

CodeTracer: Towards Traceable Agent States

Han Li^{1*} Yifan Yao^{1*} Letian Zhu^{1*} Rili Feng^{1*} Hongyi Ye¹ Jiaming Wang¹
Yancheng He¹ Pengyu Zou¹ Lehan Zhang⁵ Xinping Lei¹ Haoyang Huang²
Ken Deng² Ming Sun² Zhaoxiang Zhang³ He Ye⁴ Jiaheng Liu^{1†}

¹ Nanjing University ² Kuaishou Technology ³ Institute of Automation, CAS
⁴ University College London ⁵ Renmin University of China

han.li.cs@smail.nju.edu.cn, liujiaheng@nju.edu.cn

Abstract

Code agents are advancing rapidly, but debugging them is becoming increasingly difficult. As frameworks orchestrate parallel tool calls and multi-stage workflows over complex tasks, making the agent’s state transitions and error propagation hard to observe. In these runs, an early misstep can trap the agent in unproductive loops or even cascade into fundamental errors, forming hidden error chains that make it hard to tell *when* the agent goes off track and *why*. Existing agent tracing analyses either focus on simple interaction or rely on small-scale manual inspection, which limits their scalability and usefulness for real coding workflows. We present CODETRACER, a tracing architecture that parses heterogeneous run artifacts through evolving extractors, reconstructs the full state transition history as a hierarchical trace tree with persistent memory, and performs **failure onset localization** to pinpoint the failure origin and its downstream chain. To enable systematic evaluation, we construct CODETRACEBENCH from a large collection of *executed* trajectories generated by four widely used code agent frameworks on diverse code tasks (e.g., bug fixing, refactoring, and terminal interaction), with supervision at both the stage and step levels for failure localization. Experiments show that CODETRACER substantially outperforms direct prompting and lightweight baselines, and that replaying its diagnostic signals consistently recovers originally failed runs under matched budgets. Our code and data are publicly available.¹

1 Introduction

Large language models (LLMs) are increasingly used to power code agents that autonomously interact with software repositories and development environments (Yang et al., 2024; Wang et al., 2025; 2024; Xia et al., 2025). These agents execute long sequences of heterogeneous actions—searching code, reading files, editing implementations, running builds, and interpreting test feedback—to solve complex engineering tasks such as repository-level bug fixing or system configuration (Jimenez et al., 2024; Merrill et al., 2026). As the capability of such agents improves, their executions grow longer and more complex, making understanding failures increasingly difficult. For example, when an agent run fails, it is often unclear where the trajectory first went wrong or which intermediate decisions caused the final failure.

Current evaluation practices provide only limited visibility into these processes. Most benchmarks summarize agent performance using end-to-end metrics such as pass rate or patch correctness, collapsing entire trajectories into a single success or failure label (Jimenez et al., 2024; Merrill et al., 2026; Xia et al., 2025). Existing analyses of agent trajectories either rely on coarse outcome-level judgments or manual inspection of small numbers of runs, making them difficult to scale to realistic software engineering workloads with long execution traces (Bouzenia and Pradel, 2025; Kuang et al., 2025).

In this work, we introduce CODETRACER, a framework for converting heterogeneous agent run directories into structured hierarchical traces and automatically identifying the earliest failure critical stage in a trajectory. The system can also produce actionable debugging signals that can be fed back into the agent through reflective replay, enabling targeted recovery from earlier mistakes (Shinn et al., 2023; Madaan et al., 2023; Chen et al., 2024). To support systematic evaluation, we construct CODETRACEBENCH, a high-quality benchmark of code agent trajectories with step-level annotations. The benchmark aggregates trajectories generated by multiple widely used agent frameworks across diverse software engineering workloads, including repository level bug fixing and long horizon terminal interaction tasks (Jimenez

*Equal contribution.

†Corresponding author.

¹<https://github.com/NJU-LINK/CodeTracer>

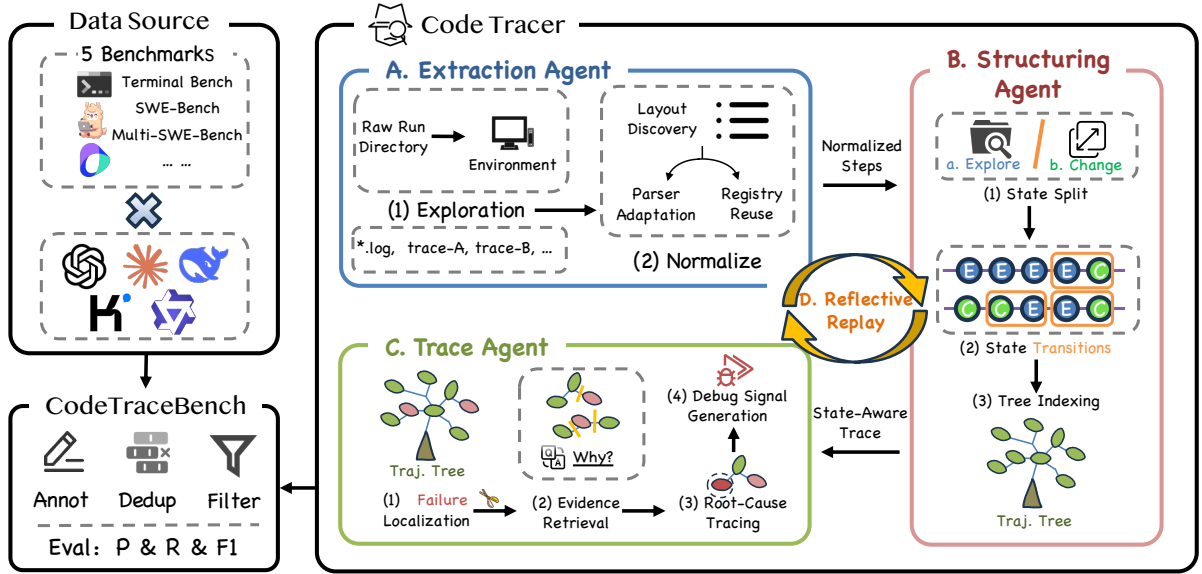


Figure 1: **Overview of the CODETRACER pipeline.** Raw trajectories are standardized into hierarchical traces, curated into CODETRACEBENCH with step-level supervision, and diagnosed via failure onset localization with optional reflective replay.

et al., 2024; Yang et al., 2024). Each trajectory is annotated with structured step metadata and failure critical labels, enabling evaluation of both stage level localization and evidence retrieval.

Using this benchmark, we conduct a large-scale empirical study of code agent behavior across multiple model backbones and agent frameworks. Our analysis reveals several systematic patterns, including an evidence-to-action gap, where agents often retrieve relevant information but fail to translate it into correct state changing actions; significant variance in action efficiency across trajectories even for strong models; and stage-dependent error modes that concentrate failure-critical decisions in specific phases of the workflow.

Overall, the contributions of this work are as follows: (1) CODETRACER, a scalable framework for hierarchical trajectory tracing and failure onset localization in code agent executions. (2) CODETRACEBENCH, a benchmark of thousands of annotated code agent trajectories enabling systematic evaluation of process level diagnosis. (3) A large-scale empirical analysis of failure patterns in modern code agents, revealing actionable insights into their reasoning and execution behavior.

2 Trajectory Analysis

2.1 Trajectory Collection

We broadly collect execution trajectories from benchmarks that evaluate code generation and software engineering capabilities, covering both repository-level bug fixing (*SWE-bench Verified*, *SWE-bench Pro*, *MultiSWE-bench*, *SWE-PolyBench*) and long horizon terminal interaction (*TerminalBench*) (Yang et al., 2024; Jimenez et al., 2024; Wang et al., 2025; Merrill et al., 2026). Each benchmark is executed under four widely used agent frameworks (*SWE-Agent*, *MiniSWE-Agent*, *OpenHands*, *Terminus 2*) (Yang et al., 2024; Wang et al., 2025; Merrill et al., 2026) paired with five model backbones (*Claude-sonnet-4*, *GPT-5*, *DeepSeek-V3.2*, *Qwen3-Coder-480B*, *Kimi-K2-Instruct*) (DeepSeek-AI et al., 2025; Anthropic, 2025; Team et al., 2026; Qwen Team, 2025; OpenAI, 2025), yielding a diverse corpus that spans different task regimes, orchestration strategies, and backbone capabilities.

2.2 Filtering

To obtain a consistent unit of progression across heterogeneous frameworks, we normalize each run into *iterations* defined by executed commands and apply quality-based filtering in this normalized space. Starting from 7,936 raw trajectories, we apply four successive filters: (i) removing runs that time out before completion (retaining 6,511); (ii) removing runs with incomplete or truncated generation traces (6,109); (iii) removing runs whose Docker environment is misconfigured or whose task files are corrupted,

Table 1: End-to-end success versus process cost on the no intersection setting. Tok.(k) is the average token consumption per task in thousands. SWE-Agent and OpenHands consume $2\times$ the tokens of MiniSWE-Agent for only modest success gains.

Agent	Succ. (%)	Steps	Avg. Steps	Tok.(k)
MiniSWE-Agent	32.8	18,792	19.9	44.6
Terminus 2	35.2	25,381	24.7	51.3
SWE-Agent	37.5	43,216	41.2	86.7
OpenHands	38.3	46,754	43.9	91.4

rendering the execution output unreliable (5,284); (iv) removing *correct* trajectories with fewer than 10 normalized steps, which typically complete trivially and provide limited signal for failure analysis (3,326). After filtering, we retain 3,326 trajectories spanning all benchmark–framework–backbone combinations.

