their drafts. All variants retain runner observations, final-test acceptance, and operational-error masking; supervised-token counts may differ. Steps average tool calls, not assistant turns, over all 500 tasks including failures and then five evaluations, unlike Figure 2's shared-success set. Cost follows Section 4.2.

F.5 RCDS THRESHOLD SENSITIVITY

Table 6 varies one bound at a time around $(0.02, 0.10)$ for GPT-5.5/Python. Each setting reruns RCDS and rationale reconstruction on the full training pool, with input witnesses, other construction settings, SFT recipe/seed, and evaluation held fixed. Pass@1 ranges from 68.60% to 69.60%; all alternatives lie within 1.00 point of the default. The pattern is consistent with a compression–context trade-off: tighter bounds restrict redundancy removal, while looser bounds permit larger changes in proxy action predictability.

Table 6: RCDS threshold sensitivity for GPT-5.5/Python. Student Pass@1 is evaluated on SWE-bench Verified.

Configuration	ϵ_{agg}	$\epsilon_{\max}$	Pass@1 (%)
Tighter aggregate bound	0.01	0.10	68.60
Default	0.02	0.10	**69.60**
Looser aggregate bound	0.04	0.10	69.20
Tighter per-action bound	0.02	0.05	69.00
Looser per-action bound	0.02	0.20	68.60

F.6 Training-Trajectory Statistics

Calls count recorded `execute_bash` and `str_replace_editor` entries, including grouped siblings; `finish`/`think` are excluded, and a multi-command shell invocation counts once. Recorded STaR calls do not imply actual execution during construction. Tokens sum saved message contents using the Qwen3.5-35B-A3B tokenizer with `add_special_tokens=False`: system/problem appear once, with rationales, actions, and observations; outer chat markers, repeated API prefixes, and SFT truncation are excluded.

On the same 292 tasks, ASSERT uses 8.65% fewer tokens than P2T with higher matched-task Pass@1 (Table 7; Figure 4). STaR is shorter still but performs worse, supporting preservation of useful decision context rather than length minimization alone.

Table 7: Training-record statistics on the same 292 Python tasks (GPT-5.5). Calls count individual recorded environment-tool entries; tokens count saved message contents under a common tokenizer, excluding outer chat-template markers.

Method	Shared tasks	Mean calls	Mean tokens
RS	292	29.41	31,716
STaR	292	7.52	6,914
P2T	292	29.61	31,675
ASSERT	292	26.05	28,934

F.7 DCTR Construction Ablations

All variants use GPT-5.5, 2,353 eligible Python tasks, up to five complete attempts per task, and the same initial privileged candidate within each attempt. Accepted trajectories receive identical RCDS, reconstruction, and SFT treatment; retained task sets may differ because recovery is the intervention. Evaluation follows Appendix F.1 on all 500 tasks.

Without DCTR, failed candidates receive no internal repair. Without locality, repair may change any group or insertion gap while preserving feedback, atomic grouping, dependency validity, and acceptance gates. Independent resampling replaces repair proposals with fresh solves from the clean repository, issue, and patch, without prior plans or failure feedback; ranking uses $J(A)$, then generation order. The latter two variants retain Full ASSERT's internal search/verification budgets and generation limits (Appendix F.1), which match available budgets, not realized token consumption. Pass@1 tests downstream utility, not construction yield or recovery expenditure.

Table 8: Supplementary DCTR construction ablations for GPT-5.5/Python. Pass@1 averages five evaluations of each fixed checkpoint on all 500 SWE-bench Verified tasks.

Variant	Pass@1 (%)
Full ASSERT	**69.60**
Without DCTR (single-pass construction)	66.20
Without dependency locality	67.00
Independent privileged resampling	67.60

Full ASSERT leads independent resampling by 2.00 points and unrestricted repair by 2.60 points, supporting failure-guided refinement that preserves valid substructure. Removing internal repair reduces Pass@1 by 3.40 points, showing the downstream cost of discarding rejected candidates.

G Limitations

ASSERT requires reference patches, executable repositories, and task-level verification. Replay and access isolation do not guarantee semantic correctness of patches or rationales. Our separately trained Python and Go students establish workflow reuse with two teachers; cross-language student transfer and generalization to other languages or teachers remain untested.