DEPTRACER: Propagation-Aware Failure Attribution for Agentic Optimization

Anonymous ACL submission

Abstract

Failure attribution for LLM agents is normally evaluated as localization: given a failed trajectory, identify the decisive agent-step pair responsible for the outcome. Although this output form facilitates standardized benchmarking, it is poorly aligned with repair and optimization, which require denser, fine-grained signals about how failures propagate through subsequent decisions. We present DEPTRACER, a failure attribution framework that reframes attribution as continuous scoring over trajectory segments and their constituent steps. DEPTRACER builds a trajectory-grounded attribution dependency graph with LLM-based signal extraction, then applies deterministic attribution propagation to compute attribution scores. The resulting scores remain compatible with standard single-step localization while providing propagation-aware context for downstream optimizers. Across attribution benchmarks, DEPTRACER achieves competitive top-ranked localization and substantially higher propagation-oriented recall. In closed-loop optimization, the same feedback interface improves performance across agent architectures, coding harnesses, and four optimization methods spanning inference-time refinement and self-evolution; iterative attribution yields up to +10.16pp on GEPA, compared to +2.16pp with single-pass attribution. Our code is available at `https://anonymous.4open.science/r/DepTracer-B078`.

1 Introduction

Large Language Model (LLM) agents have demonstrated strong capabilities in code generation (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025), general task planning (Mialon et al., 2024), and scientific reasoning (Ghafarollahi and Buehler, 2024). As task complexity grows, these systems increasingly rely on multi-agent collaboration (Hu et al., 2025; Li et al., 2023), sophisticated harness engineering (Jimenez et al., 2024; Merrill et al., 2026; Anthropic, 2026), and multi-stage workflows (Su et al., 2026; Wang et al., 2025). However, this architectural complexity also exacerbates fragility: empirical studies show that leading multi-agent systems fail at surprisingly high rates on real-world tasks (Cemri et al., 2025), and errors that arise in long interactions tend to propagate rather than self-correct (Laban et al., 2025). Pass/fail metrics alone are insufficient for improvement—they fail to offer any signal about which decision caused the failure or how it spread downstream. *Failure attribution*—identifying which steps in a trajectory were responsible for a failure—is therefore essential for understanding and improving agent systems, underpinning use cases from debugging and failure-derived data curation to targeted Self-Refine.

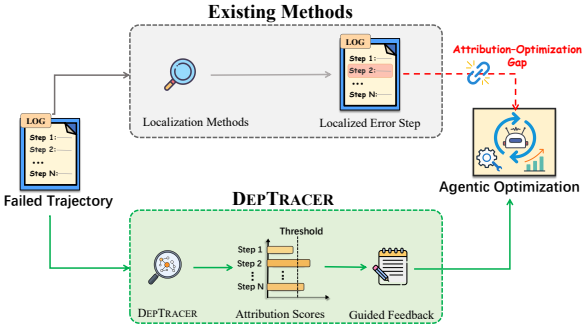


Figure 1: **Closing the attribution–optimization gap.** From the same *failed trajectory*, existing *localization methods* (top) output a single *localized error step*, which carries no graded or propagation information and cannot drive a downstream optimizer—the *attribution–optimization gap*. DEPTRACER (bottom) instead scores every step and converts the ranked *attribution scores* into structured *Guided feedback* that directly drives *agentic optimization*.

Existing failure attribution methods and benchmarks largely adopt a *localization protocol*: given a failed trajectory, identify the decisive agent-step pair responsible for the failure (Zhang et al., 2025b; Li et al., 2026; Zhang et al., 2026). Identifying

such a decisive step is a sound diagnostic goal; the limitation is that this line of work develops attribution as a *standalone* task, disconnected from the optimization it is meant to serve, in three respects. **(I). Evaluation-oriented design.** Attribution methods are built and benchmarked to match gold error labels, so success is measured by localization accuracy rather than by whether the resulting signal improves an agent—the output is tuned for clean evaluation, not for downstream repair. **(II). Optimization-opaque output.** A bare error label tells an optimizer *where* a failure surfaced but not *how* its effect propagated through later steps, and exposes no consumable structure—comparable scores or rankings—for deciding where and how much to revise. We term this mismatch the *attribution–optimization gap*. **(III). Limited generalizability.** Existing methods are tightly coupled to specific settings: CODETRACER (Li et al., 2026) is designed for code generation harnesses, AGENTRACER (Zhang et al., 2026) primarily targets multi-agent collaboration, and Who&When (Zhang et al., 2025b) is evaluated on GAIA-based multi-agent frameworks. No single attribution interface transfers across the architectures, harnesses, and optimizers one would actually want to improve.

We propose DEPTRACER, an attribution framework that reframes failure attribution as continuous scoring over trajectory segments and steps. DEPTRACER infers a trajectory-grounded attribution dependency graph and propagates attribution mass along it to assign each segment a non-negative *attribution score* A_i. The top-ranked step supports standard localization evaluation, while the remaining scores provide propagation-aware context for downstream optimization.

Generally, our contributions are:

- We reframe failure attribution from single-point localization to continuous scoring over trajectory segments and steps, preserving localization compatibility while adding denser feedback for optimization.
- We propose DEPTRACER, which traces trajectory-grounded dependencies, scores faulty steps via propagation, and generates *Guided feedback* for agentic optimization.
- We demonstrate that DEPTRACER attains strong localization results on two failure attribution benchmarks, and that its feedback improves agent performance across four optimizers, spanning both *per-task refinement* and *cross-task self-evolution* of the agent program.

2 Related Work

Localization-oriented failure attribution. Existing agent attribution benchmarks and methods operationalize failure diagnosis as localization: identify the responsible agent, step, or compact set of steps in a failed trajectory (Zhang et al., 2025b; Li et al., 2026; Zhang et al., 2026). Concurrent graph-based approaches (Zhang et al., 2025a; Wang et al., 2026) and verification-centric methods (Qiao et al., 2026) represent complementary directions within this space. This protocol is valuable because it gives a clear target and enables comparison across systems. However, a benchmark-facing localization label is not necessarily an optimization-facing signal. DEPTRACER remains compatible with this protocol by using the top-ranked step for evaluation, but its primary output is a trajectory-wide attribution vector.

Optimization feedback without structured attribution. Self-improvement methods such as Reflexion and Self-Refine use feedback from previous failures to guide future attempts (Shinn et al., 2023; Madaan et al., 2023). Prompt-level optimizers such as GEPA further aggregate feedback across tasks to update the agent program (Agrawal et al., 2026). These methods show that feedback can improve agent behavior, but their feedback is usually free-form text or scalar performance information. They do not expose comparable attribution scores over trajectory segments and steps, nor do they tie the feedback to an explicit dependency graph.

3 Methodology

3.1 Problem Definition

Given a failed agent system trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$, let $Z(\tau) \in \{0, 1\}$ denote the task outcome, where $Z(\tau) = 1$ indicates failure. Existing attribution benchmarks expose a localization target: identify the responsible agent-step pair (I, t), or a small set of erroneous steps, that best explains the failure (Zhang et al., 2025b; Li et al., 2026; Zhang et al., 2026). This benchmark-facing formulation is useful because it yields compact labels and supports direct comparison across methods.