2.3 Annotation

All annotations are performed by the authors of this paper. Each annotator is assigned a set of *tasks* together with *all* corresponding trajectories across the full backbone–agent grid (15 groups per task), and is provided with the task specification, the reference solution, and access to the execution environment for manual verification of ambiguous cases (e.g., environment or configuration issues). This allocation allows each annotator to build deep familiarity with the task, facilitates horizontal comparison of how different models and agents approach the same problem, and enables efficient deduplication of similar failure modes.

For every trajectory, the annotator assigns a *stage label* to each step, indicating the phase of the workflow (environment verification, dependency installation, inspection/debugging, patching, verification). For *successful* trajectories (i.e., those that pass all tests), the annotator further identifies *redundant* and *trial-and-error* steps—actions that do not contribute to the final solution or that are later reverted. For *failed* trajectories, we adopt a *chain based backward tracing* protocol: starting from the failing test output, the annotator identifies the immediately preceding step whose output or action produced the observed error and recursively traces upstream, asking what earlier decision led to this intermediate failure, until either (i) the preceding steps contain no error, or (ii) the failure cause is unrelated to earlier trajectory decisions. Each chain terminates at an *error critical step*—the earliest decision that triggers the downstream cascade (Lightman et al., 2024; Zheng et al., 2025; Kuang et al., 2025)—and receives an *error type label* from a controlled vocabulary (environment/setup issues, dependency resolution failures, mislocalized edits, incorrect hypotheses, verification misinterpretation, and unproductive looping). To assess annotation reliability, a random 15% subset of trajectories was independently double annotated; interannotator agreement on the error-critical step label reached Cohen’s $\kappa = 0.73$. The full annotation guidelines, including edge case handling, are provided in Section A.

The annotation process surfaces several recurring patterns across backbones and agent frameworks that motivate the design of CODETRACER.

Model preferences and shared failure modes. We analyze per-category pass rates across all 15 backbone–agent groups (Figure 2). Among 340 task categories, the 66 categories solved by all five models are mostly routine transformation and scripting workloads, such as regex manipulation, JSON/CSV processing, and standard numerical procedures. At the other extreme, the 65 categories unsolved by every model skew toward tasks that require harder external grounding or longer horizon reasoning, including formal verification, computer vision, advanced scientific computing, and legacy environments. Between these two poles, the strongest backbones remain broadly competitive but exhibit clear task preferences: GPT-5 is relatively stronger on graph heavy, chemistry, and forensics tasks; Claude-sonnet-4 on Bayesian inference and speculative decoding; Kimi-K2-Instruct on graphics and ray tracing; and DeepSeek-V3.2 on data pipeline and package management workloads. When none of the models can truly solve the task, their behaviors become strikingly similar: instead of explicitly recognizing failure, they often bypass the bottleneck through fabricated evidence, placeholder outputs presented as real results, or premature stopping after becoming trapped in unproductive loops.

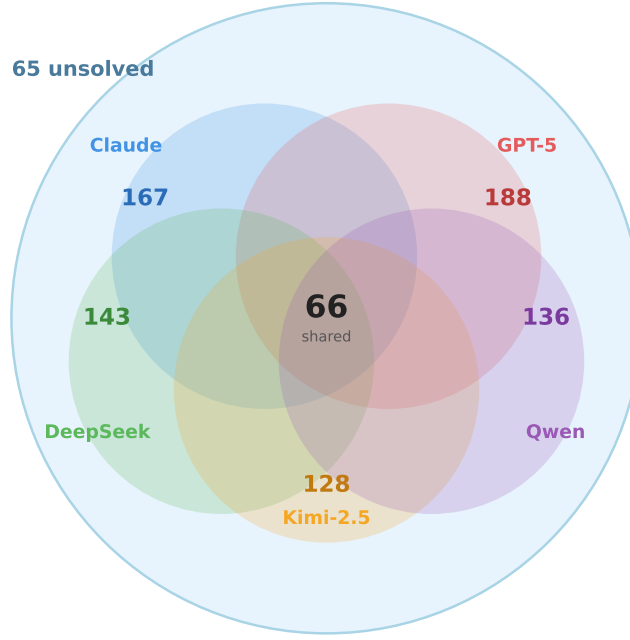


Figure 2: Task categories solved per backbone. The central 66 categories are solved by all five models; 65 categories remain unsolved by any model.

Takeaway 1 for Task Preferences and Failure Masking Behaviors

Backbones differ more in *which* task regimes they solve reliably than in aggregate solved count, but on universally hard categories they converge to the same failure pattern: bypassing the problem through fabrication, placeholder outputs, or early stopping (Figure 2).

Overengineering: when complexity does not buy success. We compare a lightweight agent design (*MiniSWE-Agent*) with progressively more complex frameworks (*Terminus 2*, *SWE-Agent*, *OpenHands*) along two axes: end-to-end task success and *process cost*, measured by both step count and token consumption (Table 1). *Terminus 2* adds moderate orchestration overhead over *MiniSWE-Agent* (51.3k vs. 44.6k tokens), while *SWE-Agent* and *OpenHands* nearly double token usage (86.7k and 91.4k). Yet these additional expenditures yield only modest success gains (+2.4–5.5 pp). This pattern suggests that, for general terminal-based coding tasks, success is primarily bounded by backbone capability, while architectural complexity often translates into higher cost and longer interaction loops rather than reliably better outcomes.

Takeaway 2 for Overengineering

Additional orchestration increases cost without proportional success gains; backbone capability is the primary lever (Table 1).

Stage dynamics, failure onset, and error type shift. Using the stage labels assigned during annotation (environment verification, dependency installation, inspection/debugging, patching, and verification), we examine how stage occupancy and transitions relate to end-to-end success. Solved runs display a more coherent, forward moving progression with fewer back-and-forth oscillations. In contrast, unsolved runs allocate disproportionate budget to early setup activities and recurrent inspection loops, often after an early wrong commitment that later steps fail to undo. Error types also shift systematically across stages: environment and dependency errors concentrate early, while mislocalized edits, incorrect hypotheses, and verification misinterpretation dominate later patching and verification. The concentration of error critical steps in a small set of stages (Figure 3) directly motivates our failure onset localization objective.

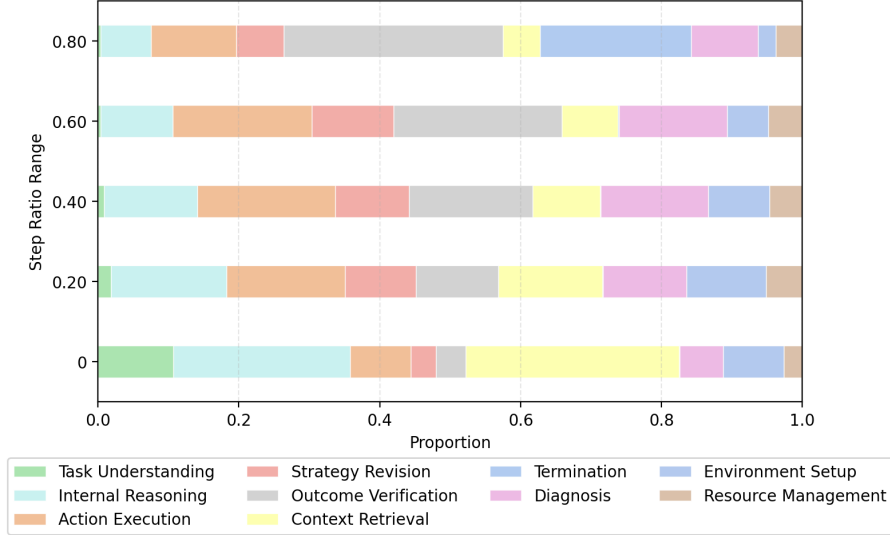


Figure 3: Distribution of error critical steps across stages, contrasting solved and unsolved runs.

Table 2: **Resolved rate (%) and token consumption (in thousands) across iteration budgets and backbones.** Tok.(k) reports average token usage within the first `max_iter` rounds over all valid samples with extractable prefix token traces. Token coverage by model and budget is reported separately for auditability.

max_iter	Claude-sonnet-4		GPT-5		DeepSeek-V3.2		Qwen3-Coder-480B		Kimi-K2-Instruct	
	Res.%	Tok.(k)	Res.%	Tok.(k)	Res.%	Tok.(k)	Res.%	Tok.(k)	Res.%	Tok.(k)
10	12.67	116.98	38.69	106.19	12.43	86.79	9.76	99.38	10.61	78.81
20	30.13	266.53	45.48	163.89	27.50	212.60	18.22	255.66	23.02	192.28
40	38.00	493.46	47.06	214.60	38.18	400.96	28.85	543.64	29.14	402.20
100	41.65	767.77	47.06	249.80	39.75	479.77	31.67	970.46	32.55	715.24
≥ 150	41.65	886.00	47.06	266.32	39.75	512.42	32.10	1182.06	33.27	805.38

Takeaway 3 for Stage Dependent Errors

Error type is largely determined by workflow phase: environment and dependency errors dominate early stages, while mislocalized edits and incorrect hypotheses concentrate in later patching and verification. This predictability enables stage-aware guardrails that can preempt failures before they cascade, rather than relying solely on post-hoc diagnosis (Figure 3).

