TRACE-SWE: TRAJECTORY REFINEMENT WITH ACTION-ORIENTED EXECUTION EVIDENCE FOR CODE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Effective supervised fine-tuning (SFT) of code agents benefits from trajectories that connect intermediate actions with execution feedback. Real-user interaction histories from code agents such as Claude Code provide these records. However, they interleave useful actions with redundant exploration and failed attempts, limiting their value as direct training supervision. We introduce **TRACE-SWE** (Trajectory Refinement with Action-oriented Execution Evidence for Code Agents), a framework that refines real-user trajectories into action-oriented supervision using recorded execution evidence. For successful trajectories assessed as completing user requests, TRACE-SWE uses recorded outcome evidence to retain relevant actions and their supporting context while removing unnecessary steps. Inspired by cloze-style learning, TRACE-SWE then masks individual actions or action sequences and reconstructs them from preceding and subsequent context. For failed trajectories with unresolved user requests, TRACE-SWE uses recorded failure evidence to construct corrective supervision. TRACE-SWE improves performance on SWE-bench Verified, Pro, and Multilingual across four open-weight models. On LING-3.0-FLASH, it achieves an average gain of 6.70 percentage points across six SWE-bench configurations under THINK inference, with improvements also extending to Terminal-Bench. TRACE-SWE also reduces average interaction rounds and non-cached tokens per solved task, indicating improved efficiency.

1 INTRODUCTION

Code agents powered by large language models (LLMs) address software-engineering tasks through code search, file inspection, editing, and execution-based verification (Yang et al., 2024; Wang et al., 2025b; Xia et al., 2024; Zhang et al., 2024). Training these agents requires supervision that links intermediate actions to execution feedback and the surrounding repository context. A common approach collects agent rollouts on executable tasks in sandboxed repositories with controlled initial states and outcome checks (Pan et al., 2025; Yang et al., 2025; Jain et al., 2025; Badertdinov et al., 2025; Chen et al., 2026). This enables systematic verification, but requires environment preparation, repeated agent inference, and test execution. Moreover, predefined tasks do not fully capture the changing requests and user feedback of everyday use.

Histories generated during routine use of code agents such as Claude Code offer another source of supervision. They record user requests, actions, execution feedback, and corrections without requiring new agent rollouts for collection (Baumann et al., 2026; Zi et al., 2025). These records provide evidence for assessing task progress and capture user feedback when the agent falls short. However, they are not curated demonstrations: useful actions are mixed with redundant exploration, abandoned edits, and failed attempts (Baumann et al., 2026; Zi et al., 2025; Chi et al., 2025). Their value therefore depends on separating useful supervision from behavior that should not be imitated.

Execution-based filtering and error masking improve trajectory quality (Tao et al., 2026; Ludwig et al., 2026; Ahmad et al., 2026), but learning from real-user histories still presents three challenges.

First, deciding what to retain. A successful outcome does not make every preceding step useful: the history may include repeated searches, abandoned edits, and unsuccessful attempts. Training on the entire history can reinforce these behaviors, while removing steps without considering their

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.

dependencies can discard necessary context. Orchard-SWE selects progress segments, and P2T uses a developer reference patch to guide trajectory construction (Peng et al., 2026; Ma et al., 2026). However, a locally useful action may depend on earlier observations or edits, and reference patches are not assumed available for real-user requests. The challenge is to use recorded outcome evidence to identify useful steps and retain the context needed to understand them.

Second, learning from key actions. In standard autoregressive supervised fine-tuning (SFT) on trajectories, each action is predicted from the preceding history. This provides the context for choosing an action, but does not directly use its later results as context for learning that action. Yet later feedback can show whether an edit passed verification, a command exposed an error, or a step supported further progress. Preceding context and subsequent outcomes therefore provide different evidence for understanding key actions. This motivates supervision that combines both sources to connect an action's rationale with its observed consequences.

Third, learning from failed attempts. Outcome-filtered training often excludes failed trajectories because they do not provide successful demonstrations (Pan et al., 2025). Discarding them also loses execution errors, attempted fixes, and user feedback that can reveal why a request remains unresolved. Simply imitating the history, however, risks reinforcing the same mistakes. Retaining useful fragments from failed histories preserves some learning signals (Tao et al., 2026; Ahmad et al., 2026; Peng et al., 2026), but does not itself explain what should be corrected. Learning from these failures requires turning recorded evidence into useful diagnoses and repair guidance.

We introduce **TRACE-SWE**, a trajectory refinement framework that uses recorded execution evidence to derive action-oriented supervision from real-user histories. TRACE-SWE first selects histories containing code modifications. An LLM uses recorded edits, verification results, and user feedback to assess whether each history fulfills the user's request. Successful trajectories are routed to refinement, while failed trajectories with useful evidence are routed to correction. For successful trajectories, TRACE-SWE starts from edits supported by successful verification results and traces their dependencies on earlier actions and observations. It retains these actions and their supporting context while removing unrelated steps. Inspired by cloze-style learning, TRACE-SWE then masks single actions or action sequences in the refined history. Reconstruction uses both preceding context and subsequent outcomes, making the connection between an action's rationale and its observed consequences an explicit learning target. The targets retain the original tool calls. For failed trajectories, TRACE-SWE uses recorded failure evidence to construct corrective supervision with diagnoses and proposed repair and verification steps. The resulting single-step reconstruction, multi-step reconstruction, and failure-correction datasets are combined for SFT. TRACE-SWE improves SWE-bench Verified, Pro, and Multilingual performance across four open-weight models (Jimenez et al., 2024; Deng et al., 2025; Khandpur et al., 2025). On LING-3.0-FLASH, the overall average gain across the three benchmarks and two scaffolds is 6.70 percentage points under THINK inference. Gains also extend to Terminal-Bench (Merrill et al., 2026; The Terminal-Bench Team, 2026), consistent with learning command-line actions from SWE trajectories. Otherwise, TRACE-SWE reduces average interaction rounds and non-cached tokens per solved task, indicating improved efficiency.

Our contributions are:

- We introduce TRACE-SWE, a trajectory refinement framework for real-user code agent histories. It refines successful and failed interactions into action-oriented supervision using recorded execution evidence, without collecting new sandbox rollouts.
- We construct single-step reconstruction, multi-step reconstruction, and failure-correction data. These three types of supervision are combined for SFT.
- TRACE-SWE improves four models' performance on SWE benchmarks, averaging $+6.70$ points on LING-3.0-FLASH, with gains also on Terminal-Bench for this model. Ablations support all three supervision types, and efficiency analysis shows fewer interactions and lower token costs per solved task.

2 RELATED WORK

Research on code agents spans task and environment construction, code verification, real-world developer data, and trajectory-based learning from interaction records. Table 1 compares selected methods for trajectory-based SFT.

Table 1: **Trajectory-supervision methods for code agents.** Failed-trace supervision uses unsuccessful histories or their useful fragments. Context preservation denotes explicit dependency or prerequisite handling. Action selection identifies trainable actions or segments; key-step supervision applies selective loss or dedicated targets. ✓: explicitly reported; ✗: not reported.

Method	Execution feedback	Action selection	Failed-trace supervision	Context preservation	Key-step supervision
SWE-Gym Pan et al. (2025)	✓	✗	✗	✗	✗
SWE-Lego Tao et al. (2026)	✓	✓	✓	✗	✓
SWE-HERO Ludwig et al. (2026)	✓	✗	✗	✗	✗
Open-SWE-Traces Ahmad et al. (2026)	✓	✗	✓	✗	✗
Orchard-SWE Peng et al. (2026)	✓	✓	✓	✗	✓
P2T Ma et al. (2026)	✓	✓	✗	✓	✓
TRACE-SWE (ours)	✓	✓	✓	✓	✓

Task and environment construction. SWE-smith synthesizes repair tasks by modifying code to break tests (Yang et al., 2025). SWE-Gym pairs repository tasks with environments and tests; R2E-Gym, SWE-rebench, and SWE-Universe further scale executable training resources (Pan et al., 2025; Jain et al., 2025; Badertdinov et al., 2025; Chen et al., 2026). For environment setup, Repo2Run builds Docker images through iterative execution feedback, whereas RepoST isolates functions and dependencies in sandbox scripts (Hu et al., 2025; Xie et al., 2025). These resources support controlled data collection with defined execution conditions and outcomes, but prepared tasks differ from open-ended user workflows.