However, localization is not the same as repair. Downstream optimizers need a denser, fine-grained

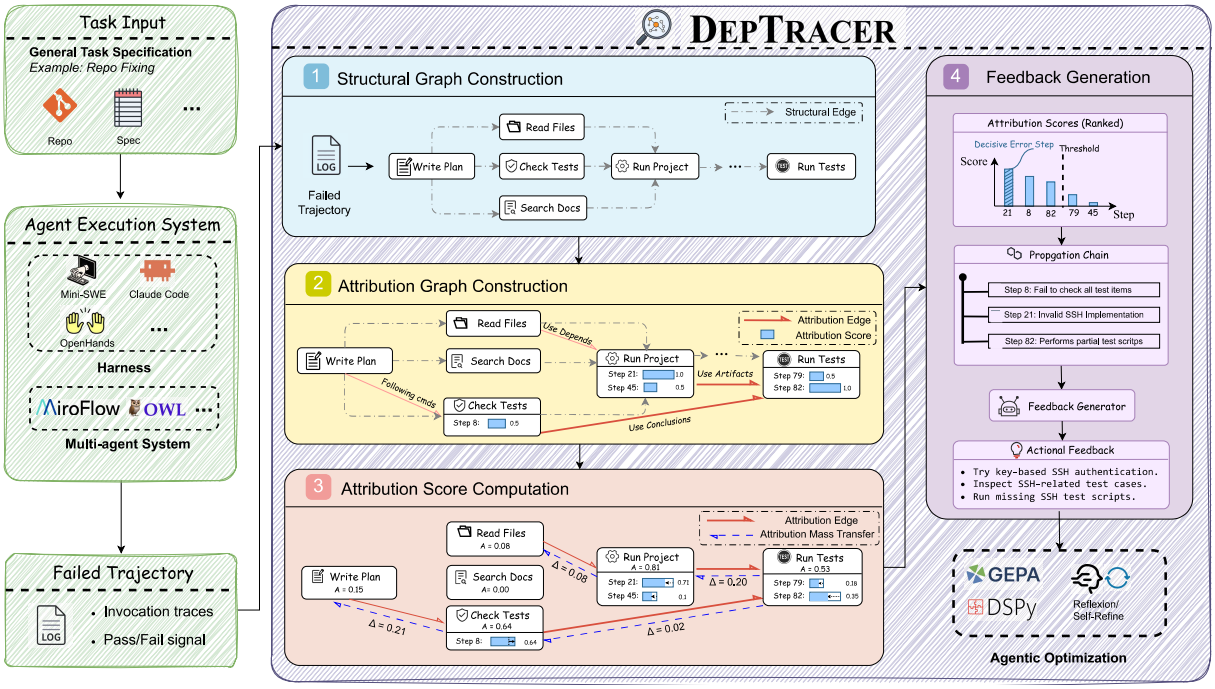


Figure 2: Overview of DEPTRACER. Diverse agent architectures are unified as segment-level DAGs and processed through structural graph construction, attribution dependency extraction, score computation, and feedback generation. The resulting ranked scores guide downstream agentic optimization.

signal that is tied to the trajectory structure. Especially in complex, long-horizon tasks, a single pair (I, t) conveys little about how early mistakes affect later tools, observations, or decisions.

We therefore formulate failure attribution as *continuous scoring*: DEPTRACER assigns each step a non-negative attribution score $A_{i,k}$ reflecting its contribution to the failure, with the top-scoring step remaining compatible with standard localization evaluation while the full vector supplies propagation-aware optimization signals.

3.2 Overview

To produce these scores, DEPTRACER builds an *attribution dependency graph* of the trajectory—capturing which outputs were referenced, which issues were inherited, and where local failures surfaced. A raw trajectory, however, exposes only the *invocation topology*—who called whom and who executed after whom—which differs from this failure-relevant dependency structure; we term the mismatch the *structural–attribution gap*. As illustrated in Figure 2, DEPTRACER bridges this gap through three phases: **Phase 1** (§3.3) constructs a structural DAG $\mathcal{G}_{\text{struct}}$ from the invocation topology; **Phase 2** (§3.4) uses LLM-based analysis to infer failure-relevant dependency edges, yielding the attribution dependency graph $\mathcal{G}_{\text{attr}}$; **Phase 3** (§3.5) deterministically propagates attribution mass backward along $\mathcal{G}_{\text{attr}}$ to produce the attribution scores. The procedure is summarized in Algorithm 1.

3.3 Structural Graph Construction

We model the execution trajectory of any agent architecture as a directed acyclic graph (DAG) $\mathcal{G}_{\text{struct}} = (\mathcal{V}, \mathcal{E}_{\text{struct}})$, where each node $v_i \in \mathcal{V}$ corresponds to a *trajectory segment*: a coherent block of turns or steps that serves as the graph's observation unit. The partitioning of a raw trajectory into segments is architecture-dependent (see Appendix B), and edges $\mathcal{E}_{\text{struct}}$ reflect the invocation topology: sequential dependencies for single-agent trajectories and fan-out/fan-in structures for multi-agent systems (Figure 2, left). Crucially, this structural graph encodes only temporal and organizational relationships, not the failure-relevant dependencies—a distinction that Phase 2 addresses.

3.4 Attribution Graph Construction

Phase 2 turns the structural graph into the attribution dependency graph $\mathcal{G}_{\text{attr}}$ through two LLM-based steps: a *local* pass that characterizes each segment on its own, and a *global* pass that calibrates these local signals against the full trajectory.

Algorithm 1 DEPTRACER

Require: Failed trajectory τ with $Z(\tau) = 1$
Ensure: Step attribution scores $\{A_{i,k}\}$, segment aggregates $\{A_i\}_{i=0}^{M-1}$

// Phase 1: Structural Graph Construction
$\mathcal{G}_{\text{struct}} \leftarrow \text{BuildDAG}(\tau)$ ▷ §3.3
// Phase 2: Attribution Graph Construction
Context $\leftarrow \emptyset$
for v_i in topological order of $\mathcal{G}_{\text{struct}}$ **do**
  $(O_i, Q_i) \leftarrow \mathcal{F}(\text{Context},\ \tau_i)$ ▷ §3.4
  Context $\leftarrow$ Context $\cup\, O_i \cup Q_i$
end for
$\{(O'_i, Q'_i)\} \leftarrow \mathcal{C}(\{(O_i, Q_i)\},\ Z(\tau))$ ▷ Global Calibration
// Phase 3: Deterministic Attribution (no LLM)

for each $v_i \in \mathcal{V}$ **do**
  Compute B_i, OU_i, C_i via Eq. 1–3 ▷ §3.5
end for
for each edge $v_j \to v_i$ in $\mathcal{G}_{\text{attr}}$ where $B_i > 0$ **do**
  Propagate attribution mass $\Delta_{j \leftarrow i}$ via Eq. 4
end for
for each $v_i \in \mathcal{V}$ **do**
  Compute A_i via Eq. 5
  Compute $A_{i,k}$ for each step k via Eq. 7
end for
return $\{A_{i,k}\}, \{A_i\}$

Local Dependency Characterization. We process segments in topological order, so that when analyzing v_i the outputs and issues of all upstream segments are available as context. The analyzer $\mathcal{F}$ serves two purposes. *(1) Local fault scoring:* $\mathcal{F}$ identifies the issues within v_i (indexed by m) and annotates each with a fault score $s_m \in \{0, 0.5, 1\}$—its confidence that the issue is a genuine fault (high, medium, or low)—together with the set of steps $R_m \subseteq \tau_i$ it concerns; the per-issue annotations $\{(s_m, R_m)\}_m$ constitute v_i's *local fault issues* Q_i. *(2) Dependency edges:* $\mathcal{F}$ determines which upstream outputs v_i referenced and which upstream issues it inherited, adding a dependency edge $v_j \to v_i$ for each dependency; it also marks the upstream issues that v_i resolves, preventing already-resolved issues from propagating further downstream. The outputs O_i produced by v_i and Q_i are added to the context so that downstream segments can declare their dependencies.