Success saturates with additional iterations. We sweep `max_iterations` over $\{5, \dots, 300\}$ and reevaluate every backbone–framework pair under each budget. Table 2 reports the resolved rates and token usage aggregated at the backbone level, while the full 5×3 per combination trajectories are provided in Figure 10. Two consistent patterns emerge. First, success improves rapidly as the iteration budget increases to around 40 steps, but the gains quickly diminish thereafter and the curves largely flatten. Second, the *saturation ceiling* is mainly backbone dependent: stronger models plateau at higher resolved rates, but not substantially later. This suggests that extra iterations mainly help recover from low budget underexploration, rather than fundamentally improving the agent’s reasoning ability. When an agent commits early to an incorrect hypothesis, additional iterations are often spent on redundant exploration and trial-and-error edits rather than correcting the underlying mistake.

Takeaway 4 for Iteration Budget

Success saturates quickly; extra iterations amplify unproductive loops rather than enabling recovery (Table 2).

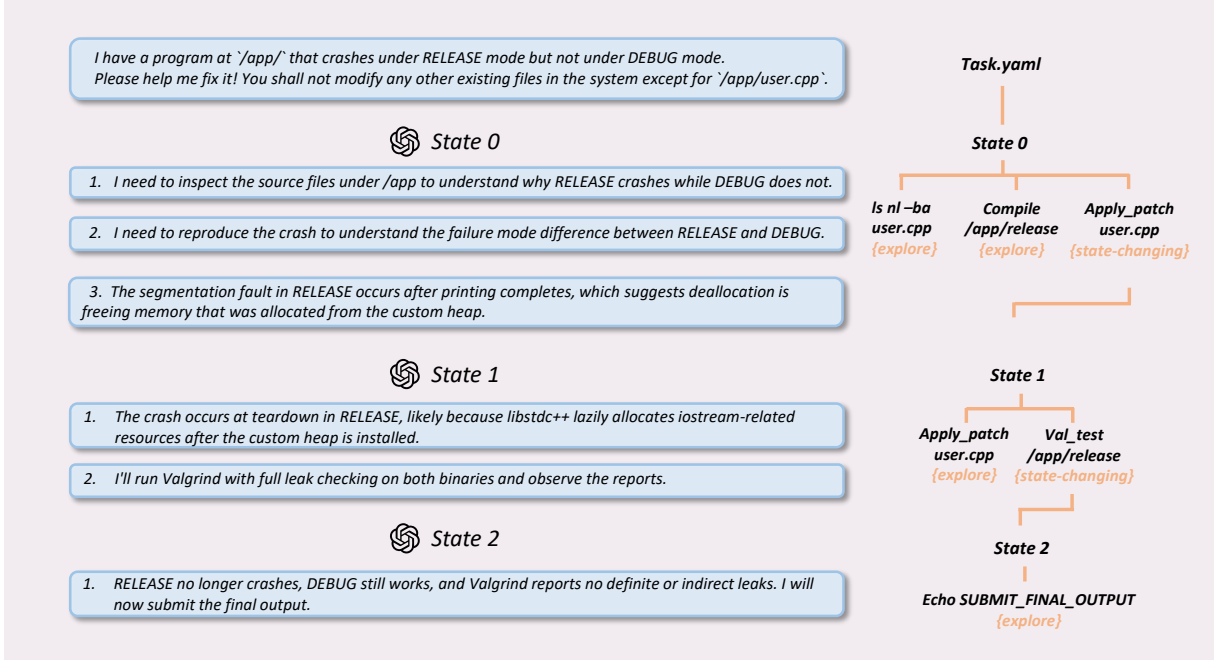


Figure 4: Hierarchical trace tree. Exploration steps remain under the current state node, whereas state changing steps induce transitions to child states. CODETRACER builds this structure during tree indexing to provide compressed navigation of execution traces.

3 CodeTracer

3.1 Method

CODETRACER tackles **failure onset localization**: given a standardized trajectory τ segmented into ordered stages, it predicts a failure responsible stage $\hat{s}$, a set of error relevant steps P within $\hat{s}$, and a compact evidence set E supporting the diagnosis. Supervision marks stages that are causally responsible for the eventual failure under our annotation guidelines, and evaluation measures step level Precision/Recall/F1 against the gold incorrect step set G plus token cost under matched budgets.

Because run directories are heterogeneous and their file layouts are agent dependent, hardcoded parsers are brittle. CODETRACER therefore separates *evolving extraction*, *tree indexing*, and *diagnosis*, while accumulating reusable parsers so that support for new formats improves over time, following the broader need for process-aware trajectory analysis in software engineering agents (Bouzenia and Pradel, 2025; Kuang et al., 2025).

Pipeline. CODETRACER decomposes tracing into three stages (details in Section B): (1) **Evolving extraction** scans a run directory, produces a compact layout spec describing which artifacts record execution steps, and adapts a parser accordingly: it checks a registry of existing parsers; if no match is found, it synthesizes and registers a new one, then instantiates a format-specific diagnosis prompt with a fixed query schema. This stage emits normalized step records with typed fields (action, observation, diff, verification outcome) while preventing format drift across runs. (2) **Tree indexing** converts the flat sequence of normalized steps into a *hierarchically structured* trace tree (Figure 4). Steps that only inspect the current environment without modifying the codebase or execution state are represented as exploration nodes under the same state, whereas steps that modify the codebase or execution environment are represented as state-changing nodes that induce transitions to child states. Each node is further annotated with a summary of intent and outcome. The resulting tree makes explicit whether subsequent actions are taken under an unchanged context or after an intervention, and serves as a compressed navigation index for diagnosis. (3) **Diagnosis** traverses the tree, issues structured evidence queries, and outputs the failure responsible stage, error-relevant steps, and a compact evidence set justifying the localization.

3.2 CodeTraceBench

We construct CODETRACEBENCH directly from the trajectory corpus in Section 2. The source pool spans five benchmarks (SWE-bench Verified, SWE-bench Pro, MultiSWE-bench, SWE-PolyBench, and Terminal-

Table 3: **Main localization results on CodeTraceBench (macro, step level) stratified by difficulty.** P/R/F1 are reported as percentages (%). Tok denotes total LLM tokens (in thousands). Overall uses the full evaluation set; Easy/Medium/Hard split by trajectory difficulty.

Backbone	Overall				Easy				Medium				Hard			
	P	R	F1	Tok	P	R	F1	Tok	P	R	F1	Tok	P	R	F1	Tok
Bare LLM																
CLAUDE-SONNET-4	16.64	15.82	16.22	105.1	18.31	17.08	17.67	68.5	16.07	15.14	15.59	97.8	13.52	12.83	13.17	155.6
GPT-5	16.69	21.46	18.78	58.5	18.42	23.54	20.65	38.2	16.14	20.73	18.13	54.5	16.94	17.42	17.18	86.6
DEEPSEEK-V3.2	13.11	21.66	16.33	83.4	14.37	23.72	17.87	54.6	12.58	20.74	15.67	77.5	10.28	17.95	12.89	123.6
Mini-CodeTracer																
CLAUDE-SONNET-4	20.50	26.24	19.17	82.4	23.58	28.46	22.04	54.2	19.83	29.17	18.42	77.1	15.47	20.83	14.08	120.6
GPT-5	26.03	21.39	19.33	44.8	28.71	24.26	22.31	29.5	25.42	20.63	18.68	42.0	20.73	16.18	14.17	65.8
DEEPSEEK-V3.2	24.08	20.98	19.24	63.8	27.14	23.62	21.68	42.0	23.24	19.87	18.43	59.5	18.27	16.41	14.62	94.2
CodeTracer																
CLAUDE-SONNET-4	40.47	54.87	46.57	56.8	45.28	59.13	51.29	37.6	39.36	53.58	45.38	53.4	32.71	47.24	38.67	82.5
GPT-5	45.02	51.46	48.02	31.1	49.86	55.78	52.68	20.6	43.81	50.42	46.84	29.2	37.08	43.79	40.14	45.7
DEEPSEEK-V3.2	43.17	49.58	46.14	44.6	47.68	53.94	50.62	29.5	42.02	48.53	45.04	41.8	35.74	42.21	38.72	65.3

Bench), five backbones (Claude-sonnet-4, GPT-5, Kimi-K2-Instruct, Qwen3-Coder-480B, DeepSeek-V3.2), and four frameworks (SWE-Agent, MiniSWE-Agent, OpenHands, Terminus 2), totaling 4,354 standardized and step-level annotated trajectories (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025). From this pool, we curate long horizon instances with clear failure cascades and sufficient in-trace evidence, while removing short or near-duplicate runs.

The resulting benchmark is produced with a full split (3.32K instances) and a higher quality verified split (1.06K). Each instance records the framework, backbone, task metadata (236 tasks, 26 categories, difficulty labels), raw artifact pointers, stage boundaries, failure critical stage labels, and incorrect step annotations, supporting stage localization and within stage evidence retrieval.

4 Experiments

4.1 Setup

We evaluate CodeTraceBench using the intersection subset (backbone–method pairs with complete coverage). We test Claude-sonnet-4, GPT-5, and DeepSeek-V3.2 under three localization methods (Bare LLM, MINI-CODETRACER, CODETRACER), all operating on the same run directories under matched budgets and unified decoding settings. We report macro averaged step level Precision, Recall, and F1 (formal definitions in Section C) and total token usage.