Code verification and test generation. Randoop uses execution feedback to guide random test generation (Pacheco et al., 2007). TestPilot generates tests from code and documentation and revises them using failure messages (Schäfer et al., 2024). SWE-Dev synthesizes tests for patch evaluation during code-agent training and inference (Wang et al., 2025a). These methods generate test-based feedback, whereas TRACE-SWE organizes supervision using execution feedback already recorded.

Real-world developer data. SWE-chat captures user–agent sessions and studies user intervention and code retention (Baumann et al., 2026). AgentPack collects co-authored edits, while Copilot Arena gathers preferences over code suggestions (Zi et al., 2025; Chi et al., 2025). Interaction sessions preserve action-observation sequences alongside feedback and changing requirements, while edit collections and preference datasets provide edit-level and preference-level signals. Richer context supports trajectory learning, but recorded behavior is not necessarily suitable for direct imitation.

Agent learning from trajectories and feedback. AgentBank studies cross-task trajectory tuning, while SWE-HERO selects successful rollouts using execution outcomes (Song et al., 2024a; Ludwig et al., 2026). SWE-Lego combines step-level error masking with supervision from partially resolved trajectories (Tao et al., 2026). Orchard-SWE retains successful trajectories and selectively supervises progress segments from unresolved ones (Peng et al., 2026). Open-SWE-Traces includes unresolved traces in its training corpus (Ahmad et al., 2026). Beyond SFT, Reflexion reuses verbal feedback without weight updates, and exploration-based trajectory optimization (ETO) uses failed attempts to form preference pairs (Shinn et al., 2023; Song et al., 2024b). P2T derives prerequisite graphs from external reference patches to guide trajectory construction (Ma et al., 2026). TRACE-SWE instead constructs single-step reconstruction, multi-step reconstruction, and failure-correction data directly from real-user histories using recorded execution evidence. Its cloze-style reconstruction uses preceding context and recorded outcomes, and the three datasets are combined for SFT.

3 TRACE-SWE Framework

TRACE-SWE transforms real-user code agent trajectories into action-oriented supervision grounded in their recorded execution evidence. Supervised actions include command-line operations for repository exploration, code modification, and verification, linking commands to their observed

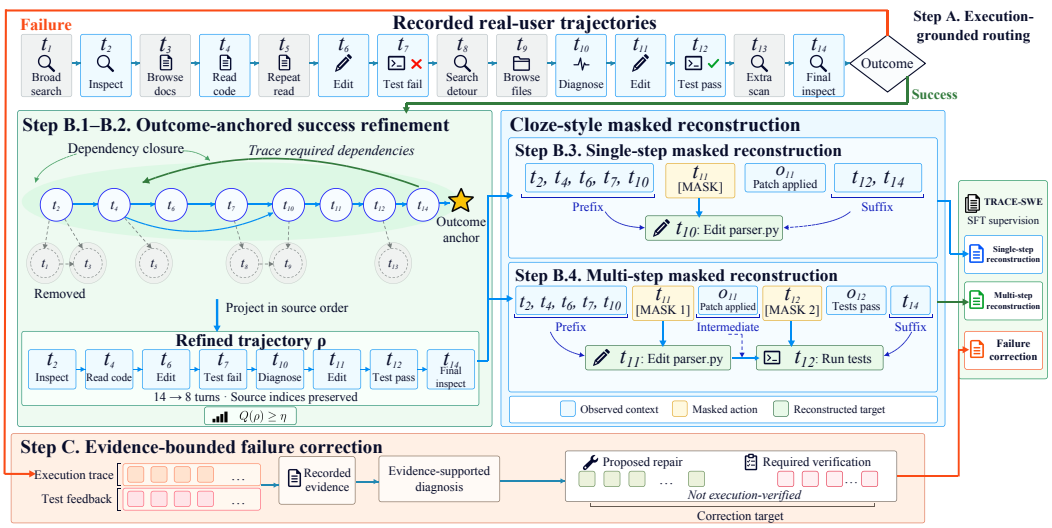


Figure 1: Overview of TRACE-SWE. Step A retains code-modifying trajectories and uses an LLM to route successful interactions to refinement and failed interactions with learning value to correction. Step B.1 identifies outcome-supported anchors and constructs typed dependencies from recorded execution evidence. Step B.2 retains dependency-linked context in source order and applies an LLM confidence check to the refined trajectory. Step B.3 masks a selected action and reconstructs it using preceding and subsequent context with distinct temporal roles. Step B.4 extends this cloze construction to multiple actions while retaining their intervening observations. Step C constructs correction targets that distinguish evidence-grounded diagnoses from unexecuted repair and verification proposals.

execution feedback. As shown in Figure 1, three stages route trajectories by assessed outcome, refine successful trajectories for cloze-style learning of key actions, and construct failure correction targets.

3.1 Problem Formulation

We represent a trajectory as $\tau = (u, (a_1, o_1), \ldots, (a_T, o_T))$, where u is the task, T the number of assistant turns, a_t the assistant action at turn t (text and any tool calls), and o_t its associated tool results in recorded order, empty for turns without tool calls. User interventions remain in the source context, and all subsequent action indices refer to assistant turns. Let z summarize recorded execution evidence and the LLM assessment in Step A, and let $\mathcal{T}^+$ and $\mathcal{T}^-$ denote the resulting disjoint sets of successful and failed trajectories. For each trajectory, the construction yields supervised records

$$\Phi(\tau, z) = \begin{cases} \mathcal{Y}_\tau^{\text{single}} \cup \mathcal{Y}_\tau^{\text{multi}}, & \tau \in \mathcal{T}^+, \\ \mathcal{Y}_\tau^{\text{correction}}, & \tau \in \mathcal{T}^-, \\ \varnothing, & \text{otherwise.} \end{cases} \tag{1}$$

Here Φ is the construction procedure, $\mathcal{Y}_\tau$ denotes a per-trajectory record set of the indicated type, and $\varnothing$ denotes an empty set; subsequent filtering may also leave a routed trajectory with no records. Each record pairs a constructed input with a target response; retained tool calls and observations remain source-exact, while proposed correction actions are marked unexecuted. Automatic checks cover format, source-call fidelity, protocol, and correction fields; human auditors sample records for context adequacy, reasoning–action consistency, evidence attribution, and fabricated execution (Appendix A.2).

3.2 Step A: Execution-Grounded Routing

TRACE-SWE first retains trajectories containing code modifications and uses an LLM to assess task outcomes from recorded edits, verification results, other execution observations, and user feedback. Success is assessed from recorded evidence that the requested code change is complete and consistent

with the final observed state. Trajectories with earlier command or test failures can still be classified as successful when later evidence shows that the problem was resolved. Trajectories assessed as successful enter $\mathcal{T}^+$. Failed trajectories enter $\mathcal{T}^-$ only when they contain concrete failure evidence, sufficient code or tool context, and a clear behavior that could be corrected. Ambiguous outcomes and incomplete or low-value histories are excluded.

3.3 Step B: Outcome-Anchored Success Refinement

Algorithm 1 in Appendix A.1 summarizes the success branch using the operations and input–target pairs defined below.

Step B.1: Outcome graph and anchors. The graph construction $G = \mathcal{G}(\tau, z)$ produces an execution graph $G = (V, E, \mathcal{X})$ with turn-level nodes $v_t = (a_t, o_t)$ in V, typed edges E, and candidate outcome anchors $\mathcal{X}$. Edge predicates inspect the individual tool calls and paired results within each turn. A verified anchor combines an edit not superseded in the recorded history with a successful result from the latest relevant post-edit test, build, or lint check; $A = \mathsf{Anchor}(G)$ is the set of these anchors. For each anchor, the turns containing its surviving edits, latest available verification, and post-edit diff/status evidence jointly seed graph traversal. At least one verified anchor initializes refinement, while other anchors provide supporting context under the rules in Appendix A.

Each edge records a relation and its strength (strong, medium, or weak), covering localization, edit history, failure recovery, verification, and final inspection. Deterministic rules use tool types, normalized paths, recorded edit payloads, observations, and execution status. Localization and recovery links require matching recorded evidence, verification links use segment-local post-edit order, and edit-to-edit links require recorded content dependence. Only strong and medium edges participate in closure; Tables 5 and 6 give the anchor and edge predicates.

Step B.2: Dependency-closed projection. The closure operation $K = \mathsf{Close}_G(A)$ traces strong and medium edges backward from verified anchors A, admitting supporting anchors under the same rules until the retained turn set K stabilizes. This retains the outcome anchors together with their supporting observations and edit history. Protocol completion $\mathsf{Proto}(K)$ retains each selected turn's complete tool-call/result groups and adds substantive user instructions and required non-tool context. Projection $\rho = \mathsf{Proj}_\tau(\mathsf{Proto}(K))$ preserves source order and removes exploration and superseded edits outside this completed context. This projection operates on the recorded interaction history; retained turns keep their source indices in our notation. An LLM assesses its evidence support and contextual coherence, producing a confidence score $Q(\rho)$; projections below threshold η are rejected.