Global Calibration. Analyzing segments in isolation introduces three biases—*repetition inflation*, *environmental contamination*, and *latent failure omission* (Appendix C). A global calibrator $\mathcal{C}$ (LLM-based) revisits all segments together with the task outcome $Z(\tau)$ and revises the issues accordingly: merging duplicates, discarding environment-induced issues, adding overlooked ones, and adjusting issue scores. This yields the final attribution dependency graph with calibrated issue annotations.

3.5 Attribution Computation

This phase computes attribution score A_i via deterministic formulas, with no LLM invocation. Analogous to how backpropagation assigns credit through a computation graph, DEPTRACER propagates attribution mass backward through the attribution dependency graph, from the failure outcome to the steps responsible for it.

Segment-Level Scoring. A segment's own contribution to failure is its *base score* B_i: the mean of the $\{0, 0.5, 1\}$ scores of its unsolved issues, or 0 when it has none:

$$B_i = \begin{cases} \frac{1}{|\mathcal{U}_i|} \sum_{m \in \mathcal{U}_i} s_m, & \mathcal{U}_i \neq \varnothing \\ 0, & \mathcal{U}_i = \varnothing \end{cases} \tag{1}$$

where $\mathcal{U}_i$ is v_i's set of unsolved issues and $s_m \in \{0, 0.5, 1\}$ is issue m's local fault score (§3.4). Output utilization OU_i measures how much of v_i's output is referenced downstream, capped at κ:

$$OU_i = \min\left(1 - \frac{|\text{unused outputs}_i|}{|\text{all outputs}_i|},\ \kappa\right) \tag{2}$$

The two combine into the segment's *critical score*:

$$C_i = B_i \cdot (1 + OU_i) + \alpha_{OU} \cdot OU_i \tag{3}$$

OU_i enters C_i in two ways. As a multiplier, it lets an erroneous output that is referenced downstream increase the segment's failure contribution. As an additive term $\alpha_{OU} \cdot OU_i$, it serves as a safeguard: if issue detection misses a segment's faults and yields $B_i = 0$, a heavily-referenced segment—one many downstream steps depend on—still retains a small nonzero score instead of being dropped entirely.

Attribution Propagation. C_i captures only a segment's local faults, but a failure often originates upstream and only surfaces downstream. We therefore propagate attribution mass backward along the attribution dependency graph: when a downstream

Backbone	Who&When – Algo		Who&When – Hand		CodeTraceBench†		
	w/ GT	w/o GT	w/ GT	w/o GT	Prec.	Rec.	F1
Claude-Sonnet-4							
All-at-Once	50.79 (→51.59)	48.41 (→49.21)	13.79 (→14.59)	15.09 (→15.89)	25.71	26.94	26.31
Outline Tracer	46.83 (→48.41)	45.24 (→46.83)	15.51 (→27.59)	10.34 (→24.14)	**34.30**	35.40	33.10
DEPTRACER (Ours)	**72.23** (→**73.02**)	**70.63** (→**72.23**)	**31.03** (→**60.71**)	**29.31** (→**65.52**)	31.30	**65.80**	**38.30**
Qwen3-Max							
All-at-Once	47.62 (→47.62)	46.82 (→47.62)	16.67 (→15.52)	18.97 (→20.69)	28.37	40.12	33.24
Outline Tracer	46.03 (→48.41)	40.48 (→42.06)	13.79 (→13.79)	8.62 (→10.34)	**30.20**	26.10	28.00
DEPTRACER (Ours)	**57.94** (→**65.08**)	**60.32** (→**65.08**)	**32.14** (→**60.71**)	**25.00** (→**64.29**)	29.30	**79.20**	**38.80**
DeepSeek-V4-Flash							
All-at-Once	46.03 (→46.83)	43.92 (→46.03)	22.12 (→24.56)	20.11 (→22.41)	22.14	31.26	25.92
Outline Tracer	46.03 (→46.83)	43.65 (→45.24)	8.62 (→10.34)	6.90 (→8.62)	24.50	58.00	34.50
DEPTRACER (Ours)	**63.49** (→**73.02**)	**69.84** (→**77.78**)	**29.31** (→**60.34**)	**20.69** (→**58.62**)	**28.07**	**72.50**	**36.41**

Table 1: Failure attribution accuracy on two attribution benchmarks. *w/ GT*: with ground-truth answer; *w/o GT*: without. Values in parentheses denote F1 under ±1 step tolerance. †Filtered subset; see Appendix D.

segment v_i fails partly because it referenced an upstream segment v_j's erroneous output, v_j receives attribution mass from v_i in proportion to how much of v_i's references point to v_j:

$$\Delta_{j \leftarrow i} = \alpha_b \cdot C_i \cdot \frac{|\text{refs}_{i \to j}|}{|\text{refs}_i|} \tag{4}$$

Summing these per-edge terms gives each segment's received and sent attribution mass, $\text{recv}_i = \sum_j \Delta_{i \leftarrow j}$ and $\text{sent}_i = \sum_j \Delta_{j \leftarrow i}$. A segment keeps its critical score after subtracting sent mass, floored at 0, and adds received mass capped at $\gamma \cdot B_i$:

$$A_i = \max(0,\ C_i - \text{sent}_i) + \min(\text{recv}_i,\ \gamma \cdot B_i) \tag{5}$$

The cap $\gamma \cdot B_i$ with $\gamma < 1$ bounds inherited attribution below the segment's fault level, so propagation refines rather than overrides local evidence.

Step-Level Attribution. The scores above are segment-level; we decompose each segment's score onto its constituent steps. Writing the two terms of Eq. 5 as

$$A_i^{\text{own}} = \max(0,\ C_i - \text{sent}_i), \quad A_i^{\text{recv}} = \min(\text{recv}_i,\ \gamma \cdot B_i) \tag{6}$$

the score of step k within segment v_i is:

$$A_{i,k} = A_i^{\text{own}} \sum_{m \in \mathcal{U}_i} \frac{s_m}{S_i} \cdot \frac{\mathbb{1}[k \in R_m]}{|R_m|} + A_i^{\text{recv}} \cdot \frac{\mathbb{1}[k \in \bigcup_m R_m]}{|\bigcup_m R_m|} \tag{7}$$

where $S_i = \sum_{m \in \mathcal{U}_i} s_m$. The own-fault term A_i^{own} is distributed across unsolved issues in proportion to their scores s_m, then spread over each issue's steps R_m; the received term A_i^{recv} is spread uniformly over all issue-related steps. By construction $\sum_k A_{i,k} = A_i$, so the step scores are decomposed from the segment score, and $A_{i,k}$ is DEPTRACER's final per-step attribution score.