4.2 Main Results

Table 3 summarizes the main localization results. Across all backbones, adding structure and tracing signals substantially improves localization quality over raw log prompting. MINI-CODETRACER already recovers part of the gain, showing that lightweight standardization is a strong baseline, while CODETRACER further improves both precision and recall by prioritizing genuinely failure relevant steps rather than merely salient artifacts. Token efficiency is also improved because tracing narrows evidence retrieval to compact candidate sets. The three frontier backbones reach comparable F1 (46–48%) but diverge in how they traverse the trace. GPT-5 terminates its search earlier, committing to a compact set of failure relevant steps with high confidence; this yields the best precision (45.0% overall, 49.9% Easy) and the lowest token cost (31.1k overall, 20.6k Easy) but misses some error contributing steps. Claude-sonnet-4 continues scanning deeper into the trace before concluding, surfacing more failure relevant evidence and achieving the highest recall across all splits (54.9% overall, 59.1% Easy) at the expense of higher token consumption (56.8k) and lower precision (40.5%). DeepSeek-V3.2 falls between the two, balancing coverage and conciseness with the most uniform P/R gap across difficulty levels. Harder tasks correlate with longer trajectories and proportionally higher token budgets—the Easy to Hard token ratio roughly doubles for every backbone—confirming that localization difficulty scales with trajectory complexity. We further stratify results by difficulty, category, and execution stage (Appendix); the relative ordering among methods remains stable, with CODETRACER providing the largest gains on harder instances where failure relevant evidence is dispersed.

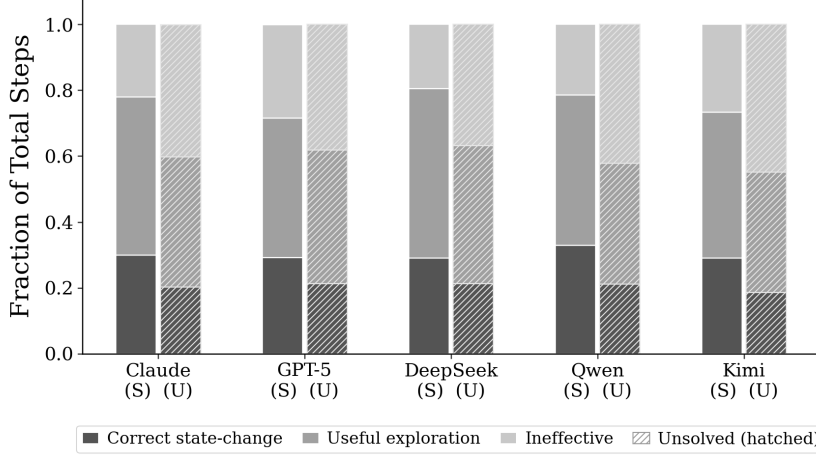


Figure 5: Step budget decomposition per backbone (solved vs. unsolved) on the intersection subset. Each bar shows the fraction of total steps that are correct state changes, useful exploration, or ineffective. Hatched bars denote unsolved trajectories.

Table 4: **Component ablation.** Each row incrementally adds one component on top of MINI-CODETRACER (diagnosis loop). Evolving extraction adds layout discovery and parser adaptation with accumulating reuse; tree indexing builds a hierarchical trace tree for compressed navigation. P/R/F1 are step level macro metrics (%); Tok is average tokens per instance (k).

Configuration	CLAUDE-SONNET-4				GPT-5				DEEPSEEK-V3.2			
	P	R	F1	Tok	P	R	F1	Tok	P	R	F1	Tok
Bare LLM	16.64	15.82	16.22	105.1	16.69	21.46	18.78	58.5	13.11	21.66	16.33	83.4
MINI-CODETRACER	20.50	26.24	19.17	82.4	26.03	21.39	19.33	44.8	24.08	20.98	19.24	63.8
+ Evolving Extraction	27.83	33.45	28.12	71.0	31.56	29.87	29.45	37.9	29.74	28.91	28.38	55.3
+ Tree Indexing	40.47	54.87	46.57	56.8	45.02	51.46	48.02	31.1	43.17	49.58	46.14	44.6

Takeaway 5 for Diagnosis Behaviors

Frontier backbones achieve similar F1 but exhibit different search depth strategies: GPT-5 commits early for higher precision at lower cost, Claude-sonnet-4 searches exhaustively for higher recall, and DeepSeek-V3.2 balances both. Localization cost scales with trajectory length, not model identity (Table 3).

4.3 Analysis and Replay

We conduct analyses using CODETRACER’s hierarchical trace structure and diagnostic outputs.

Evidence to action gap. Figure 5 decomposes each trajectory’s step budget into correct state changes, useful exploration, and ineffective steps. Across all five models the ineffective fraction nearly doubles from solved (22%) to unsolved (40%), while correct state changes drop consistently (30%→21%). Exploration usefulness degrades only mildly, indicating that agents still gather relevant information but fail to translate it into correct actions—a comprehension bottleneck whose severity varies by backbone (Qwen3-Coder-480B and Kimi-K2-Instruct show the sharpest drops, $\Delta=11.7$ and 10.3 pp).

Takeaway 6 for Evidence to Action Gap

Agents often gather useful evidence through exploration, but fail to translate it into effective state changing actions (Figure 5).

Component ablation. We incrementally add CODETRACER’s two key components—evolving extraction (layout discovery + parser adaptation) and tree indexing—on top of MINI-CODETRACER (Table 4). Tree indexing drives the largest single gain (18.3 pt F1) by providing compressed hierarchical navigation,

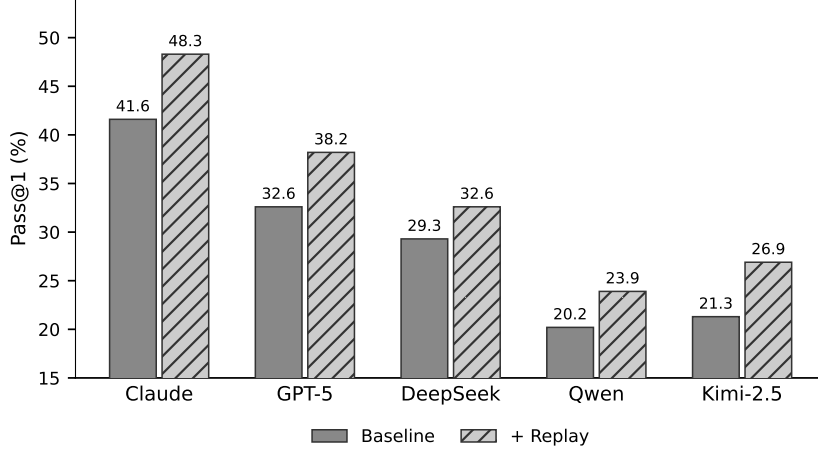


Figure 6: **Reflective replay.** Pass@1 on originally failed runs before and after injecting CODETRACER’s diagnostic signals under matched budgets.

while evolving extraction contributes a further 9.4-point lift through format standardization with parser reuse.

Reflective replay. We feed CODETRACER’s localized evidence back into agents: on originally failed runs, the same backbone is reinvoked under matched budget with the diagnosis injected as a prefix hint (Figure 6) (Shinn et al., 2023; Madaan et al., 2023; Chen et al., 2024). Replay consistently improves Pass@1 across all backbones, helping agents revise early wrong commitments instead of repeating unproductive exploration. The diagnosis pass itself consumes on average 8.4k tokens for Claude-sonnet-4, 5.2k for GPT-5, and 7.1k for DeepSeek-V3.2; these tokens are counted *outside* the replay budget so that the replayed run receives exactly the same iteration and token budget as the original.

Industrial agent analysis. Beyond the academic agent frameworks studied above, we apply CODETRACER to analyze trajectories from Claude Code, an industrial coding agent. Compared to research agents, Claude Code employs substantially richer tooling infrastructure (40+ specialized tools across 8 categories) and sophisticated context management (compaction, token budgeting), yielding a lower exploration-to-change ratio that correlates with higher trajectory efficiency; however, parallel tool execution—unique to industrial agents—introduces ordering-sensitivity issues absent from sequential academic frameworks. Full architectural comparison and trajectory statistics are provided in Section F.

Action efficiency. We define the *effective action ratio* as the fraction of goal-advancing steps per trajectory. Figure 7 shows that stronger models achieve higher means (73% for Claude-sonnet-4, 71% for GPT-5), yet all exhibit broad distributions with pronounced left tails—every backbone contains a nontrivial share of trajectories where fewer than half of actions are goal advancing.

Takeaway 7 for Action Efficiency

Even strong models execute many ineffective actions; improving early unproductive detection matters as much as adding steps (Figure 7).

5 Related Work

Code Agents and Software Engineering Benchmarks. Recent work has advanced software engineering agents in realistic development settings, including repository-level bug fixing and long-horizon terminal interaction. Representative systems and paradigms include SWE-Agent, OpenHands, CodeAct-style executable action agents, and Agentless, while evaluation has increasingly centered on executable benchmarks such as SWE-bench and Terminal-Bench (Yang et al., 2024; Wang et al., 2025; 2024; Xia et al., 2025; Jimenez et al., 2024; Merrill et al., 2026). However, these works primarily emphasize end-to-end task success, patch correctness, and resolved rates, offering limited visibility into where a long agent trajectory first becomes failure critical.