Step B.3: Single-step masked reconstruction. Let $\mathcal{C}(\rho)$ contain the source indices of candidate tool-calling assistant turns retained in ρ, and let $S \subseteq \mathcal{C}(\rho)$ denote the selected indices. For each $t \in \mathcal{C}(\rho)$, TRACE-SWE combines a role prior $w_{\varrho(t)}$, evidence features f_t, and diversity features $d(t, S)$:

$$s_0(t) = w_{\varrho(t)} + \alpha^\top f_t, \quad s_{\mathrm{loc}}(t \mid S) = s_0(t) + \beta^\top d(t, S). \tag{2}$$

Here $\varrho(t)$ is the operational role of turn t, s_0 its base score, and s_{loc} its selection score; α and β are coefficient vectors, with $\top$ denoting transpose. Evidence features cover recovery, verification, anchors, and incident dependencies; diversity rewards new operational roles and repository paths. Starting with $S_0 = \varnothing$, greedy selection chooses

$$t_j = \arg \max_{t \in \mathcal{C}(\rho) \setminus S_{j-1}} s_{\mathrm{loc}}(t \mid S_{j-1}), \quad S_j = S_{j-1} \cup \{t_j\}. \tag{3}$$

At selection round j, t_j is the chosen turn index and S_j the updated set. The operation Greedy_{m_s} in Algorithm 1 applies this rule until budget m_s is reached or candidates are exhausted, returning the final set S. Coefficients, clipped dependency features, and tie-breaking rules appear in Appendix B.

The cloze input replaces the selected assistant turn, including its text and tool calls, with a mask while retaining its actual tool results and surrounding history:

$$x_t^{\mathrm{cloze}} = (h_t, \langle \mathrm{MASK} \rangle, o_t, p_t) \longmapsto y_t = (r_t, \ell_t, m_t, a_t^{\mathrm{source}}). \tag{4}$$

The pair $(x_t^{\mathrm{cloze}}, y_t)$ is the single-step record used in Algorithm 1. Here $\langle \mathrm{MASK} \rangle$ denotes the omitted turn, h_t pre-action history, o_t the associated tool results, and p_t subsequent context; a_t^{source} denotes

the source tool calls within action a_t. The teacher provides rationale r_t, evidence-aligned labels ℓ_t, and assistant text m_t, while the target tool calls are copied from the source. Combining pre-action history h_t with posterior context (o_t, p_t) makes the relation between an action's rationale and its observed consequences an explicit reconstruction target. Retrospective explanations may use later observations, while claims about decision-time knowledge remain grounded in h_t.

Step B.4: Multi-step masked reconstruction. The multi-step view extends this construction to a source-ordered window $C = (a_0, o_0, \ldots, a_k, o_k)$, with locally indexed turns, retaining (a_0, o_0) as context and masking the k subsequent assistant decisions. The candidate set $\mathcal{W}(\rho)$ contains windows with $k \geq 2$, a recorded preceding observation for each target, no task-changing interruption, and a terminal successful verification or edit with a successful tool result. A successful edit result indicates completion of the tool operation, whereas verification provides evidence about the repair outcome.

Windows are ranked by

$$s_{\text{chain}}(C) = \lambda_k k + b_{\text{t}}(C) + \sum_{i=1}^{k} s_0(t_i) + \lambda_{\text{f}} I_{\text{fail}}(C). \tag{5}$$

Here i indexes the k target decisions and t_i is the source-turn index of local action a_i, so $s_0(t_i)$ uses the same base score as Step B.3. The term b_{t} rewards terminal evidence, I_{fail} indicates preceding failure, and λ_k and λ_{f} weight length and failure evidence. The operation $\mathsf{Rank}_{m_{\text{m}}}$ selects a set $W \subseteq \mathcal{W}(\rho)$ of up to m_{m} windows by s_{chain}, applying the overlap filtering in Appendix B, which also specifies length limits, coefficients, and budgets. With h_C ending at (a_0, o_0), p_C following o_k, and $\mu_C = (\langle \text{MASK} \rangle_1, o_1, \ldots, \langle \text{MASK} \rangle_k, o_k)$, the record is constructed as

$$x_C^{\text{cloze}} = (h_C, \mu_C, p_C) \longmapsto y_C = \big((r_i, \ell_i, m_i, a_i^{\text{source}})\big)_{i=1}^{k}. \tag{6}$$

Each o_i is posterior evidence for a_i and, for $i < k$, preceding evidence for a_{i+1}. The pair $(x_C^{\text{cloze}}, y_C)$ is the multi-step record used in Algorithm 1, with the same rationale, label, text, and source-call fields as the single-step record. Preceding context establishes the subgoal, intervening observations connect successive decisions, and subsequent context reveals their collective relevance to later progress; reconstruction preserves the decision-time boundary for each action. For both reconstruction views, $\mathsf{Validate}$ retains records that satisfy source order, mask–target alignment, source-exact calls, label requirements, and assistant–tool protocol consistency relative to ρ.

3.4 Step C: Failure-Evidence-Guided Correction

For a trajectory $\tau \in \mathcal{T}^-$, Step C collects observed symptoms, repository context, attempted edits, and user corrections as evidence E^-. The teacher annotation procedure Ψ produces a correction target $c^- = \Psi(\tau, E^-)$ subject to the annotation constraints

$$\begin{aligned} E^- \models \Gamma_o(c^-), \qquad & \Gamma_d(c^-) \subseteq \mathcal{H}(E^-), \\ \chi(q_c) = \chi(v_c) = 0, \qquad & \kappa_c \in [0,1], \quad v_c \neq \varnothing. \end{aligned} \tag{7}$$

Here Γ_o extracts factual claims, which the record must support ($\models$), while Γ_d extracts diagnostic hypotheses restricted to the evidence-compatible set $\mathcal{H}(E^-)$; κ_c is the reported confidence. The execution indicator χ marks proposed repair q_c and verification v_c as unexecuted, while $v_c \neq \varnothing$ requires a verification strategy. These semantic constraints guide annotation and review of factual support, diagnostic uncertainty, and execution status. Automatic checks cover required fields, confidence range, and execution claims; accepted records pair the failed history with the correction response, including an optional unexecuted patch sketch.

3.5 Supervision Outputs

The final corpus comprises three datasets: single-step masked reconstruction D_{single}, multi-step masked reconstruction D_{multi}, and evidence-bounded failure correction $D_{\text{correction}}$. Each dataset pools the corresponding per-trajectory record sets $\mathcal{Y}_\tau$ across accepted source trajectories. All retained records pass automatic validation, and the resulting corpus undergoes the sampled human audit described in Section 3.1. The dependency-closed projection serves as intermediate context for the two successful-trajectory views. The three datasets are combined for SFT with equal sampling weights.

Table 2: **Results on LING-3.0-FLASH (%).**

Benchmark	Baseline		TRACE-SWE (ours)		Δ (pp)	
	NOTHINK	THINK	NOTHINK	THINK	NOTHINK	THINK
OpenHands						
SWE-bench Verified	67.80	63.80	72.00	70.20	**+4.20**	**+6.40**
SWE-bench Pro	41.30	38.00	44.20	43.62	**+2.90**	**+5.62**
SWE-bench Multilingual	56.00	49.30	63.70	58.40	**+7.70**	**+9.10**
Claude Code						
SWE-bench Verified	55.00	58.20	59.70	66.60	**+4.70**	**+8.40**
SWE-bench Pro	32.83	37.21	35.57	40.40	**+2.74**	**+3.19**
SWE-bench Multilingual	55.00	51.33	62.17	58.80	**+7.17**	**+7.47**
OpenHands						
Terminal-Bench 2.0	45.84	40.67	46.80	44.80	**+0.96**	**+4.13**
Terminal-Bench 2.1	46.44	42.32	48.50	45.30	**+2.06**	**+2.98**

4 EXPERIMENTS

We evaluate TRACE-SWE through benchmark comparisons, cross-model evaluation, supervision component ablations, and interaction-efficiency analysis.

4.1 EXPERIMENTAL SETTING

Trajectory data. We use real-user Claude Code trajectories covering everyday tasks and code writing. From 45,364 raw trajectories, Step A retains 22,843 successful and 11,496 failed histories. The final corpus contains 5,825 single-step reconstruction, 4,350 multi-step reconstruction, and 9,773 failure-correction records, totaling 19,948. All records pass automatic validation, and the corpus undergoes human spot checks across the three views (Appendix A.2).