Consistency. The scoring satisfies the objective in §3.1: root-like faulty segments receive high A_i via a high base score B_i amplified by OU_i; dependency-chain intermediaries receive bounded scores ($\gamma \cdot B_i$); irrelevant segments approach zero.

4 Experiments

We evaluate DEPTRACER through three research questions:

RQ1: Attribution fidelity. Do scores remain effective under localization metrics while exposing propagation-oriented evidence at practical cost?

RQ2: Optimizer universality. Can the attribution signal improve multiple optimizer families?

RQ3: Framework universality. Does the same scoring interface transfer across agent architectures and execution harnesses?

Implementation. We extract attribution signals with Claude-Sonnet-4 (Anthropic, 2025), Qwen3-Max (Qwen Team, 2025), and DeepSeek-V4-Flash (DeepSeek-AI, 2026); optimization experiments use DeepSeek-V4-Flash unless otherwise stated. Default hyperparameters are α_{OU}=0.3, α_b=0.3, γ=0.5, and κ=0.5. For localization, predictions are derived from attribution scores by selecting the top-ranked step for Who&When, and by collecting issue-associated steps within the top-3 ranked segments for all other benchmarks.

4.1 RQ1: Attribution Fidelity and Cost

Setup. We evaluate on two failure attribution benchmarks: CodeTraceBench (Li et al., 2026) (filtered subset; Appendix D) and Who&When (Zhang et al., 2025b), across three LLM backbones (Claude-Sonnet-4, Qwen3-Max, DeepSeek-V4-Flash). We compare against **All-at-Once** direct

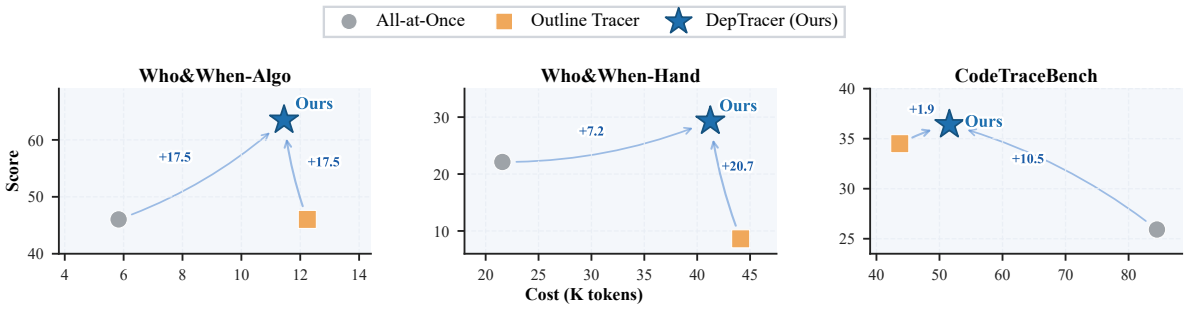


Figure 3: Attribution score vs. token cost per case (localization accuracy for Who&When; F1 for CodeTraceBench).

	Inference-time		Self-evolution	
	Self-Refine	**Reflexion**	**COPRO**	**GEPA**
Vanilla	69.89 ± 0.38	64.25 ± 1.01	59.38 ± 3.62	56.25 ± 2.73
DEPTRACER$_{\text{one-shot}}$	73.12 ± 1.01	68.28 ± 2.66	61.37 ± 3.15	58.41 ± 2.38
DEPTRACER$_{\text{full}}$	**74.73** ± 1.37	**68.55** ± 0.66	**62.50** ± 2.64	**66.41** ± 1.93
Gain over Vanilla	+4.84	+4.30	+3.12	**+10.16**

Table 2: DEPTRACER improves all four optimizer families on GAIA (single-agent ReAct backbone).

LLM judgment and an **Outline Tracer** baseline; detailed descriptions and metrics are in Appendix A.1–A.2. Table 1 and Figure 3 report results; we highlight two findings.

DEPTRACER consistently outperforms both baselines across all three backbones. DEPTRACER outperforms both baselines across three backbones. On Who&When, it exceeds direct LLM judgment by up to +21pp and Outline Tracer by up to +25pp, confirming structured graph extraction with deterministic scoring is stronger than end-to-end LLM-as-judge attribution. Under ±1 step tolerance, gains exceed +33pp on Who&When-Hand, indicating attribution scores place high mass in the correct failure neighborhood even when the exact step is missed. On CodeTraceBench, Recall improves from 35.40 to 65.80 over Outline Tracer while F1 rises (38.30 vs. 33.10); the modest Precision reduction reflects scoring all propagation-relevant steps rather than collapsing attribution to a single point.

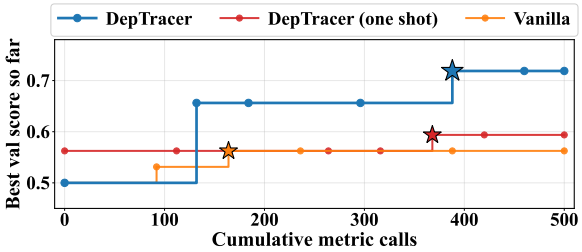
Accuracy gains come at no additional token cost; DEPTRACER is Pareto-superior on Who&When-Hand. DEPTRACER's accuracy gains come at token cost comparable to the Outline Tracer (Figure 3). Cost parity arises because Phase 3 attribution propagation is fully deterministic and requires no additional LLM calls. On Who&When-Hand the advantage is more pronounced: the Outline Tracer costs more than DEPTRACER while scoring below All-at-Once, making DEPTRACER Pareto-superior on that benchmark.

4.2 RQ2: Optimizer Universality

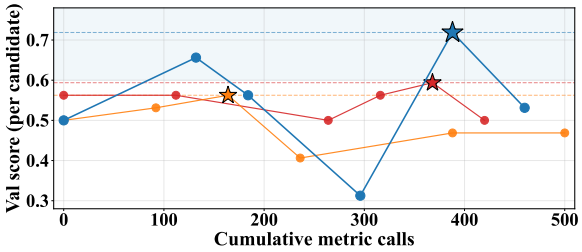
Setup. We test transfer across optimizer families on GAIA (Mialon et al., 2024) with a ReAct agent (Yao et al., 2023), covering two paradigms: inference-time refinement (Self-Refine, Reflexion) and self-evolution (COPRO, GEPA); paradigm definitions are in Appendix A.3. We compare each optimizer under vanilla run-level feedback against the same optimizer driven by DEPTRACER attribution feedback. Localization baselines are excluded as direct comparators—bare error labels cannot bridge the attribution–optimization gap (§1)—with feedback-augmented ablations in §6. Table 2 reports results; we highlight two findings.

DEPTRACER improves all four optimizers; one-shot attribution suffices for inference-time repair. DEPTRACER improves over vanilla across all four optimizers in both paradigms. For inference-time methods, one-shot attribution already captures most of the benefit: Self-Refine gains +3.23pp and Reflexion gains +4.03pp over vanilla, while full iterative attribution adds only a marginal further +1.61pp and +0.27pp respectively. Per-task repair primarily requires attribution *quality*—identifying which step to revise—rather than cross-task signal aggregation.