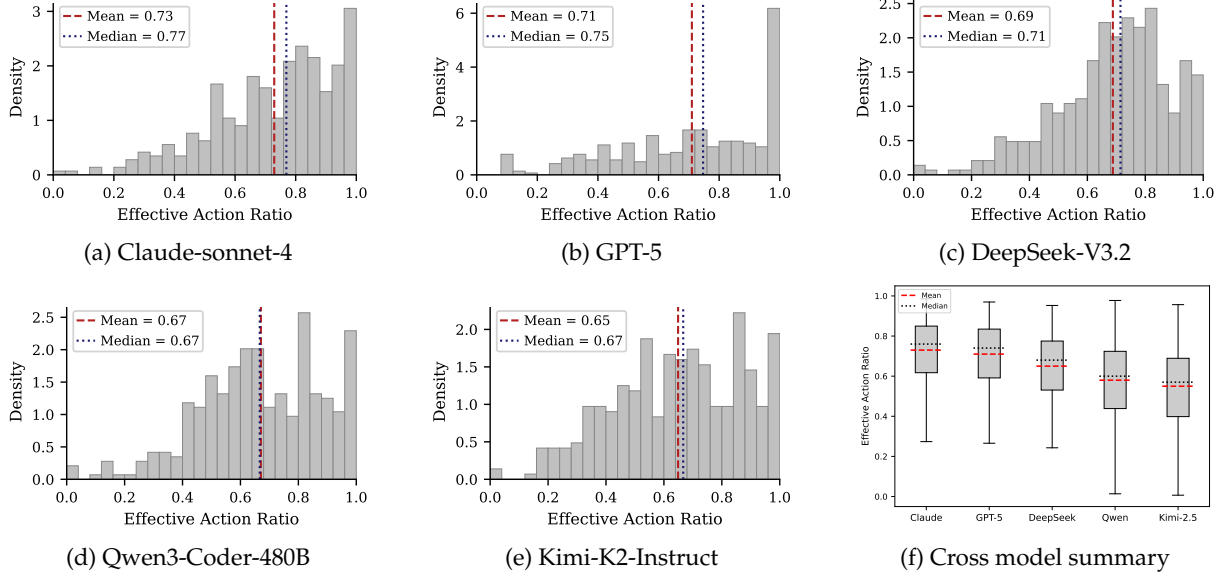


Figure 7: **Effective action ratio.** (a)–(e) per model histograms; (f) cross model violin and box summary.

Trajectory Error Localization, Debugging, and Replay. Recent work has also begun to study software engineering agent trajectories and process-level quality (Bouzenia and Pradel, 2025; Kuang et al., 2025). Related benchmark efforts in other domains further show that step-level supervision and process error identification can expose failures that outcome only evaluation misses (Lightman et al., 2024; Zheng et al., 2025). Our localization objective is also related to issue localization, classical fault localization, and LLM-based debugging: prior work has localized issue relevant code regions for software engineering tasks, ranked suspicious program elements from failing executions, and leveraged runtime traces to debug candidate programs (Xia et al., 2025; Jones and Harrold, 2005; Abreu et al., 2009; Zhong et al., 2024; Chen et al., 2024). Reflection based self-improvement methods such as Reflexion and Self-Refine further show how critique, feedback, and memory can improve subsequent attempts (Shinn et al., 2023; Madaan et al., 2023). In contrast, we focus on hierarchical trace standardization, stage level failure onset localization, evidence retrieval, and replay oriented diagnostic outputs on executed agent trajectories.

For completeness and reproducibility, the appendix summarizes the full annotation protocol, additional framework and metric details, and the complete prompts and output schemas used for diagnosis and replay, together with extended empirical results, including per backbone and per framework breakdowns across difficulty settings and iteration budgets, that support the main text findings.

6 Conclusion

We presented CODETRACER, a tracing framework that converts heterogeneous code agent logs into structured hierarchical traces and performs automated failure onset localization, together with CODETRACEBENCH, a benchmark of thousands of step level annotated trajectories spanning four agent frameworks, five frontier backbones, and diverse software engineering tasks. Our empirical study across this corpus revealed that agents frequently gather the right diagnostic evidence yet fail to act on it, that extra orchestration complexity and iteration budget yield diminishing returns once the backbone reasoning ceiling is reached, and that error types shift predictably across workflow stages. CODETRACER substantially outperforms raw log prompting and lightweight baselines in step level localization, reaching up to 48% macro F1 with lower token cost, and injecting its diagnostic outputs through reflective replay consistently recovers originally failed runs under matched budgets.

Limitations

Our study has several limitations. First, although CodeTraceBench is constructed from executed trajectories spanning multiple agent frameworks, backbones, and workloads, it does not cover the full design space of software engineering agents or real world repositories; conclusions about failure patterns and replay gains may therefore not transfer uniformly to other frameworks, domains, or deployment settings. Second, our supervision relies on stage and step level annotation of failure critical behavior. While our guidelines aim to make these judgments consistent, labels such as “incorrect,” “unuseful,” and “error critical” still involve annotator interpretation, especially for long trajectories with intertwined exploration and state changes. Third, our tracing and replay results are evaluated under matched offline budgets on previously collected runs. This setting isolates diagnostic quality, but it cannot fully capture how agents might adapt to interactive human oversight, changing environments, or repeated online intervention. Finally, our reflective replay experiments test whether localized evidence can help recover failed runs, but they do not by themselves establish a general training signal or guarantee robust improvement across all task categories and model families. We therefore view CODETRACER and CODETRACEBENCH as tools for structured diagnosis and controlled evaluation, rather than as a complete account of code agent reliability.

References

- Rui Abreu, Peter Zoetewij, Rob Golsteijn, and Arjan J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2009.06.035>. URL <https://www.sciencedirect.com/science/article/pii/S0164121209001319>. SI: TAIC PART 2007 and MUTATION 2007.
- Anthropic. Claude sonnet, 2025. URL <https://www.anthropic.com/claude/sonnet>. Accessed: 2026-03-17.
- Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2846–2857, 2025. doi: 10.1109/ASE63991.2025.00234.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=KuPixIqPiq>.
- DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruosheng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yudian Wang, Yue Gong, Yuhang Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni,

- Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL <https://arxiv.org/abs/2512.02556>.
- 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 B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.
- James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, page 273–282, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139934. doi: 10.1145/1101908.1101949. URL <https://doi.org/10.1145/1101908.1101949>.
- Jiayi Kuang, Yinghui Li, Xin Zhang, Yangning Li, Di Yin, Xing Sun, Ying Shen, and Philip S. Yu. Process-level trajectory evaluation for environment configuration in software engineering agents, 2025. URL <https://arxiv.org/abs/2510.25694>.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=v8L0pN6EOi>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.
- Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Jenia Jitsev, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=a7Qa4CcHak>.
- OpenAI. Introducing gpt-5, 2025. URL <https://openai.com/index/introducing-gpt-5/>.
- Qwen Team. Qwen3-coder-next technical report. Technical report, Alibaba Cloud, 2025. URL https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf. Accessed: 2026-02-03.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.
- Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jiale Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan,

- Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaying Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026. URL <https://arxiv.org/abs/2507.20534>.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232. PMLR, 21–27 Jul 2024. URL <https://proceedings.mlr.press/v235/wang24h.html>.
- 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 Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 65882–65919, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715754. URL <https://doi.org/10.1145/3715754>.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.
- Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. ProcessBench: Identifying process errors in mathematical reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1009–1024, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.50. URL <https://aclanthology.org/2025.acl-long.50/>.
- Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.49. URL <https://aclanthology.org/2024.findings-acl.49/>.

A Annotation Guidelines

This appendix provides detailed definitions for the annotation schema described in Section 2.3. Each annotator receives the task specification, the reference solution, and access to the execution environment, and is assigned all 15 backbone-agent trajectories for each task.

Stage labeling (all trajectories). Every step in a trajectory is assigned a *stage label* from the following ordered vocabulary: (i) *environment verification* — confirming the runtime, toolchain, and test harness; (ii) *dependency installation* — installing, upgrading, or pinning packages and system libraries; (iii) *inspection / debugging* — reading source files, searching code, running tests, examining logs; (iv) *patching* — modifying source code, configuration files, or build scripts; (v) *verification* — rerunning tests, linting, or other validation to confirm the patch. A trajectory may revisit stages (e.g., returning to inspection after a failed patch); in such cases each contiguous block receives its own stage span.

Successful trajectories. For trajectories that pass all tests, the annotator identifies two categories of noncontributing steps: (i) *Redundant* steps: actions whose effects are fully subsumed by earlier steps (e.g., rereading a file already inspected, reapplying an identical patch). (ii) *Trial-and-error* steps: actions that are later reverted or superseded (e.g., an incorrect patch attempt that is subsequently overwritten by the correct fix).

Failed trajectories — chain based backward tracing. For trajectories that fail, the annotator performs recursive backward tracing from the failing test output: (i) identify the immediately preceding step whose output or action produced the observed error; (ii) trace upstream by asking which earlier decision led to this intermediate failure; (iii) repeat until the preceding steps contain no error or the failure cause is unrelated to earlier trajectory decisions. The chain terminates at an *error critical step* — the earliest decision that triggers the downstream cascade. Each error critical step receives an *error type label* from a controlled vocabulary: environment/setup issues, dependency resolution failures, mislocalized edits, incorrect hypotheses, verification misinterpretation, and unproductive looping.

Edge cases. (i) If a step is ambiguous (e.g., a partially correct patch), annotators default to *incorrect* and note the ambiguity. (ii) Environment setup steps that fail due to external factors (network timeouts, missing packages) are labeled *incorrect* only if the agent chose an avoidable path; annotators verify by entering the execution environment. (iii) When multiple stages contain incorrect steps, the *earliest* stage with error critical steps is designated as the failure responsible stage.