SFT Setting. Baseline is the reference checkpoint without additional trajectory-based SFT. Each model is trained for three epochs with shared hyperparameters (Table 8). TRACE-SWE samples the three supervision views equally.

Models and scaffolds. We evaluate LING-3.0-FLASH with OpenHands (Wang et al., 2025b) and Claude Code under both THINK and NOTHINK inference. LING-TINY-3.0, QWEN3.5-4B, and QWEN3.5-27B use OpenHands under THINK only.

Benchmarks and metrics. We evaluate issue resolution on SWE-bench Verified (Jimenez et al., 2024), Pro (Deng et al., 2025), and Multilingual (Khandpur et al., 2025). Terminal-Bench 2.0 (Merrill et al., 2026) and 2.1 (The Terminal-Bench Team, 2026) assess transfer to terminal-based tasks for LING-3.0-FLASH. Scores are percentages; gains Δ over Baseline are percentage points (pp). Benchmark and ablation scores are means over three inference runs using the same checkpoint.

4.2 LING-3.0-FLASH: COMPLETE SFT EVALUATION

Consistent improvements on SWE-bench. TRACE-SWE improves all six SWE-bench configurations, averaging $+4.90$ and $+6.70$ points under NOTHINK and THINK, respectively (Table 2). Gains under OpenHands support transfer from Claude Code training histories to another interaction scaffold. TRACE-SWE improves performance under both inference modes.

Transfer to terminal-based tasks. Terminal-Bench gains average $+1.51$ and $+3.56$ points under NOTHINK and THINK, with improvements on both benchmark versions. These results extend the observed benefits beyond issue resolution to terminal-based tasks. Command-line exploration, editing, and verification are shared actions across these settings, providing a plausible basis for this transfer.

4.3 CROSS-MODEL SWE EVALUATION

Table 3: **Cross-model SFT results under THINK (%).** All rows use OpenHands.

	SWE-bench Verified			SWE-bench Pro			SWE-bench Multilingual		
Model	**Baseline**	**TRACE-SWE**	**Δ**	**Baseline**	**TRACE-SWE**	**Δ**	**Baseline**	**TRACE-SWE**	**Δ**
Ling3.0 family									
Ling-3.0-tiny	35.00	43.20	**+8.20**	27.00	34.80	**+7.80**	27.97	38.00	**+10.03**
Ling-3.0-flash	63.80	70.20	**+6.40**	38.00	43.62	**+5.62**	49.30	58.40	**+9.10**
Qwen3.5 family									
Qwen3.5-4B	42.80	48.60	**+5.80**	26.70	31.33	**+4.63**	42.67	52.40	**+9.73**
Qwen3.5-27B	56.20	62.40	**+6.20**	34.04	37.30	**+3.26**	45.22	52.62	**+7.40**

Gains across model families and scales. All additional models improve on every benchmark, with mean gains of $+8.68$, $+6.72$, and $+5.62$ points for LING-TINY-3.0, QWEN3.5-4B, and QWEN3.5-27B, respectively (Table 3). Both Qwen model sizes benefit, demonstrating improvements across different scales and starting performance levels. The consistent direction across the Ling and Qwen families supports the broader applicability of TRACE-SWE supervision.

4.4 SFT COMPONENT ABLATIONS

Supervision configurations. Selected-action-only supervision (SA-only) retains the complete trajectory as the training sequence and computes loss only on selected assistant actions. Raw-rollout uses the original trajectories, and dependency-closed projection SFT (DC-proj.) uses the refined histories after Step B.2. Both supervise all assistant responses throughout their respective trajectories. TRACE-SWE combines the two reconstruction views (D_{single},D_{multi}) and failure correction ($D_{\text{correction}}$) with equal sampling weights.

Table 4: **Component ablations on LING-3.0-FLASH under THINK.** Parentheses show gains over Baseline (pp); Mean equally weights the eight configurations. Scores are percentages; gains are calculated from the displayed scores. Multi. denotes SWE-bench Multilingual.

	OpenHands			Claude Code			Terminal-Bench		
SFT condition	**Verified**	**Pro**	**Multi.**	**Verified**	**Pro**	**Multi.**	**2.0**	**2.1**	**Mean**
Baseline	63.80	38.00	49.30	58.20	37.21	51.33	40.67	42.32	47.60
Comparison supervision methods									
SA-only	64.07 (+0.27)	40.15 (+2.15)	50.17 (+0.87)	58.20 (+0.00)	35.29 (-1.92)	52.67 (+1.34)	39.13 (-1.54)	40.08 (-2.24)	47.47 (-0.13)
Raw-rollout	64.07 (+0.27)	39.40 (+1.40)	49.80 (+0.50)	58.70 (+0.50)	36.20 (-1.01)	52.90 (+1.57)	40.20 (-0.47)	42.10 (-0.22)	47.92 (+0.32)
DC-proj.	67.20 (+3.40)	38.95 (+0.95)	52.33 (+3.03)	60.60 (+2.40)	34.34 (-2.87)	51.67 (+0.34)	41.57 (+0.90)	42.70 (+0.38)	48.67 (+1.07)
TRACE-SWE supervision views									
D_{single}	66.00 (+2.20)	41.79 (+3.79)	58.00 (+8.70)	64.30 (+6.10)	38.44 (+1.23)	57.33 (+6.00)	43.82 (+3.15)	42.32 (+0.00)	51.50 (+3.90)
D_{multi}	66.00 (+2.20)	39.61 (+1.61)	54.87 (+5.57)	65.00 (+6.80)	38.58 (+1.37)	55.33 (+4.00)	41.57 (+0.90)	43.80 (+1.48)	50.60 (+3.00)
$D_{\text{correction}}$	67.00 (+3.20)	39.67 (+1.67)	50.67 (+1.37)	62.80 (+4.60)	37.76 (+0.55)	52.67 (+1.34)	39.33 (-1.34)	42.70 (+0.38)	49.08 (+1.48)
TRACE-SWE	**70.20 (+6.40)**	**43.62 (+5.62)**	**58.40 (+9.10)**	**66.60 (+8.40)**	**40.40 (+3.19)**	**58.80 (+7.47)**	**44.80 (+4.13)**	**45.30 (+2.98)**	**53.52 (+5.92)**

Training value of context-preserving refinement. SA-only and raw-rollout provide little overall benefit, with mean changes of -0.13 and $+0.32$ points across eight configurations (Table 4). Dependency-closed projection yields a larger mean gain of $+1.07$ points. This pattern is consistent with refinement removing distracting exploration while preserving the observations and edit history needed to interpret useful actions.

Utility of the three supervision views. Single-step reconstruction, multi-step reconstruction, and failure correction yield mean gains of $+3.90$, $+3.00$, and $+1.48$ points, respectively. Among individual views, single-step reconstruction has the largest average gain, while multi-step reconstruction leads on Claude Code. TRACE-SWE reaches a mean score of 53.52%, exceeding the strongest individual view (D_{single}) by 2.02 points. These comparisons support the effectiveness of trajectory refinement and all three supervision constructions. Their joint use yields the highest average score among the tested configurations.

4.5 Interaction Efficiency

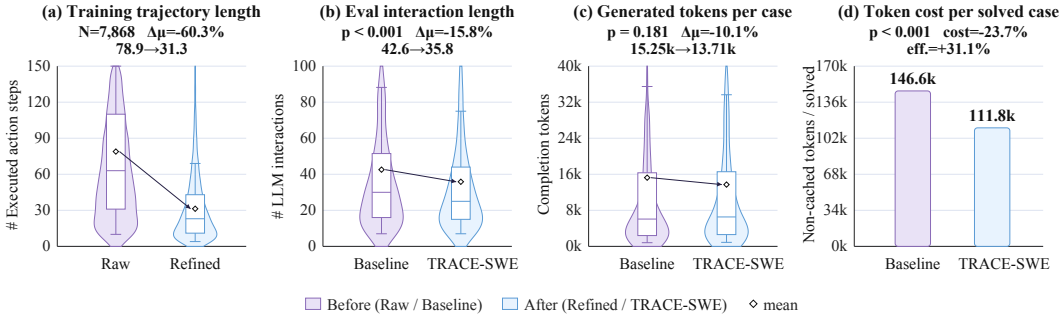


Figure 2: **Trajectory refinement and inference efficiency.** (a) Training refinement; (b)–(d) evaluation on SWE-bench Verified. Diamonds mark means; violins are clipped at the 99th percentile.