Self-evolvement benefits most: GEPA gains +10.16pp under iterative attribution. Under prompt-level optimization the gap between full



(a) Best validation score achieved so far as training progresses.



(b) Per-candidate validation score at each evaluation step.

Figure 4: Visualization of GEPA optimization on GAIA. ☆ marks each method's best-scoring candidate.

Setting	Self-Refine Acc. (%) ↑	Self-Refine Avg. Steps ↓	Reflexion Acc. (%) ↑	Reflexion Avg. Steps ↓
Single Agent – ReAct				*Base: 59.68*
Vanilla	69.89 ± 0.38	7.3	64.25 ± 1.01	5.4
Ours	**74.73** ± 1.37	**7.2**	**68.55** ± 0.66	**5.1**
Instr.-Exec. – OWL				*Base: 60.63*
Vanilla	62.20 ± 1.20	7.6	63.00 ± 1.20	3.7
Ours	**63.00** ± 0.23	**6.0**	**64.57** ± 2.08	**2.0**
Workflow – MiroFlow (w/o sub)				*Base: 48.82*
Vanilla	57.74 ± 0.98	38.5	55.91 ± 1.29	22.1
Ours	**64.04** ± 3.17	**33.5**	**56.17** ± 1.86	**20.9**
Orch.-Worker – MiroFlow (w/ sub)				*Base: 66.93*
Vanilla	**82.41** ± 0.98	34.0	81.89 ± 0.64	27.9
Ours	81.10 ± 1.21	**30.3**	**82.68** ± 1.29	**25.7**

Table 3: Cross-architecture optimization on GAIA (*Ours*: DEPTRACER feedback; accuracy %, mean±std).

Harness	Base	With Reflexion Vanilla	With Reflexion Ours	With Reflexion Δ
Claude Code	48.31	50.56	**56.18**	+5.62
OpenHands	58.42	64.04	**66.30**	+2.26
mini-swe-agent	40.45	41.57	**46.07**	+4.50

Table 4: Accuracy (%) on Terminal-Bench (*Ours*: DEPTRACER feedback; Δ: improvement over Reflexion).

and one-shot attribution widens substantially. COPRO gains +3.12pp with full DEPTRACER versus +1.99pp with one-shot; GEPA gains +10.16pp versus +2.16pp—an +8.00pp advantage from iterative over one-shot attribution. Figure 4(a) shows the mechanism directly: DEPTRACER's best validation score steps upward to 0.72 across search rounds, while one-shot plateaus at 0.60 and vanilla stalls at 0.56; panel (b) shows DEPTRACER generating higher-scoring candidates at late iterations, consistent with a search process whose repair signal sharpens as more failures are attributed.

4.3 RQ3: Framework Universality

Setup. We test framework universality on two fronts. For architecture diversity, we evaluate on GAIA across four agent patterns: single-agent ReAct, instructor-executor, workflow, and orchestrator-worker. For harness diversity, we evaluate on TerminalBench 2.0 (Merrill et al., 2026) across three coding harnesses: Claude Code, OpenHands, and mini-swe-agent. Both settings use Reflexion as the optimizer; full configuration details are in Appendix A.4. Tables 3 and 4 report results; we highlight two findings.

DEPTRACER lifts accuracy across agent architectures and coding harnesses. DEPTRACER improves over vanilla in seven of the eight architecture–optimizer combinations on GAIA (Table 3). The lone exception—Self-Refine on the orchestrator-worker setting—occurs at an already-high baseline (82.41%), where the ceiling limits further gains. DEPTRACER feedback also lifts Reflexion across all three coding harnesses (+5.62, +2.26, +4.50pp on TerminalBench; Table 4), despite substantial differences in execution environment and logging format.

DEPTRACER feedback is compact, with accuracy gains preserved. DEPTRACER produces a smaller feedback footprint than vanilla. On GAIA, feedback is shorter in both absolute tokens (213.0 vs. 310.3) and as a share of optimizer input (29.7% vs. 60.4%). On TerminalBench, feedback is slightly longer in absolute terms but occupies only 0.61% of total input—a ∼ 0.2% increment over vanilla—while lifting mini-swe-agent's accuracy by +4.50pp. The accuracy gains therefore reflect signal quality, not feedback-budget inflation.

	GAIA Tokens	GAIA % Input	TerminalBench Tokens	TerminalBench % Input
Vanilla	310.3	60.4	203.4	0.41
DEPTRACER	**213.0**	**29.7**	305.7	0.61
Δ Acc (pp)	+4.03		+4.50	

Table 5: Feedback token cost per task, share of total optimizer input (%), and accuracy gain.

5 Case Study

We inspect a Terminal-Bench `build-cython-ext` task that asks the agent to make the Cython extension. The trajectory failed because the agent's

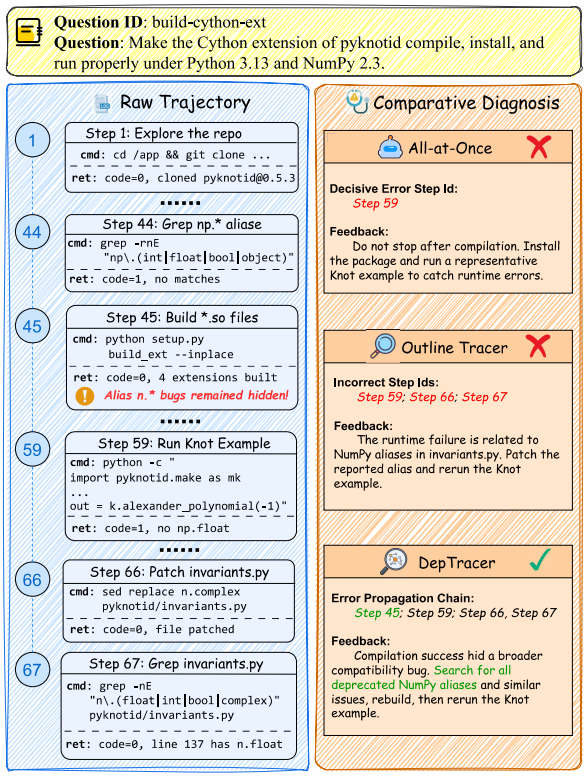


Figure 5: Case study on a TerminalBench `build-cython-ext` task.

NumPy-alias check at Steps 44–45 used the pattern `np.*` and silently overlooked legacy `n.*` aliases in `invariants.py`: the build succeeded, but downstream verification (Step 59) raised a runtime error and triggered two follow-up patches (Steps 66–67) that addressed the symptom rather than the cause.

Figure 5 contrasts three attribution methods. *All-at-Once* anchors on the visible Step 59 error and returns a generic "install and run a Knot example" hint, leaving the root cause untouched. *Outline Tracer* finds the alias issue but prescribes "patch the reported alias"—a symptomatic fix, because Step 45's build-success segment is never expanded. DEPTRACER traces Step 59's failure back through the dependency graph to Step 45, elevating its output-utilization score despite `code=0`, and yields feedback that targets the actual defect: *search for all deprecated NumPy aliases*. By propagating failure signals through the attribution graph, DEPTRACER recovers silent root causes masked by apparent success—converting a `code=0` step into an actionable repair target.