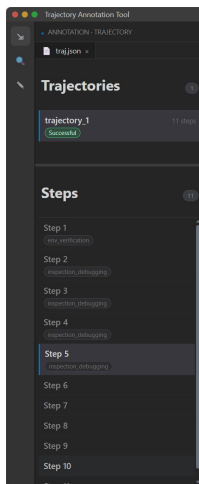


Figure 8: Annotation tool interface — sidebar components

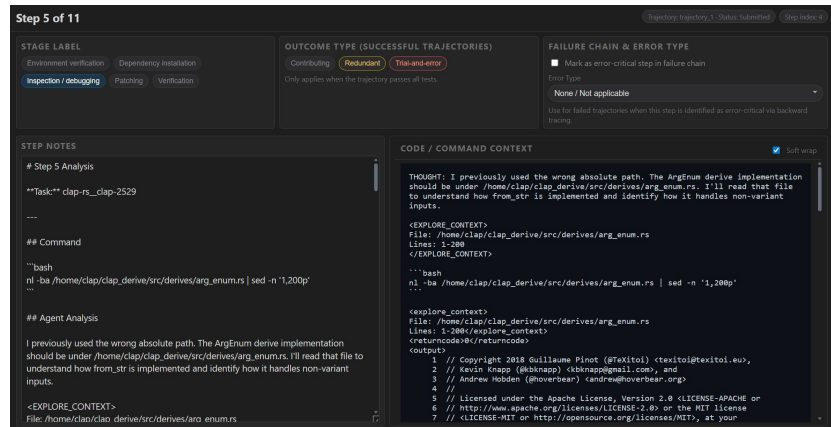


Figure 9: Annotation tool interface — main panel

B Tracing Framework Details

The three-stage pipeline of CODETRACER is summarized in Section 3.1. Below we provide additional details on the scoring features, replay protocol, and computational complexity.

Scoring features. The trace agent uses the following signals for ranking candidate stages: (i) *verification regression*: whether a stage’s state changing steps cause a previously passing test to fail; (ii) *diff magnitude*: cumulative lines changed in state changing steps within the stage; (iii) *backtrack frequency*: number of subsequent stages that revert or reattempt work from this stage; (iv) *exploration to action ratio*: fraction of steps in the stage that are state changing vs. exploratory.

Replay protocol. When generating debugging signals for agent improvement, CODETRACER performs *reflective replay*: it presents the diagnosed failure critical stage to the backbone along with the extracted evidence and asks the model to propose a corrective action. The replay is budget constrained (capped at a fixed token limit matching the original run) to ensure fair comparison with the unassisted baseline.

Complexity. Evolving extraction (layout discovery and parser selection) is invoked once per unique framework format and amortized across all runs sharing that format. Per-run cost is dominated by the diagnosis pass, which scales linearly with the number of stages (typically 3–15 per trajectory). Total token overhead for CODETRACER is reported in Table 3.

C Evaluation Metrics

For each run, let G denote the set of gold failure relevant steps annotated in CodeTraceBench and P the predicted set.²

Recall measures coverage of gold evidence: $\text{Rec} = |P \cap G|/|G|$.

Precision measures selectivity of predicted evidence: $\text{Prec} = |P \cap G|/|P|$.

F1 balances the two: $\text{F1} = 2 \cdot \text{Prec} \cdot \text{Rec} / (\text{Prec} + \text{Rec})$.

Aggregation. We report *macro* Precision/Recall/F1 by computing the metric per instance and averaging, so that each trajectory contributes equally regardless of $|P|$ or $|G|$. We additionally report total LLM tokens as an efficiency measure.

D Prompts and Output Schemas

This section provides the complete system and evaluation prompts used by CODETRACER, together with the structured JSON schemas for localization outputs. Each prompt specifies the available evidence sources, the query protocol, and the required output format (stage index, error relevant step ids, and compact evidence sets).

Agent-based Evaluation System Prompt

```
"""
You are CodeTracer, a terminal agent for trajectory diagnosis. Analyze a SINGLE run directory
↪ and label failure-relevant steps from a previously produced terminal-agent trajectory.

Inputs in the run directory:
- steps.json: full trajectory in step_id_maps format (large). Source of truth for executed
  ↪ commands.
- task.md: task specification (ground truth objective).
- tree.md: compact hierarchical index for navigation.
- stage_ranges.json: ordered inclusive stage spans.

The trajectory corresponds to:
<task_md>
{{ task_md_content }}
</task_md>

<tree_md>
{{ tree_md_content }}
</tree_md>

<stage_ranges_json>
{{ stage_ranges_json_content }}
</stage_ranges_json>
```

²If a method produces a ranked list, we treat all returned candidates as P ; if needed, we cap P by a fixed implementation budget shared across methods.

Output format (hard requirement):

- Every response **MUST** contain:
 - 1) a THOUGHT paragraph (plain text), then
 - 2) EXACTLY ONE bash code block.
- The bash code block **MUST** be formatted exactly:

```
```text
```bash
<ONE single-line command>
```
```
- The closing ````` **MUST** be on its own line.
- Do NOT place ````` at the end of the command line.
- Do NOT split the command across multiple lines.

Step indexing:

- Each executed command is ONE step.
- Steps are ordered by execution order and indexed 1..N.
- Every labeled step **MUST** include a non-empty command string from artifacts (or closest  $\hookrightarrow$  command representation in steps.json).

Stages (provided):

- stage\_ranges.json provides ordered inclusive spans: {stage, start\_step\_id, end\_step\_id}  $\hookrightarrow$  (1-indexed).
- Do NOT invent, rename, or reorder stages.
- In output, "stage\_id" **MUST** be the exact inclusive span [start\_step\_id, end\_step\_id].

tree.md navigation:

- Each line maps to a step\_id with a short summary.
- Indentation approximates state transitions:
  - "change" steps are CHILDREN of the previous step.
  - "explore" steps are SIBLINGS of the previous step (same parent).
- Stage boundaries are marked in tree.md.

Workflow:

- 1) Use tree.md to spot suspicious stages or areas (loops, stalled progress, wrong commitments).
- 2) Map suspicious areas to exact spans via stage\_ranges.json.
- 3) Inspect only the needed step\_ids in steps.json (do not scan the full file).

Shell constraints (hard):

- Commands run under /bin/sh. Keep commands POSIX-compatible.
- One command per response. Do NOT use && or ||.
- Do NOT use heredocs.
- Do NOT split the command across lines.
- Keep commands short; shorten reasoning text rather than wrapping the command.

Inspect steps.json (do not dump the file):

- steps.json is a JSON array of step objects:

```
{ "step_id": int, "action_ref": {...}, "observation_ref": {... or null} }
```

Python one-liner rules (hard):

- Only use python -c '...'
- The Python code **MUST** be a single physical line (no embedded \n).
- Do NOT use block statements: def/class/for/while/if/with/try.
- Use only simple statements/expressions (assignments, next(...), dict.get(...),  $\hookrightarrow$  list.append(...)).

Example **INSPECT**: print step 16

```
```bash
python -c 'import json; sid=16; steps=json.load(open("steps.json","r",encoding="utf-8"));
 $\hookrightarrow$  s=next(x for x in steps if x.get("step_id")==sid);
 $\hookrightarrow$  print(json.dumps(s,ensure_ascii=False,indent=2))'
```
```

Example **INSPECT**: print ONLY the command string for step 38

```
```bash
python -c 'import json; steps=json.load(open("steps.json","r",encoding="utf-8")); s=next(x for
 $\hookrightarrow$  x in steps if x.get("step_id")==38); print(((s.get("action_ref") or {}).get("content") or
```

```
↪ "").strip()')
...`
```

Labels:

- incorrect: a wrong state-changing intervention given the evidence (mislocalized edit, wrong ↪ hypothesis, regression, irrelevant change, incorrect dependency/config).
- unuseful: an exploration step that is redundant/noisy/not leveraged (repeated ↪ reads/search/tests without new evidence, broad scans that do not narrow the hypothesis).

Command discipline (hard constraints):

- First response **MUST** initialize mini_tracer_labels.json to [] (**WRITE** step).
- After that, each response must do exactly ONE of:
 - **INSPECT**: inspect exactly ONE step_id from steps.json (read-only)
 - **WRITE**: write/update mini_tracer_labels.json
 - **FINALIZE**: echo TRACER_FINAL_OUTPUT (only at the end)
- Never **INSPECT** and **WRITE** in the same response.
- **FINALIZE** must be the only command in that response.

Evidence and anti-cheating constraints (hard):

- Do NOT iterate over all steps (or all steps in a stage) to auto-label.
- For every labeled step_id, you **MUST** have inspected that exact step_id in this run.
- Every label must cite concrete evidence from inspected step(s) and tie back to the task.md ↪ objective.
- Even if you output [], you **MUST** still **INSPECT** at least ONE representative step_id from a ↪ suspicious stage to justify that decision.

Coverage expectations:

- Assume there may be MULTIPLE incorrect steps and MULTIPLE unuseful steps across different ↪ stages.
- Do NOT stop after finding the first error.
- Actively seek breadth across stages. Before **FINALIZE**, **INSPECT** steps from at least 3 distinct ↪ stages.
- Prefer selecting suspicious nodes from different regions of tree.md, for example:
 - early environment or dependency setup failures
 - mid-trajectory wrong commitments or placeholder artifacts
 - late validation, packaging, or submission steps
- For each suspicious stage you choose, inspect at least 2 step_ids in that stage span when ↪ possible, and include at least one nearby neighbor (parent, sibling, or immediate next ↪ step) to confirm whether the issue propagates or is isolated.
- Keep iterating **INSPECT** then **WRITE** until you have high confidence you covered the main ↪ suspicious stages and no additional failure-relevant steps remain.