Shorter trajectories and lower evaluation cost. Refinement reduces mean training actions by 60.3% (Figure 2). At evaluation time, mean interaction length decreases by 15.8%, and non-cached token cost per solved case decreases by 23.7% (reported $p < 0.001$ for both). Equivalently, solved cases per non-cached token increase by 31.1%, including the cost of failed attempts. Together with the benchmark gains, these results show that TRACE-SWE improves task performance while reducing interaction costs in the evaluated setting. This combination supports its practical value for code agents that must balance task success with inference cost.

4.6 Limitations and Future Work

While TRACE-SWE improves performance and efficiency in the evaluated settings, two directions warrant further study. First, evaluation across more models, scaffolds, and user workflows would clarify its generality. Second, supervision quality depends on source records and automated assessment; stronger evidence checks and human review could improve robustness. These extensions would support broader use in practical software development.

5 Conclusion

We presented TRACE-SWE, a framework that refines real-user code agent trajectories into action-oriented supervision using recorded execution evidence. It combines single-step and multi-step cloze-style reconstruction from refined successful trajectories with evidence-bounded failure correction for SFT. Experiments show improvements across four open-weight models on SWE-bench, with gains on Ling-3.0-flash extending to Terminal-Bench. TRACE-SWE also reduces interaction length and non-cached token cost per solved case. These findings highlight real-user trajectories as a valuable source of supervision for improving code agent capability and efficiency.

AI use statement

Generative AI tools, including Codex, assisted with code development, manuscript drafting, and LaTeX formatting. The authors reviewed and revised the assisted content and take responsibility for the manuscript and its findings. The use of LLMs for trajectory assessment and supervision construction is described separately in Section 3.

Ethics statement

This work aims to improve the capability and efficiency of code agents by learning from real-user interactions. The trajectories used in this study have undergone privacy-protection processing to safeguard user information and mitigate the risk of personal information disclosure. Responsible reuse also requires respect for code ownership, and any public release requires an additional review for sensitive content.

Reproducibility statement

To support reproducibility, we document the supervision construction procedure and experimental settings in Sections 3 and 4.1. Appendices A and B provide the anchor and dependency rules, success-branch algorithm, and action-selection weights. Appendix C reports the training configuration in Table 8. Appendix D includes the routing and data construction templates, specifying their inputs and output requirements.

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. URL https://arxiv.org/abs/2606.16038.

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

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding agent interactions from real users in the wild. *arXiv preprint arXiv:2604.20779*, 2026. URL https://arxiv.org/abs/2604.20779.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. SWE-universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*, 2026. URL https://arxiv.org/abs/2602.02361.

Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. Copilot arena: A platform for code LLM evaluation in the wild. *arXiv preprint arXiv:2502.09328*, 2025. URL https://arxiv.org/abs/2502.09328.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2Run: Automated building executable environment for code repository at scale. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-1096. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/2f2b1d6bbd50865eca40e2774a057eef-Abstract-Conference.html.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025. URL https://arxiv.org/abs/2504.07164.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. SWE-bench multilingual. SWE-bench official benchmark announcement, 2025. URL https://www.swebench.com/multilingual.html.

Nikolai Ludwig, Wasi Uddin Ahmad, Somshubra Majumdar, and Boris Ginsburg. From SWE-zero to SWE-hero: Execution-free to execution-based fine-tuning for software engineering agents. *arXiv preprint arXiv:2604.01496*, 2026. URL https://arxiv.org/abs/2604.01496.

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. URL https://arxiv.org/abs/2605.21996.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007. URL https://people.csail.mit.edu/cpacheco/publications/feedback-random-abstract.html.

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 *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Xiao Yu, Rui Yang, Tao Ge, Alessandro Sordoni, Xingdi Yuan, Yelong Shen, Pengcheng He, Tong Zhang, Zhou Yu, and Jianfeng Gao. Orchard: An open-source agentic modeling framework. *arXiv preprint arXiv:2605.15040*, 2026. URL https://arxiv.org/abs/2605.15040.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024. doi: 10.1109/TSE.2023.3334955. URL https://ieeexplore.ieee.org/document/10329992.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Yifan Song, Weimin Xiong, Xiutian Zhao, Dawei Zhu, Wenhao Wu, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. AgentBank: Towards generalized LLM agents via fine-tuning on 50000+ interaction trajectories. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 2124–2141, 2024a. doi: 10.18653/v1/2024.findings-emnlp.116. URL https://aclanthology.org/2024.findings-emnlp.116/.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7584–7600, 2024b. doi: 10.18653/v1/2024.acl-long.409. URL https://aclanthology.org/2024.acl-long.409/.

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.

The Terminal-Bench Team. Terminal-bench 2.1. Official benchmark release announcement, 2026. URL https://www.tbench.ai/news/terminal-bench-2-1.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3742–3761, 2025a. doi: 10.18653/v1/2025.findings-acl.193. URL https://aclanthology.org/2025.findings-acl.193/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel 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*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rosé. RepoST: Scalable repository-level coding environment construction with sandbox testing. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/pdf/0f0642854eb629ef6d848738749e7fc2ffc1b800.pdf.

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. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024. URL https://arxiv.org/abs/2404.05427.

Yangtian Zi, Zixuan Wu, Aleksander Boruch-Gruszecki, Jonathan Bell, and Arjun Guha. Agentpack: A dataset of code changes, co-authored by agents and humans. *arXiv preprint arXiv:2509.21891*, 2025. URL https://arxiv.org/abs/2509.21891.

A Detailed Outcome-Anchor and Dependency Rules

This appendix specifies the anchor predicates and dependency rules used in Steps B.1–B.2, operating on recorded assistant turns and their tool-call/result groups.

Task segments. A segment g is a contiguous interaction interval defined by the source task grouping, spanning the earliest user request and latest response assigned to that group. When this metadata is unavailable, each substantive user message starts a segment ending before the next such message; empty or system-noise messages do not create boundaries. If no such boundary exists, the history forms one segment.

Surviving and superseded edits. A successful edit with an identified path survives unless a later successful edit in the same segment and file supersedes it. Supersession is detected by a complete-file write or by a replacement whose removed text contains the earlier added text and whose new text no longer contains it.

Relevant verification. Verification commands are recognized as test, build, or lint operations from their command text. For anchor scoring, the latest relevant verification is the last such operation in the same segment after the final surviving edit. Its outcome is assigned by case-insensitive matching of recorded tool output: failure markers take precedence, a present result without a failure marker is treated as successful, and a missing result is marked missing. Failure markers include `FAILED`, `FAILURE(S)`, Python tracebacks, `npm ERR!`, tool-use errors, permission or file-access errors, command-not-found messages, nonzero exit-code text, and `error:` followed by a nonzero value or error description. Edit-to-verification edges connect surviving edits to subsequent verification operations within the segment. These edges encode segment-local execution order; test coverage of individual edits remains unmeasured.

For a segment g containing code edits, $S(g)$ indicates at least one surviving successful edit, and $V^+(g)$ and $V^-(g)$ indicate success and failure of the latest relevant post-edit verification, respectively. The Boolean predicates $D(g)$ and $C(g)$ indicate final-state inspection and recorded completion evidence, respectively. Segments with any missing edit-tool result are assigned level L; for the remaining segments, the evidence level $\zeta(g)$ is defined as

$$\zeta(g) = \begin{cases} \text{H}, & S(g) \wedge V^+(g), \\ \text{M}, & S(g) \wedge \neg V^+(g) \wedge (D(g) \vee C(g)) \wedge \neg V^-(g), \\ \text{L}, & \text{otherwise.} \end{cases} \tag{8}$$

Table 5: Outcome-anchor levels and their use in trajectory refinement.

Level	Use
High	Required to initialize the main dependency closure.
Medium	Supplies supporting outcome context when a high anchor exists.
Low	Retained only when its strong/medium closure connects to the main closure.

For turn-level nodes $v_i \prec v_j$, predicates compare individual tool calls and their paired observations; the order relation requires turn i to precede turn j. For each compared call, π_i denotes its normalized path, q_i its search query, (Δ_i^-, Δ_i^+) its removed and added edit text, and I_i the identifiers in its edit payload; $\text{text}(o_i)$ denotes the paired observation text. An edge is added when a call pair satisfies the corresponding predicate in Table 6. Edges are constructed within a segment, except for cross-segment recovery edges $E_\times$. Here $x \sqsubseteq y$ denotes case-insensitive literal occurrence, $\text{base}(\pi)$ extracts a filename, and $\text{Tok}(x)$ returns the lexical tokens in observation text x.