6 Ablation Study

Module Ablation. We ablate four components of DEPTRACER on CodeTraceBench, reporting Hit@3 (whether the true erroneous segment appears in the top-3 ranked segments) and F1 (Table 6). Global Calibration matters most (-15.52pp), as removing it leaves repetition inflation and environmental contamination uncorrected. Output Utilization follows (-11.79pp), confirming OU_i surfaces root-cause segments whose outputs propagate downstream. Removing the cap $\gamma \cdot B_i$ costs -11.35pp; Attribution Propagation contributes -9.05pp. All components are necessary: omitting any one degrades both Hit@3 and F1.

Configuration	Hit@3	F1
Full	77.62	38.30
w/o Attribution Propagation	68.57	35.42
w/o Clipping ($\gamma \cdot B_i$)	66.27	34.06
w/o Output Utilization	65.83	32.10
w/o Global Calibration	62.10	29.80

Table 6: Module ablation on CodeTraceBench.

Attribution Quality Ablation. To isolate whether gains stem from *attribution quality* rather than feedback format alone, we equip All-at-Once and Outline Tracer with DEPTRACER's feedback generator, fixing format and optimizer while varying only the attribution source (Table 7). All-at-Once and Outline Tracer yield marginal gains; DEPTRACER delivers $+4.84$ and $+4.30$pp—confirming attribution quality drives the gap. Structured feedback is only as useful as the attribution that feeds it—poor localization yields poor repair.

Attribution feedback	Self-Refine	Reflexion
None (vanilla)	69.89±0.38	64.25±1.01
All-at-Once	69.89±0.27	64.83±1.34
Outline Tracer	70.02±0.42	65.86±2.13
DEPTRACER (Ours)	**74.73**±1.37	**68.55**±0.66

Table 7: Ablation on attribution-feedback source. ReAct agent on GAIA under two inference-time optimizers.

7 Conclusion

DEPTRACER reframes failure attribution as continuous scoring over trajectory units, yielding signals compatible with localization benchmarks while providing propagation-aware feedback for downstream optimization. Across various optimizer families, multiple agent architectures, and diverse coding harnesses, structured attribution proves a broadly applicable foundation for agent improvement.

Limitations

DEPTRACER relies on inferred dependency graphs from observed trajectories. Missing or spurious edges can distort attribution propagation, so the method should be understood as trajectory-grounded structured attribution rather than ground-truth causal identification.

Existing benchmarks provide limited supervision for complete propagation chains. We therefore evaluate propagation indirectly through recall, tolerance-based localization, closed-loop optimization gains, and qualitative case analysis. Richer propagation annotations would allow a cleaner fidelity test.

Our current evaluation focuses on within-trajectory propagation. Long-term failure accumulation across repeated task attempts, persistent memory, or repeated deployments remains outside the present scope.

References

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*. ICLR 2026 Oral.

Anthropic. 2025. Claude 4. Model release announcement.

Anthropic. 2026. Claude Code: Overview. Product documentation.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*.

DeepSeek-AI. 2026. DeepSeek-V4-Flash. Model card.

Alireza Ghafarollahi and Markus J. Buehler. 2024. SciAgents: Automating scientific discovery through bioinspired multi-agent intelligent graph reasoning. *Advanced Materials*, page 2413523.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Bowei Xia, Tao Sun, Ziyu Ye, Zhaoxuan Jin, Yingru Li, Qiguang Chen, Zeyu Zhang, Yifeng Wang, Qianshuo Ye, Bernard Ghanem, Ping Luo, and Guohao Li. 2025. OWL: Optimized workforce learning for general multi-agent assistance in real-world task automation. *Preprint*, arXiv:2505.23885.

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

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling declarative language model calls into self-improving pipelines. *Preprint*, arXiv:2310.03714.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs get lost in multi-turn conversation. *arXiv preprint arXiv:2505.06120*.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative agents for "Mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36, pages 51991–52008.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, and 1 others. 2026. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *Preprint*, arXiv:2601.11868.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations*.

Hezhe Qiao, Hanghang Tong, Ee-Peng Lim, Bing Liu, and Guansong Pang. 2026. VerifyMAS: Hypothesis verification for failure attribution in LLM multi-agent systems. *Preprint*, arXiv:2605.17467.

Qwen Team. 2025. Qwen3-Max. Model release blog.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Shiqian Su, Sen Xing, Xuan Dong, Muyan Zhong, Bin Wang, Xizhou Zhu, Yuntao Chen, Wenhai Wang, Yue Deng, Pengxiang Zhu, Ziyuan Liu, Tiantong Li, Jiaheng Yu, Zhe Chen, Lidong Bing, and Jifeng Dai. 2026. MiroFlow: Towards high-performance and robust open-source agent framework for general deep research tasks. *Preprint*, arXiv:2602.22808.

SWE-agent Team. 2026. mini-swe-agent: The minimal AI software engineering agent. GitHub repository.

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, and 5 others. 2025. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*.

Yawen Wang, Wenjie Wu, Junjie Wang, and Qing Wang. 2026. From flat logs to causal graphs: Hierarchical failure attribution for LLM-based multi-agent systems. *Preprint*, arXiv:2602.23701.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. 2026. Agentracer: Who is inducing failure in the LLM agentic systems? In *International Conference on Learning Representations*.

Heng Zhang, Yuling Shi, Xiaodong Gu, Haochen You, Zijian Zhang, Lubin Gan, Yilei Yuan, and Jin Huang. 2025a. Graphtracer: Graph-guided failure tracing in llm agents for robust multi-turn deep search. *Preprint*, arXiv:2510.10581.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025b. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 76583–76599. PMLR.

Appendix

A Experiment Setting

A.1 Benchmarks and Metrics

CodeTraceBench. CodeTraceBench (Li et al., 2026) contains complete code-agent trajectories with human-annotated erroneous steps (`incorrect_step_ids`); these labels often cover the neighborhood of a root-like failure rather than a single isolated point. We report step-level Precision, Recall, and F1. We evaluate on a filtered subset; see Appendix D.

Who&When. Who&When (Zhang et al., 2025b) provides failure logs with responsible-agent and decisive-step labels, evaluated on its Hand-Crafted and Algorithm-Generated subsets. We report localization accuracy; values in parentheses denote ± 1-step tolerance.

A.2 Baselines

All-at-Once. The LLM receives the full trajectory and is prompted to identify the agent and step most responsible for the failure in a single forward pass, with no graph construction or structured extraction, following the All-at-Once formulation of Zhang et al. (2025b).

Outline-guided Tracer. Inspired by CodeTracer (Li et al., 2026), this baseline performs coarse-to-fine inspection. It first presents the LLM with a high-level outline of the trajectory—a compact listing of segments and their summaries—and asks the LLM to select the most suspicious segment. It then expands that segment, showing the LLM all steps within it to produce a final localization. The segmentation strategy is benchmark-dependent: on CodeTraceBench, the outline follows the code directory structure used by the original CodeTracer; on Who&When, where no directory structure exists, we construct the outline using generic trajectory segmentation (fixed-size windows aligned to agent boundaries). On CodeTraceBench, this baseline is therefore identical to the original CodeTracer method; on Who&When, it is our faithful re-implementation of the same outline-guided strategy.