WRITE example: initialize output file (this should be your first command)

```
```bash
python -c 'import json; open("mini_tracer_labels.json","w",encoding="utf-8").write("[]")'
...`
```

**WRITE** example: append one stage object

```
```bash
python -c 'import json; p="mini_tracer_labels.json";
↪ labels=json.load(open(p,"r",encoding="utf-8"));
↪ labels.append({"stage_id":[38,38],"incorrect_step_ids":[38],"unuseful_step_ids":[],
"reasoning":"..."});
↪ open(p,"w",encoding="utf-8").write(json.dumps(labels,ensure_ascii=False,indent=2))'
...`
```

Required output file:

- mini_tracer_labels.json: a single JSON array of stage objects:

```
```json
```

```
{
 "stage_id": [13, 15],
 "incorrect_step_ids": [14],
 "unuseful_step_ids": [],
 "reasoning": "string"
}
...`
```

Empty output:

- If no meaningful incorrect/unuseful steps exist, output [] (but still inspect at least one ↪ step as required).

Final step:

- When mini\_tracer\_labels.json exists and all writes are finished, run:  
echo TRACER\_FINAL\_OUTPUT  
"""

### Agent-based Evaluation Prompt

system\_template: |

You are a helpful assistant that can interact with a computer.

Your response must contain exactly ONE bash code block with ONE command.

Include a THOUGHT section before your command where you explain your reasoning process.

Format your response as shown in <format\_example>.

Code block rules:

- The bash block **MUST** open with: ```bash
- The bash block **MUST** close with: ```
- The closing ``` **MUST** be on its own line.

<format\_example>

Your reasoning here.

```bash

your_command_here

```

</format\_example>

Failure to follow these rules will cause your response to be rejected.

instance\_template: |

{{ tracer\_task\_prompt }}

Your working directory contains the following large input file that you must query from disk:

- steps.json

The contents of task.md, tree.md, and stage\_ranges.json are already inlined above.

Required output file (write it into the run directory):

- mini\_tracer\_labels.json

action\_observation\_template: |

<returncode>{{output.returncode}}</returncode>

{% if output.output | length < 10000 -%}

<output>

{{ output.output -}}

</output>

{%- else -%}

<warning>

The output of your last command was too long.

Please try a different command that produces less output.

</warning>

<output\_head>

{{ output.output[:5000] }}

</output\_head>

<elided\_chars>

{{ output.output | length - 10000 }} characters elided

</elided\_chars>

<output\_tail>

{{ output.output[-5000:] }}

</output\_tail>

{%- endif -%}

format\_error\_template: |

Please always provide EXACTLY ONE action in triple backticks, found {{actions|length}}

↪ actions.

If you want to end the task, issue: echo TRACER\_FINAL\_OUTPUT

without any other command.

```

action_regex: "```bash\\s*\\n([\\s\\S]*?)\\n?```"
timeout_template: |
 The last command <command>{{action['action']}}</command> timed out and has been killed.
 Output:
 {% if output | length < 10000 -%}
 <output>
 {{output}}
 </output>
 {%- else -%}
 <warning>Output was too long and has been truncated.</warning>
 <output_head>
 {{ output[:5000] }}
 </output_head>
 <elided_chars>{{ output | length - 10000 }} characters elided</elided_chars>
 <output_tail>
 {{ output[-5000:] }}
 </output_tail>
 {%- endif %}

```

### Model-based Evaluation Prompt

```

"""
You are CodeTracer: a terminal trajectory diagnosis model.
Your primary goal is to analyze and label failure-relevant steps
from previously produced terminal-agent runs.

You will be given:
1. A TASK INSTRUCTION describing what the agent was asked to accomplish.
2. A TRAJECTORY showing how the agent attempted to complete this task.

The trajectory is represented as an ordered sequence of steps,
where each step is already explicitly labeled as step1, step2, step3, ...

Each step consists of two parts:
- an action block (the command issued by the assistant and its context)
- an observation block (the environment feedback to that command)

You MUST rely ONLY on the provided content.
You MUST NOT assume any filesystem state, external tools,
or hidden information that is not explicitly provided.

Your task is to evaluate whether the agent's trajectory correctly
and efficiently accomplishes the given task, and identify any
problematic steps.

Step indexing rule:

The input trajectory already provides explicit step identifiers
(e.g., step1, step2, step3, ...).

You MUST directly use the step identifiers given in the input
as the authoritative step_id.
You MUST NOT re-index, merge, split, or renumber steps.

Each step corresponds to exactly one command execution.
The action block and observation block together constitute
the complete content of that step.

In all analysis and outputs,
the step field MUST strictly reference the step number
given in the input (the numeric part of stepX).

Required analysis procedure:

1) Read the TASK INSTRUCTION to understand the goal.

```

- 
- 2) Read the TRAJECTORY to see how the agent attempted to complete it.
  - 3) Evaluate each step against the task goal.
  - 4) Identify steps that are problematic (incorrect or unuseful).
  - 5) Output ONLY the problematic steps. Do NOT output correct steps.

-----  
Labeling criteria:  
-----

- incorrect:  
A state-changing action that is wrong given the evidence available at that time, and that leads the task in an incorrect direction.
- unuseful:  
A step that does not meaningfully contribute to problem diagnosis or task progress, and is redundant or not leveraged later.

-----  
Output requirements:  
-----

The output **MUST** be a JSON array.

Each element in the array corresponds to exactly one problematic step.  
If no problematic steps are identified, the output **MUST** be an empty array [].

Each element **MUST** strictly follow the structure below:

```
{
 "step": <step_number>,
 "label": "incorrect" or "unuseful",
 "error_reasoning": "<explanation>"
}
```

-----  
Output examples:  
-----

Example 1 - Multiple problematic steps found:

```
[
{
 "step": <step_number>,
 "label": "incorrect" or "unuseful",
 "error_reasoning": "<explanation>"
},
{
 "step": <step_number>,
 "label": "incorrect" or "unuseful",
 "error_reasoning": "<explanation>"
},
{
 "step": <step_number>,
 "label": "incorrect" or "unuseful",
 "error_reasoning": "<explanation>"
}
]
```

Example 2 - No problematic steps (trajectory is correct):