Table 6: Deterministic construction rules for execution-graph edges.

Edge	Trigger
Localization (E_{sr}, E_{re})	Search→read if $\pi_j \sqsubseteq \text{text}(o_i)$; read→edit if $\pi_i = \pi_j$.
Recovery (E_{fs}, E_{fe})	Failure→search if $q_j \sqsubseteq \text{text}(o_i)$; failure→edit if $\text{base}(\pi_j) \sqsubseteq \text{text}(o_i)$ or $I_j \cap \text{Tok}(\text{text}(o_i)) \neq \varnothing$.
Edit history (E_{ee})	Edit→edit if $\pi_i = \pi_j$ and $\Delta_i^+ \sqsubseteq \Delta_j^-$.
Validation (E_{ev}, E_{ei})	Surviving edit→verify for a later test, build, or lint in the same segment; edit→inspect if $\pi_i \sqsubseteq \text{text}(o_j)$ for a later diff/status observation.
Terminal evidence (E_{vi})	Verify→inspect links the latest successful verification to the subsequent terminal diff/status observation.
Cross-segment recovery $(E_{\times})$	Links a search or edit to the most recent matching failed verification in an earlier segment, whose output names the later filename or identifier; the edge points forward in source order.

The strong and medium edge sets used for traversal are $E_{\text{S}} = E_{sr} \cup E_{fs} \cup E_{re} \cup E_{ev} \cup E_{ei}$ and $E_{\text{M}} = E_{fe} \cup E_{ee} \cup E_{vi} \cup E_{\times}$, respectively. Basename-only localization matches and links from superseded edits form E_{W}, which is stored as metadata but excluded from closure. In Algorithm 1, $\mathsf{Close}_G(A)$ first computes the strong/medium predecessor closure of the high anchors A, then adds medium anchors and their predecessors. Low anchors and their predecessors are included only when their strong/medium closure intersects the main closure; traversal continues until no further nodes are added. Edges with the same ordered node pair and relation are deduplicated, and projection preserves the original message and tool-call order. Protocol completion retains complete tool-call/result groups within selected turns, substantive user instructions, and required non-tool transitions.

A.1 Success-Branch Algorithm

Algorithm 1 summarizes Step B using the notation defined in Section 3.3. Rejection $\perp$ yields no reconstruction records for the trajectory.

Algorithm 1 Step B: outcome-anchored refinement and reconstruction

Require: Trajectory $\tau \in \mathcal{T}^+$ and evidence summary z
Ensure: Records $(\mathcal{Y}_\tau^{\text{single}}, \mathcal{Y}_\tau^{\text{multi}})$, or rejection $\perp$

Steps B.1–B.2: Select and project evidence
$G \leftarrow \mathcal{G}(\tau, z)$; $A \leftarrow \mathsf{Anchor}(G)$
if $A = \varnothing$ **then**
  return $\perp$
end if
$K \leftarrow \mathsf{Close}_G(A)$
$\rho \leftarrow \mathsf{Proj}_\tau(\mathsf{Proto}(K))$
if $Q(\rho) < \eta$ **then**
  return $\perp$
end if
Step B.3: Single-step reconstruction
$S \leftarrow \mathsf{Greedy}_{m_{\text{s}}}(\mathcal{C}(\rho), s_{\text{loc}})$
$\mathcal{Y}_\tau^{\text{single}} \leftarrow \{(x_t^{\text{cloze}}, y_t) : t \in S\}$ ▷ Eq. 4
Step B.4: Multi-step reconstruction
$W \leftarrow \mathsf{Rank}_{m_{\text{m}}}(\mathcal{W}(\rho), s_{\text{chain}})$
$\mathcal{Y}_\tau^{\text{multi}} \leftarrow \{(x_C^{\text{cloze}}, y_C) : C \in W\}$ ▷ Eq. 6
return $\mathsf{Validate}(\mathcal{Y}_\tau^{\text{single}}, \mathcal{Y}_\tau^{\text{multi}} \mid \rho)$

A.2 Human Audit

Automatic validation checks every retained record, while human review samples records from the three supervision views. Reviewers compare the source history, retained context, and generated target to assess outcome routing, missing action context or unrelated verification links, temporal attribution,

and diagnostic support. For reconstruction, later observations may explain an action's consequences but must not be presented as information available before that action. For correction, factual claims must agree with the record, diagnostic hypotheses must be distinguished from observations, and proposed edits and tests must remain explicitly unexecuted.

B Selection Weights for Steps B.3 and B.4

The selection scores in Equations 2 and 5 use fixed, deterministic ranking heuristics. Table 7 specifies the local selection weights. The role prior gives implementation the highest base score, while evidence and diversity terms adjust the ranking. For Step B.3, $g_{\mathrm{S}}(n) = \min(24, 8n)$ and $g_{\mathrm{M}}(n) = \min(12, 4n)$. Here n counts incident edges of the corresponding strength at the candidate turn, after edge deduplication. In Equation 2, f_t contains the recovery, verification, anchor, and dependency-keep indicators, followed by the clipped strong- and medium-edge terms g_{S} and g_{M}. The indicator coefficients are listed in Table 7; the two clipped terms enter with unit coefficients, while $d(t, S)$ contains new-role and new-path indicators weighted by β_{r} and β_{p}. The local selection budget is $m_{\mathrm{s}} = 3$. Ties in local selection are resolved first by incident-edge count and then by earlier source order.

Table 7: Local-action selection weights for Step B.3. Role priors and evidence/diversity terms are independent lists aligned for compact presentation.

Role prior w	**Value**	**Evidence/diversity term**	**Value**
Implementation	60	Explicit recovery, α_{r}	28
Verification	55	Successful verification, α_{+}	25
Evidence acquisition	50	Failed verification, α_{-}	12
Localization	46	Outcome anchor, α_{a}	12
Failure diagnosis	42	Dependency-keep marker, α_{d}	3
Diagnostic shell	37	New-role bonus, β_{r}	18
Patch inspection	32	New-path bonus, β_{p}	6
Other	15	Strong/medium edge functions	$g_{\mathrm{S}}, g_{\mathrm{M}}$

The Step B.4 window score uses $\lambda_k = 100$ and $\lambda_{\mathrm{f}} = 30$. The terminal term $b_{\mathrm{t}}(C)$ is 80 when the window ends in successful verification and 40 when it ends in an implementation action with an observed successful tool result. Candidates are compared with already selected windows and filtered when the Jaccard similarity of their target-turn sets exceeds 0.8. The failure-recovery exception applies when at least one turn before the terminal action in the candidate window has a recorded failure result; these candidates remain eligible despite the overlap. The multi-step range is $k_{\min} = 2$ through $k_{\max} = 6$, and the per-trajectory window budget is $m_{\mathrm{m}} = 3$.

C Hyperparameters

Table 8 reports the hyperparameters used for all trajectory-based SFT conditions in Section 4.1, each trained for three epochs.

Table 8: Key hyperparameters for supervised fine-tuning.

Parameter	Value
Global batch size	128
Maximum sequence length	131,072
Training epochs	3
Optimizer	Muon
Peak learning rate	5.5×10^{-5}
Warmup iterations	50
Learning-rate scheduler	Cosine
Weight decay	0.1
Gradient clipping	1.0
Precision	BF16
SFT random seed	42

D Construction and SFT Prompts

D.1 Step A Execution-Grounded Routing

The judge distinguishes successful trajectories, informative failures, and exclusions according to Section 3.2. Its input summarizes the user requests and recorded signals, together with source-indexed excerpts covering the beginning and end of the interaction, edits, tests, and failure-related context. The evidence summary is used for judging; the original trajectory remains unchanged. The output records the outcome, confidence, task completion, unresolved failures, correction value, context completeness, and an evidence-grounded rationale.