A.3 Optimizer Paradigms

Inference-time refinement—Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023)—consumes feedback once per task without updating the agent program; the agent is re-prompted with the attribution feedback and asked to retry the task. *Self-evolvement*—COPRO (Khattab et al., 2023) and GEPA (Agrawal et al., 2026)—aggregates feedback across a validation set to evolve the agent's system prompt across search iterations.

A.4 Architectures and Harnesses

Agent architectures (GAIA). Single-agent ReAct (Yao et al., 2023) runs a single LLM in a reason–act–observe loop with no external scaffolding. **Instructor-Executor (OWL; Hu et al. 2025)** is a two-tier system: an instructor agent decomposes the task and issues subtask directives to an executor agent, which carries out tool calls and reports results back for synthesis. **Workflow (MiroFlow w/o sub; Su et al. 2026)** wraps a ReAct agent in a fixed three-stage pipeline: hint-augmented input preprocessing, a tool-use ReAct loop, and a forced summarization stage with answer extraction. **Orchestrator-Worker (MiroFlow w/ sub; Su et al. 2026)** is the full MiroFlow multi-agent configuration: the main orchestrator dynamically spawns specialized worker subagents to handle subtasks, then aggregates their outputs into a final answer.

Coding harnesses (TerminalBench). Claude Code (Anthropic, 2026) is an agentic coding assistant that operates in a terminal environment with file-system and shell tool access. **OpenHands** (Wang et al., 2025) is an open-source agent framework that provides a sandboxed runtime with a browser, shell, and code-execution tools. **mini-swe-agent** (SWE-agent Team, 2026) is a lightweight software-engineering agent that follows the SWE-agent interaction format with a minimal tool set.

TerminalBench experiments use Reflexion only: code tasks involve long execution traces and extended runtime, making Self-Refine impractical due to context length constraints and the cost of repeated full-trajectory evaluation.

B Segment Partitioning

Segments are the basic attribution units used by DepTracer. For code-generation agents, a segment boundary is created around substantive file operations, command-execution blocks, and verification attempts. For workflow agents, segments follow workflow phases such as planning, execution, and review. For multi-agent systems, each complete invocation of a worker or specialist agent

forms a segment. For long single-agent traces without natural boundaries, we use bounded windows and then merge adjacent short windows when they form a purely linear chain.

This partitioning affects attribution granularity but not the scoring algorithm. Finer partitions improve localization resolution but may introduce noisy splits in the attribution scores; coarser partitions are more robust but less diagnostic. Unless otherwise stated, experiments use the finest partition that preserves coherent task intent within each segment. Figure 6 illustrates the structural graph construction process on a representative trajectory.

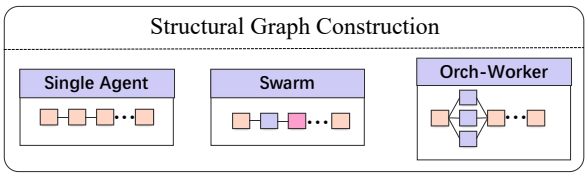


Figure 6: Structural graph topologies produced by segment partitioning under three agent architectures. Each box is a segment; edges encode inter-segment dependencies. Single-agent traces yield a linear chain; swarm traces yield a chain with heterogeneous agent segments; orchestrator-worker traces yield a branching fan-out/fan-in structure.

C Global Calibration Details

Because local dependency characterization analyzes each segment in isolation, it is prone to three biases that the global calibrator corrects. *Repetition inflation*: a single recurring mistake is recorded as a separate issue in every segment it appears in, over-counting its contribution. *Environmental contamination*: a failure caused by the environment—such as a tool error or an unavailable resource—is misattributed to an agent decision. *Latent failure omission*: a fault whose relevance is only apparent once the final task outcome is known is missed by the local pass entirely.

The global calibrator revisits all segment-level signals jointly with the final outcome: it merges duplicate issues, discards environment-induced ones, adds latent issues surfaced by the outcome, and adjusts issue scores. It does not compute attribution scores itself; its output is a corrected set of issues and attribution dependencies, passed to the deterministic scoring in §3.5. This separation confines stochastic judgment to graph and signal extraction, so that the final scores remain reproducible once the calibrated graph is fixed.

Attribution Graph	Hit@3	F1
Random graph	56.42	25.38
$\mathcal{G}_{\text{struct}}$ (no inference)	66.32	31.58
Reversed $\mathcal{G}_{\text{attr}}$	51.27	22.83
$\mathcal{G}_{\text{attr}}$ (DEPTRACER, Ours)	**77.62**	**38.30**

Table 8: Attribution graph quality ablation on CodeTraceBench. Phase 3 propagation is held fixed; only the input graph varies. *Random*: edges sampled uniformly at random, matching $|\mathcal{G}_{\text{attr}}|$ in edge count. $\mathcal{G}_{struct}$: structural graph passed directly to Phase 3 without dependency inference. *Reversed*: all edges in $\mathcal{G}_{\text{attr}}$ reversed.

D CodeTraceBench Data Filtering

CodeTraceBench releases approximately 1,000 code-agent trajectories, each annotated with erroneous steps (`incorrect_step_ids`) and per-step labels (`step_label`). During preprocessing we found that many released instances are internally inconsistent in one of two ways.

Step-count mismatch. Parsing a trajectory with the benchmark's own released parser yields a total step count that does not agree with the annotation—for instance, `incorrect_step_ids` that index steps outside the parsed trajectory. The annotated step indices then cannot be aligned to actual steps.

Label-content mismatch. For some instances the `step_label` annotation describes content that does not correspond to the actual content of the step it points to, so the label cannot be taken as reliable ground truth for that step.

Either inconsistency makes step-level evaluation unreliable. We therefore filter CodeTraceBench to instances whose trajectories, parser-derived step counts, and step labels are mutually consistent, yielding 135 evaluable instances. This filter uses only the benchmark's own released data and parser; it is purely structural and is applied before and independently of any attribution method, so it does not favor DEPTRACER over the baselines.

E Attribution Graph Quality

To verify that the attribution dependency graph $\mathcal{G}_{\text{attr}}$ is the source of DEPTRACER's gains rather than the scoring algorithm alone, we hold Phase 3 fixed and vary only the input graph. Table 8 reports Hit@3 and F1 on CodeTraceBench.

Using a *random graph* (edges sampled uniformly to match $|\mathcal{G}_{\text{attr}}|$ in edge count) degrades performance to 56.42 Hit@3, confirming that graph structure—not merely the presence of edges—matters. Passing the structural graph $\mathcal{G}_{\text{struct}}$ directly

α_{OU}	α_b	γ	κ	**Hit@3**	**F1@3**
0.3	0.3	0.5	0.5	77.62	38.30
0.0	0.3	0.5	0.5	67.21	31.58
0.6	0.3	0.5	0.5	74.18	36.29
0.3	0.15	0.5	0.5	73.82	36.48
0.3	0.6	0.5	0.5	72.58	35.47
0.3	0.3	0.3	0.5	74.63	36.18
0.3	0.3	0.5	0.0	72.96	36.04

Table 9: Hyperparameter sensitivity. α_{OU}: output-utilization addend; α_b: attribution transfer ratio; γ: received-attribution cap; κ: output-utilization cap. First row is the default configuration.

to Phase 3 without LLM-based dependency inference yields 66.32, substantially below $\mathcal{G}_{\text{attr}}$'s 77.62, showing that the attribution-relevant edges inferred in Phase 2 are critical. *Reversed* $\mathcal{G}_{\text{attr}}$ (all edges flipped) collapses to 51.27, confirming that backward propagation direction is not arbitrary but must follow the actual failure-propagation direction.