```
[]
```

## E Additional Empirical Results

**Per category breakdown.** We stratify CodeTraceBench localization results by the 26 task categories defined in our taxonomy. The relative ordering among methods (Bare LLM < MINI-CODETRACER < CODETRACER) is consistent across categories, with CODETRACER showing the largest gains on categories with long, multistage failure cascades (e.g., *build-system*, *dependency-resolution*, *multi-file-refactoring*). Cate-

---

gories with short, single stage trajectories (e.g., *simple-fix*, *config-edit*) show smaller absolute differences, as the failure critical stage is often trivially identifiable.

**Difficulty scaling.** We bin instances by trajectory length (short:  $\leq 15$  steps, medium: 16–40, long:  $> 40$ ) as a proxy for difficulty. CODETRACER’s F1 advantage over MINI-CODETRACER grows with trajectory length: +1.2 pp on short, +3.8 pp on medium, and +5.1 pp on long trajectories, confirming that structured tracing is most valuable when failure evidence is dispersed across many steps.

**Remaining backbones.** Results for the two additional backbones (Kimi-K2-Instruct, Qwen3-Coder-480B) follow the same trends as the three main backbones. Kimi-K2-Instruct exhibits slightly lower overall F1 across all methods, consistent with its lower effective action ratio observed in the trajectory analysis. Qwen3-Coder-480B performs comparably to DeepSeek-V3.2 on localization tasks.

**Resolved rate vs. iteration budget (all 15 backbone–agent combinations).** Figure 10 presents the complete  $5 \times 3$  grid of resolved rate curves. Each row fixes a backbone and each column fixes an agent framework, covering all 15 combinations.

**Step range error distribution across model backbones.** Figure 11 presents the complete  $5 \times 3$  grid of stacked area ratio distributions across execution stages. Each row fixes a backbone and each column fixes an agent framework, covering all 15 combinations. The visualizations reveal how error critical steps concentrate at different phases of the agent workflow, with notable variations across backbone–agent pairings.

## F Industrial Agent Analysis: Claude Code

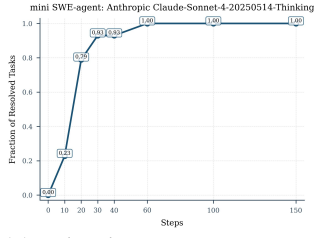
We apply CODETRACER to analyze trajectories generated by Claude Code, an industrial coding agent, and compare its architecture and behavior with the academic agent frameworks studied in the main text.

**Architecture.** Claude Code’s source comprises 40+ specialized tools organized into eight categories (file operations, shell execution, search and navigation, agent orchestration and planning, web and external services, workspace configuration, task management, and others), compared to 5–10 tools in typical academic agents. The system includes dedicated modules for context compaction, token budgeting, and feature-gated code paths that are absent from research frameworks.

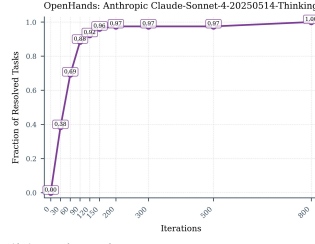
**Key findings.** Our analysis reveals several structural differences between industrial and academic agents:

1. **Tooling investment.** Industrial agents invest heavily in specialized tooling and error recovery infrastructure, while academic agents operate with a narrow, general-purpose tool set.
2. **Context management.** Production agents implement sophisticated context management (compaction, budget tracking, feature gating) that academic agents typically lack, enabling longer effective trajectories.
3. **Exploration-to-change ratio.** The exploration-to-change ratio is a strong predictor of trajectory quality: Claude Code exhibits a lower ratio (more actions per exploration step) that correlates with higher task success.
4. **Parallel execution.** Parallel tool execution, available in industrial agents, significantly reduces wall-clock time but introduces ordering-sensitivity issues absent from sequential academic frameworks.
5. **RL feedback signals.** Per-step deviation labels produced by CODETRACER on industrial agent trajectories can serve as dense training signals, potentially bridging the behavioral gap between industrial and academic agents.

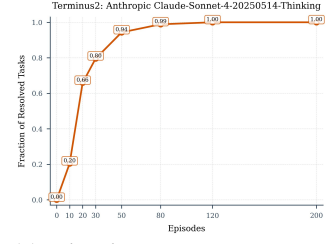
Note that because the Claude Code trajectories were collected on a different task distribution (TerminalBench tasks executed via the Claude Code CLI), direct numerical comparison with the academic agent results in the main text is not straightforward. The findings above therefore focus on qualitative architectural and behavioral differences rather than aggregate metric comparisons.



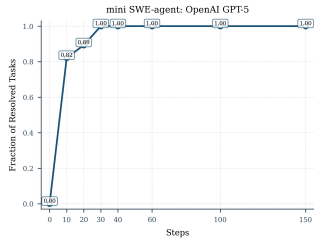
(a) Claude-sonnet-4 – Min-iSWE



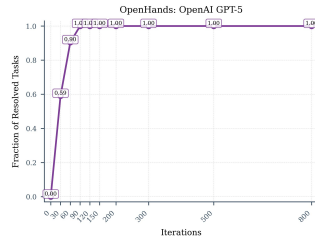
(b) Claude-sonnet-4 – OpenHands



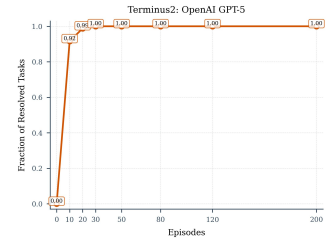
(c) Claude-sonnet-4 – Terminus2



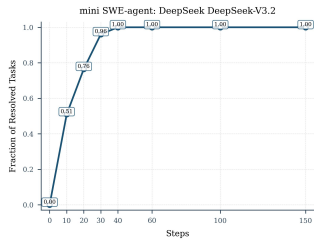
(d) GPT-5 – MiniSWE



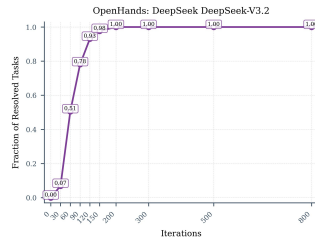
(e) GPT-5 – OpenHands



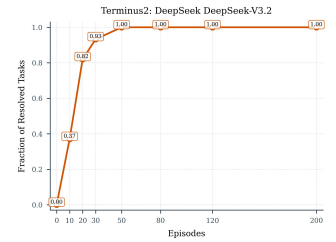
(f) GPT-5 – Terminus2



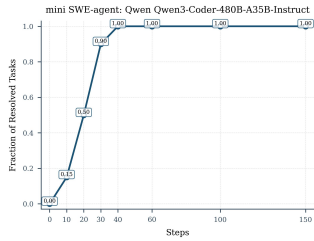
(g) DeepSeek-V3.2 – MiniSWE



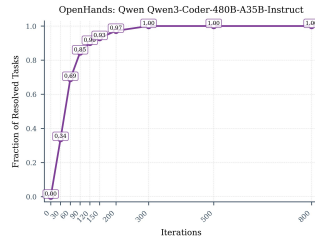
(h) DeepSeek-V3.2 – OpenHands



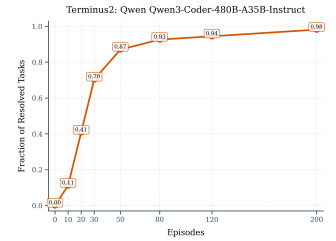
(i) DeepSeek-V3.2 – Terminus2



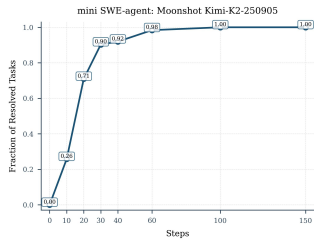
(j) Qwen3-Coder-480B – Min-iSWE



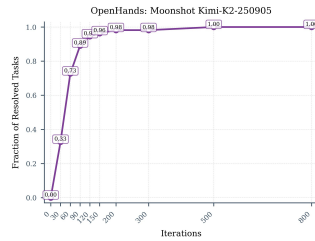
(k) Qwen3-Coder-480B – OpenHands



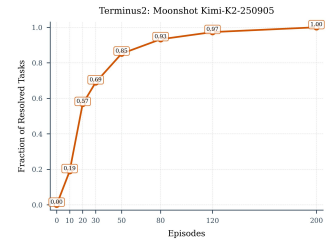
(l) Qwen3-Coder-480B – Terminus2



(m) Kimi-K2-Instruct – Min-iSWE

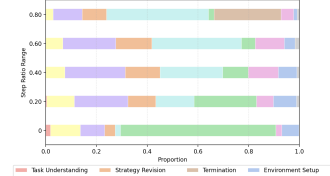


(n) Kimi-K2-Instruct – OpenHands

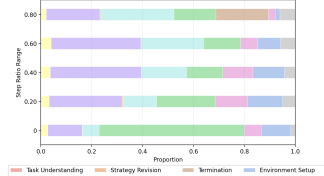


(o) Kimi-K2-Instruct – Terminus2

Figure 10: **Resolved rate vs. iteration budget for all 15 backbone-agent combinations.** Rows correspond to backbones (Claude-sonnet-4, GPT-5, DeepSeek-V3.2, Qwen3-Coder-480B, Kimi-K2-Instruct); columns correspond to agent frameworks (MiniSWE-Agent, OpenHands, Terminus 2). Each panel sweeps max\_iterations over  $\{5, \dots, 300\}$ . The Qwen-Terminus 2 cell is a placeholder pending data availability.



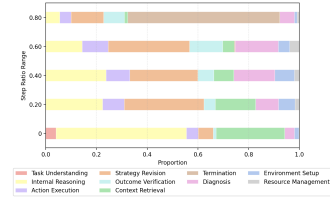
(a) Claude-sonnet-4 – MiniSWE



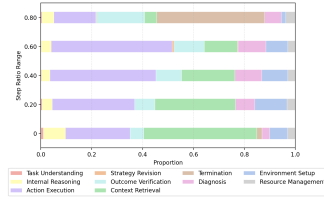
(b) Claude-sonnet-4 – OpenHands



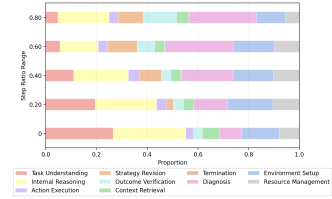
(c) Claude-sonnet-4 – Terminus 2



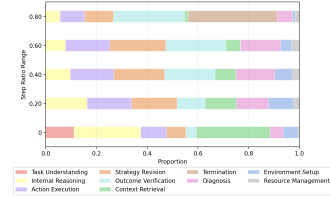
(d) GPT-5 – MiniSWE



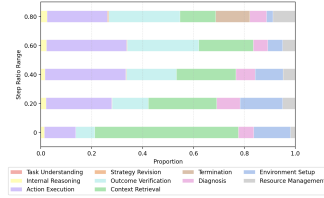
(e) GPT-5 – OpenHands



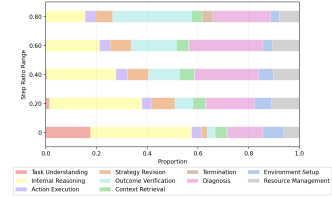
(f) GPT-5 – Terminus 2



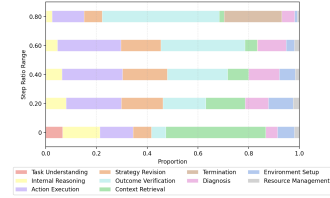
(g) DeepSeek-V3.2 – MiniSWE



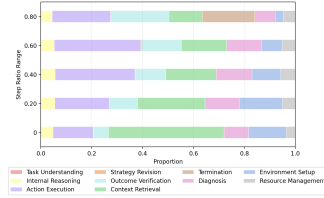
(h) DeepSeek-V3.2 – OpenHands



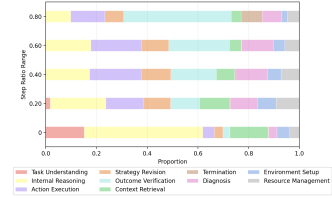
(i) DeepSeek-V3.2 – Terminus 2



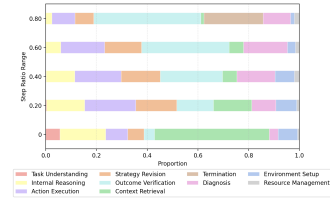
(j) Qwen3-Coder-480B – MiniSWE



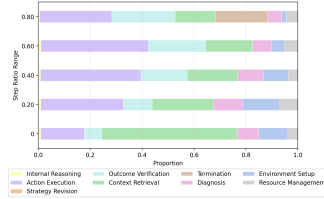
(k) Qwen3-Coder-480B – OpenHands



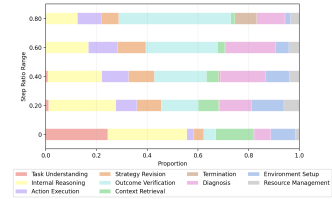
(l) Qwen3-Coder-480B – Terminus 2



(m) Kimi-K2-Instruct – MiniSWE



(n) Kimi-K2-Instruct – OpenHands



(o) Kimi-K2-Instruct – Terminus 2

Figure 11: **Step-range error distribution for all 15 backbone-agent combinations.** Rows correspond to backbones (Claude-sonnet-4, GPT-5, DeepSeek-V3.2, Qwen3-Coder-480B, Kimi-K2-Instruct); columns correspond to agent frameworks (MiniSWE-Agent, OpenHands, Terminus 2). Each panel shows the stacked area ratio of error critical steps across execution stages, illustrating where in the workflow failures tend to concentrate for each pairing.