Step A Routing system prompt

```
You are a strict trajectory-outcome judge for coding-agent data.
The input has already passed a deterministic gate: code-related topic, a real edit-tool
      call, and a code-file extension.
Judge whether the trajectory completed the user's task, is a diagnostically valuable
     failure, or must be excluded.

Return exactly one JSON object with these keys:
{
  "classification": "success" | "valuable_failure" | "exclude",
  "confidence": 0.0,
  "task_completed": true | false,
  "unresolved_failure": true | false,
  "failure_evidence": ["short evidence tags"],
  "has_clear_correction_point": true | false,
  "trajectory_complete_enough": true | false,
  "rationale": "concise evidence-grounded explanation"
}

Definitions:
- success: the requested code change appears completed; later evidence resolves earlier
      errors; final state is consistent with success.
- valuable_failure: the task remains incomplete or incorrect AND the trace has concrete
      failure evidence, enough code/tool context, and a clear behavior that could be
     corrected.
- exclude: ambiguous outcome, too short/incomplete, low-value, no clear correction
     point, or insufficient evidence.

Do not classify a trace as failure merely because an early command/test failed if later
      actions fixed it.
Do not infer success from a confident final sentence alone when tool evidence
     contradicts it.
Return JSON only.
```

Step A Routing user prompt

```
Judge this admitted code-edit trajectory.
Earlier failures may be resolved later.
Use the final observed state, concrete tool evidence, context completeness, and
     correction value.

{trajectory_evidence_json}
```

D.2 STEP B.3 SINGLE-STEP MASKED RECONSTRUCTION

The teacher pair annotates the source action, while the SFT pair defines the reconstruction input used for training. The teacher input hides the direct tool results along with the action, whereas the final SFT input retains those results as posterior context.

Step 2.3 Construction system prompt

```
You are the high-quality decision reasoning module for a coding agent. You will be
     given a complete coding trajectory where one assistant action and its immediate
     tool result are replaced by <MASK>. You must output the justification for why this
      real decision is appropriate for this situation.

The correct answer for this masked decision is the following tool calls:

'''json
{ground_truth}
'''

The original assistant content for this masked decision is:
{ground_truth_assistant_content}

The target action category is: {action_role}
For this category, output exactly these evidence-aligned process labels and no others:
{required_labels}

Use the full trajectory as hindsight supervision to understand why this action was
     useful, but write reasoning as a decision made at the masked point. Later messages
      may confirm the action's value; they must not be presented as facts already known
      before the action.
Explicitly consider the logic and necessity of the tool calls immediately before and
     after the masked action. Explain why the masked call is the necessary transition
     between them, while avoiding future-information leakage.

CRITICAL RULES FOR YOUR THINKING PROCESS:
- DO NOT use phrases like "the answer is", "the correct answer", "the provided answer",
      "as shown above", "the reference shows"
- Instead, genuinely analyze the conversation: what happened, what the user needs, what'
     s the logical next step
- Your thinking should read as if you discovered the answer through analysis alone

Think about:
1. What has happened so far in the conversation?
2. What is the user trying to accomplish?
3. What information is available / missing?
4. What is the logical next step?
5. Which tool(s) and arguments would accomplish that step?

Return one compact JSON object and nothing else. Do not wrap it in markdown fences.
     Escape newlines inside JSON strings.
Length budget: reasoning_content >=3000 characters and assistant_content >=1500
     characters.

The JSON schema is:
{output_schema}

Strict rules for output:
- tool_calls must exactly equal the supplied tool calls, with name and arguments only.
- labels must contain exactly the required keys shown above. Do not emit inapplicable
     labels.
- Never mention or acknowledge that the answer was supplied.
- reasoning_content must not claim that a later tool result was already observed.
- assistant_content must read like a normal coding agent at the masked turn.
- Every emitted label must be concrete, concise, and grounded in the trajectory.
```

Step B.3 Construction user prompt

```
# Masked decision reconstruction

The conversation below is a real coding-agent trajectory. One valuable assistant
     decision is replaced by <MASK>. The mask covers the assistant's reasoning, normal
     response text, and tool calls at that turn. Tool-result messages produced directly
      by the masked calls are removed with it.

Reconstruct the missing decision as part of the trajectory's causal chain:
- Determine the repository state, evidence, uncertainty, and unresolved subgoal
     immediately before <MASK>.
- Consider the logic and necessity of the tool calls before and after <MASK>: why the
     missing action is needed between them, and why a plausible alternative would be
     weaker.
- Use later messages only as hindsight outcome evidence. Do not claim that later
     observations were already known when the masked decision was made.
- Attribute later outcomes to the action that produced them and explain the causal link,
      rather than merely repeating that the final trajectory succeeded.
- Reconstruct only the missing assistant decision. Do not invent or output environment
     tool results.

The system message defines the response schema, applicable labels, and exact tool-call
     constraints. Follow those system instructions.

## Conversation Log

{masked_conversation}
```

Step B.3 SFT system prompt

```
You are the decision-reasoning module of a coding agent.

You receive a complete coding trajectory with one assistant decision replaced by <MASK>.
      The real tool-result messages produced by the hidden decision remain immediately
     after <MASK> as posterior execution evidence. Reconstruct the missing assistant
     turn as a decision made before those results were observed.

Use the trajectory before the mask to identify the current state and unresolved subgoal.
      Later trajectory messages may be used as outcome evidence when learning why the
     decision was valuable. When using such evidence, identify which earlier action
     produced that outcome and explain why that particular action made the outcome
     possible. Do not describe later observations as if they were already known at
     decision time.

Consider the logic and necessity of the tool calls immediately before and after the
     masked action. Explain why the missing call is the necessary transition, without
     treating later results as already known.

Produce a concise, evidence-grounded reasoning process, a normal assistant message, and
      the minimal real tool call needed at the masked point.
```

Step B.3 SFT user prompt

```
# Masked decision reconstruction

The following is a real coding-agent trajectory. `<MASK>` replaces one valuable
     assistant turn, including its reasoning, response content, and tool calls. The
     real tool-result messages produced by that hidden turn remain immediately after `<
     MASK>` as posterior execution evidence. Reconstruct the missing turn using the
     preceding context and subsequent outcomes, while preserving the trajectory's
     causal and tool-use logic. Treat the retained tool results as outcomes of the
     missing action, not as information available before choosing it.

## Conversation Log

{masked_conversation_with_real_tool_results}
```

The teacher response supplies the rewritten reasoning and assistant content; the final SFT assistant message copies the source tool call exactly rather than using a teacher-generated payload.

D.3 STEP B.4 MULTI-STEP MASKED RECONSTRUCTION

Teacher annotation is performed separately for each target turn, while the final SFT task reconstructs all masked turns in source order.

Step B.4 Construction system prompt, per target step

```
You are the high-quality chained-decision reasoning module for a coding agent. You will
     be given a complete coding trajectory where several assistant actions are
     replaced by <MASK>. You must output the justification for why this real decision
     is appropriate for this situation.

This API request reconstructs only original target step {target_step_ordinal} of {
     total_target_steps}, marked <MASK_STEP_{target_step_ordinal}>. Ground-truth
     actions and assistant-content references for all target steps are supplied so you
     can judge the complete chain logic. Return only original target step {
     target_step_ordinal}; do not emit the other masked steps.

The trajectory is a real compressed execution path. Each target action must be
     reconstructed from the real observation produced by the preceding action. The
     exact tool calls and optional original assistant-content references for every
     target step are:

{ground_truth_steps}

The supplied tool calls are the action ground truth and must be copied exactly. The
     original assistant content is contextual reference only, not a ground-truth answer:
     it may be incomplete, low quality, or empty. Use it only to help understand the
     surrounding action and writing style. Independently generate a concise, non-empty
     assistant_content for every step, including when the original reference is empty.
     Never mention that any supervision was supplied.

Before composing the JSON, analyze each decision independently as if you discovered the
     action from the trajectory:
1. What has happened so far before this step?
2. What the user is trying to accomplish at this point?
3. What information is available or missing after the preceding observation?
4. What is the logical next action, and which tool call advances the unresolved subgoal
     ?
5. Why is this action necessary between the immediately preceding and following tool
     calls, and why would a plausible alternative be weaker?

For every step:
- reason from information available before that action;
- explain how the preceding real observation changed the state, uncertainty, or subgoal
     ;
- consider the logic and necessity of the tool calls immediately before and after it;
- use later path results only as outcome evidence, identify which earlier action
     produced that result, and explain why that action made the result possible;
- emit exactly the dynamic evidence-aligned label keys listed for that step, using the
     spaced names shown;
- copy the supplied tool call name and arguments exactly.

Return one compact JSON object and nothing else. Do not use markdown fences or XML tags.
     Escape newlines inside JSON strings.
Length budget for every step: reasoning_content >=3000 characters and assistant_content
     >=1500 characters.

The exact JSON schema for this sample is:
{output_schema}

Strict rules:
- The top-level object contains only steps.
- Every step contains only reasoning_content, assistant_content, labels, and tool_calls.

- Every assistant_content must be a non-empty normal coding-agent message, even when
     original_assistant_content_reference is empty.
- The response must contain exactly one step corresponding to original target step {
     target_step_ordinal}.
- Do not invent tool results or describe a later result as known before its action.
- Every label must be concrete and grounded in a real file, function, test, error, diff,
     command output, or observation.
```