F Hyperparameter Sensitivity

Table 9 reports Hit@3 and F1 on CodeTraceBench as each hyperparameter is varied in isolation from its default value. The default configuration (α_{OU}=0.3, α_b=0.3, γ=0.5, κ=0.5) achieves the highest Hit@3 of 77.62 and F1 of 38.30; all single-parameter deviations reduce performance, confirming that the default is a stable operating point rather than an artifact of narrow tuning.

G Licenses

Released artifact. Our code is available anonymously for review; the license will be determined upon de-anonymization.

Third-party benchmarks. CodeTraceBench (Li et al., 2026), Who&When (Zhang et al., 2025b), GAIA (Mialon et al., 2024), and TerminalBench 2.0 (Merrill et al., 2026) are each released under the MIT License. We use them solely for evaluation and make no modifications to the original data.

Model APIs. We access Claude-Sonnet-4 (Anthropic, 2025), Qwen3-Max (Qwen Team, 2025), and DeepSeek-V4-Flash (DeepSeek-AI, 2026) through their respective commercial APIs in accordance with each provider's terms of service. No model weights are redistributed.

H Use of AI Assistants

The authors used GPT-5.5 to assist with academic writing (e.g., sentence-level polishing and phrasing) and Claude Code to assist with research coding and implementation. All research ideas, methodological innovations, experimental designs, and scientific conclusions are entirely the authors' own.

I Prompts

We provide the three core prompts used by DEPTRACER. Placeholders of the form `{variable}` are filled at runtime.

Part 1: Per-Segment Analysis

This prompt is applied independently to each segment. The model produces a step-by-step action summary, an integrity score $I \in \{0, 0.5, 1\}$, a list of issues, and a list of artifacts. Prior-segment context (`previous_artifacts`, `previous_issues`) is passed in so the model can track cross-segment artifact usage and issue resolution.

Per-Segment Analysis Prompt

```
Note: you are analyzing one segment of a larger
      trajectory; the
overall task may have failed, and you only see part of
      the execution.
Task: <task_description>{task_description}</
      task_description>
Dialogue log:
<full_session_context>{full_session_context}</
      full_session_context>
Prior artifacts: <previous_artifacts>{previous_artifacts}
</previous_artifacts>
Prior issues: <previous_issues>{previous_issues}</
      previous_issues>

Complete the following four analyses:

**1. Work summary**
One sentence + per-step breakdown. Each step: [Action(
      type)] +
combined content + result. run_command, create_file,
      modify_file,
read_file, install_dep, other.

**2. Integrity review** (I claims completion but artifact
       absent;
badly wrong direction.)
- I=1: entirely resolved error; I=0: reasonable attempt,
      no problem.
- I=0.5: silent failure; code written but not verified;
      partially
  wrong direction.
Consistency rule: if I=0, integrity_related_steps must be
       [].

**3. Artifacts and issues**
- Artifacts: created files and key conclusions.
- Issues: only "did something and did it wrong" (error/
      bug/wrong
  direction); required when I > 0.
- solved_issues / used_artifacts: select from prior-
      segment lists.

Output all tags:
<summary_action>one-sentence summary</summary_action>
<summary_steps>[{"step": 0, "action": "run_command: ls /
      app,
  confirmed directory layout"}]</summary_steps>
<integrity_explanation>reasoning</integrity_explanation>
<behavior_integrity>0</behavior_integrity>
<integrity_related_steps>[]</integrity_related_steps>
<summary_artifacts>{"created": [{"name": "/app/solve.py",
```

```
  "related_steps": [1]}], "conclusions": []}</
        summary_artifacts>
<summary_issues>[{"issue": "solve.py raises ImportError",
  "caused_by": [], "confidence": 1, "related_steps":
        [2]}]
</summary_issues>
<solved_issues>[]</solved_issues>
<used_artifacts>[]</used_artifacts>
```

Part 2: Cross-Segment Attribution Calibration

After all segments are analyzed independently, this prompt reviews their integrity scores and issue lists jointly with the final task outcome and makes three types of corrections: (1) reduce I for earlier redundant segments performing the same work, (2) clear issues caused by environmental factors outside the agent's control, and (3) raise I to 1 and inject a `[CROSS_SESSION]` issue for segments that produced core artifacts in a failed task.

Cross-Segment Calibration Prompt

```
You are a cross-segment attribution analyst.
Review all segments' integrity scores (I) and issues for
      correctness,
then output adjustments.

Task: <task_description>{task_description}</
      task_description>
Agent architecture: <mas_description>{mas_description}
</mas_description>
All segment summaries: <all_sessions_summary>{
      all_sessions_summary}
</all_sessions_summary>
Final task result: <task_result_info>{task_result_info}
</task_result_info>

Make three types of adjustments:
**1. Redundant work -- lower I, clear issues**
If multiple segments repeat the same operation with the
      same method,
lower I for earlier duplicates and set clear_issues=true.
Different methods = not redundant.

Output: <integrity_adjustments>[
  {"session_index": "2", "adjusted_integrity": 0.5,
   "reason": "repeats s0 with same method",
   "integrity_related_steps": [0,2], "clear_issues": true
         },
  {"session_index": "3", "adjusted_integrity": 1,
   "reason": "[CROSS_SESSION] created core script; task
         failed",
   "integrity_related_steps": [2,3],
   "new_issue": "[CROSS_SESSION] Step 2 created solution.
         py;
     task failed; implementation likely contains a bug"}
]</integrity_adjustments>
Rules: adjusted_integrity in {0, 0.5, 1}; list only
      segments that
need adjustment; mark at most 1-3 key steps in
integrity_related_steps.
```

Part 3: Attribution-Guided Feedback Generation

Given the calibrated attribution scores, the high-risk steps flagged by the scoring algorithm are passed to this prompt, which generates concise reflection hints for the next optimization round. Each hint must identify where to reconsider, which task constraint to re-read, and how to verify correctness—without providing specific answers.

Feedback Generation Prompt

```
You are a coach for an AI agent. The agent attempted a
      task and
failed; help the next attempt avoid the same mistakes.
Task description: {task_description}
Agent execution log: {summary_steps}
High-risk steps flagged by the attribution system:
(These steps are identified as likely responsible for
      failure;
attribution may be imperfect -- use as reference only.)
{problem_steps}

Generate 2-4 reflection hints for the next agent. Each
      hint must:
1. [Locate] what the agent was doing when it likely went
      wrong
   (describe behavior, not step numbers).
2. [Re-examine] which sentence or constraint in the task
      description
   to re-read.
3. [Verify] how to confirm the action is correct this
      time (write a
   check script / compare format / enumerate constraints).


Do NOT: give specific correct values; make absolute
      judgments
("the previous answer was wrong"); provide a replacement
      solution
for a methodological deviation -- instead note that "the
      core logic
based on X may have a flaw."

Output 2-4 hints, each under 100 words, in question or
      reminder form.
<case_feedback>
(output your 2-4 reflection hints here)
</case_feedback>
```