Step B.4 Construction user prompt, per target step

```
# Chained masked-decision reconstruction

The conversation below is a complete real coding-agent trajectory. Several valuable
     assistant decisions in one continuous execution chain are replaced by ordered `<
     MASK_STEP_k>` markers. Each marker hides that assistant turn's reasoning, normal
     response text, and tool calls.

Reconstruct only <MASK_STEP_{target_step_ordinal}> as its causal observation-to-action
     transition:
- For this step, determine the repository state, evidence, uncertainty, and unresolved
     subgoal immediately before its marker.
- Consider the logic and necessity of the real tool calls immediately before and after
     each masked decision. Explain why the missing action is the necessary transition
     and why a plausible alternative would be weaker.
- Real tool observations produced after a masked action remain in the trajectory. For
     that same action, treat its result only as hindsight outcome evidence and do not
     claim it was known beforehand. For the following masked action, that observation
     is valid prior evidence and must explain the state transition.
- Attribute each later result to the specific earlier action that produced it and
     explain why that action made the result possible.
- Reconstruct only <MASK_STEP_{target_step_ordinal}>. Other masked decisions remain
     context; do not emit them or invent environment tool results.

The system message defines the per-step JSON schema, dynamic evidence-aligned labels,
     optional original assistant-content references, and exact tool-call constraints.
     Follow those system instructions.

## Conversation Log

{masked_conversation}
```

Step B.4 SFT system prompt

```
You are the chained decision-reasoning module of a coding agent.

You receive a coding task followed by a compressed real execution chain. The first
     assistant tool call is the base action that produces the initial real observation.
      The subsequent target decisions must use each preceding observation to choose the
      next necessary action.

For every target decision, identify the current repository state, available evidence,
     remaining uncertainty, and unresolved subgoal. Explain how the preceding real
     observation changes the state or determines the next action.

Later trajectory messages may be used as outcome evidence when learning why a decision
     was valuable. When using such evidence, identify which earlier action produced
     that outcome and explain why that particular action made the outcome possible. Do
     not describe later observations as if they were already known at decision time.

Consider the logic and necessity of the tool calls immediately before and after every
     target decision. Explain why the chosen call is the necessary transition, without
     treating a later result as prior evidence.

At each target decision, produce concise evidence-grounded reasoning, a normal
     assistant message, and the minimal real tool call needed at that state.
```

Step B.4 SFT user prompt

```
# Chained masked-decision reconstruction

The conversation below is a complete real coding-agent trajectory. Several valuable
     assistant decisions in one continuous execution chain are replaced by ordered `<
     MASK_STEP_k>` markers. Each marker hides that assistant turn's reasoning, normal
     response text, and tool calls.

Reconstruct all masked decisions as a causal observation-to-action chain:
- For each step, determine the repository state, evidence, uncertainty, and unresolved
     subgoal immediately before its marker.
```

```
- Consider the logic and necessity of the real tool calls immediately before and after
    each masked decision. Explain why the missing action is the necessary transition
    and why a plausible alternative would be weaker.
- Real tool observations produced after a masked action remain in the trajectory. For
    that same action, treat its result only as hindsight outcome evidence and do not
    claim it was known beforehand. For the following masked action, that observation
    is valid prior evidence and must explain the state transition.
- Attribute each later result to the specific earlier action that produced it and
    explain why that action made the result possible.
- Reconstruct only the masked assistant decisions. Do not invent or output environment
    tool results.

## Conversation Log

{masked_conversation}
```

D.4 Step C Evidence-Bounded Failure Correction

The teacher pair constructs a structured correction target, and the SFT pair presents the failed history as its input. The task and JSON trace are inserted verbatim from the admitted source record.

Step C Construction system prompt

```
You generate high-quality trajectory-correction SFT data for a normal coding agent.
You will receive the original user task and the full failed trajectory from a weaker
    model.
Use the failed trajectory to identify the student's behavioral mistakes and teach the
    corrected coding-agent behavior.

The goal is not to guarantee an exact applyable patch without a repository sandbox. The
     goal is to produce a high-quality corrected trajectory target that helps a normal
     coding agent avoid the failed behavior and converge on the right fix.

Return one compact JSON object and nothing else. Do not wrap it in markdown fences.
    Escape all newlines inside JSON string values as \n.
Length budget: reasoning_content <= 4000 characters, assistant_content <= 1500
    characters, patch_sketch <= 1500 characters. Keep reasoning concise and diagnostic;
     do not copy long source snippets or replay the full trajectory.
The JSON object schema is:
{
  "reasoning_content": "Internal teacher reasoning. It may discuss the failed
     trajectory, the student's mistakes, the correct repair strategy, relevant files/
     functions, uncertainty, and validation needs.",
  "assistant_content": "The final response of a normal coding agent to the original
     user. Do not mention teacher, student, failed trajectory, SFT, or meta prompt. Do
     not pretend to have run tools. Focus on the corrected fix direction, concrete code-
     change plan, and verification strategy.",
  "root_cause": "Evidence-grounded concise explanation of the underlying cause of the
     failure.",
  "correction": {
    "student_failure_modes": ["short labels such as stuck_in_exploration, wrong_patch,
     ignored_tests, tool_protocol_errors"],
    "target_files": ["files or modules that should be changed"],
    "fix_strategy": "Concrete repair strategy the normal coding agent should follow.",
    "verification_strategy": "Concrete way to verify the fix. Do not claim tests were
     run unless the trace proves it."
  },
  "patch_sketch": "Optional best-effort unified diff or pseudo-diff. It may be empty.
     If not exact, make it a clearly labeled sketch and avoid fake hashes/index lines.",

  "quality": {
    "trajectory_correction_valid": true,
    "requires_repo_validation": true,
    "patch_is_exact": false,
    "mentions_teacher_or_student_in_content": false,
    "content_has_fake_tool_execution": false,
    "confidence": 0.0
  }
}

Strict rules:
- root_cause must state the underlying cause, rather than merely repeating the observed
     error or test failure. It must be supported by the trajectory.
```

```
- Keep reasoning_content short: summarize root cause, failed behavior, corrected
    strategy, and validation in 6-10 concise sentences or bullets.
- assistant_content must read like a normal coding agent's final update to the user.
- assistant_content must not contain teacher/student/failed trajectory/SFT/meta
    language.
- assistant_content must not contain fake command execution, shell snippets, or
    exploratory narration like "let me read".
- patch_sketch is optional and must not contain placeholder hashes, XXX hunks, or
    invented exact line numbers.
- If exact patch correctness is uncertain, set quality.patch_is_exact=false and quality.
    requires_repo_validation=true.
- The correction object is required and should carry the core training signal.
```

Step C Construction user prompt

```
# Original user task
{source_query}

# Failed agent trajectory
The JSON below is a failed attempt by a coding agent. Use it as context to diagnose
    what went wrong and provide the corrected coding-agent response. Do not continue
    the failed trajectory or pretend to run tools.
```json
{prompt_trace_json}
```

Teacher task
Use the original task and full failed trajectory to create high-quality trajectory-
    correction SFT data.
Return exactly one compact JSON object matching the correction schema in the system
    message.
The reasoning_content may discuss the failed trajectory and mistakes.
The assistant_content must be a normal coding agent final response and must not mention
     teacher, student, failed trajectory, SFT, or meta prompt.
Do not put fake command execution, shell snippets, or exploratory narration in
    assistant_content.
The correction object is required and should identify failure modes, target files, fix
    strategy, and verification strategy.
patch_sketch is optional. If exact patch correctness is uncertain, keep patch_is_exact=
    false and requires_repo_validation=true.
```

Step C SFT system prompt

```
You are a code repair agent. You receive a code issue and a failed coding-agent
    trajectory, then provide the correct diagnosis and fix.

In your thinking, analyze the failed trajectory: what the agent tried, where it went
    wrong, and what the correct approach should be.

In your response, explain the root cause concisely, explain what the failed trajectory
    missed or did wrong, and describe the minimal source-code fix with exact file
    paths. Provide concrete code-level guidance. Include a diff or code snippet when
    the failed trajectory provides enough source context or when the change is non-
    trivial. Do not claim that you edited files, saved patches, committed changes, or
    ran tests.
```

Step C SFT user prompt

```
# Original user task
{source_query}

# Failed agent trajectory
The JSON below is a failed attempt by a coding agent. Use it as context to diagnose
    what went wrong and provide the corrected coding-agent response. Do not continue
    the failed trajectory or pretend to run tools.
```json
{prompt_trace_json}
```
```