TRAJDEBUG: Tracing Error Lifecycle to Identify Critical Failures in Long-Horizon Agent Trajectories

Yunjia Qi[1], Zehua Yin[1], Xintong Shi[1], Hao Peng[1],
Songyuanyi Lu[3], Yixian Liu[3], Richeng Xuan[3], Yuhong Liu[3], Zhichao Hu[3,*],
Xiaozhi Wang[2], Lei Hou[1], Bin Xu[1,*], Juanzi Li[1]
[1]Department of Computer Science and Technology, BNRist, Tsinghua University
[2]Shenzhen International Graduate School, Tsinghua University
[3]Tencent Hunyuan
[*]Corresponding authors
qyj23@mails.tsinghua.edu.cn

Abstract

LLM-based agentic systems have shown remarkable capabilities in complex domains, while suffering from cascading errors and difficulty in debugging. Critical error detection aims to locate the earliest error step in a failed trajectory that is responsible for the final failure. However, progress faces two main challenges. First, long trajectories make it difficult to identify individual errors, since the evidence for judging a step may be scattered across distant instructions, observations, and prior context. Second, failed trajectories often contain multiple local errors whose downstream effects differ: some are repaired, some remain harmless, and only some contribute to the final failure. In this work, we propose TRAJDEBUG, an error-lifecycle tracing framework that addresses long-trajectory error discovery with multi-granularity history compression and evidence-based error identification, and supports critical attribution by tracing each error's resolution status and terminal impact. We further construct TRAJERRBENCH, a benchmark of 486 manually annotated failed trajectories from τ^2-Benchand SWE-Bench Pro, covering realistic tool-use and coding scenarios. Experiments across diverse agent benchmarks show that TRAJDEBUG achieves the best overall performance over existing baselines, and application studies further demonstrate that its diagnoses provide actionable feedback for improving downstream agent success. We will release the code and data to facilitate further research[1].

1 Introduction

Large language model (LLM)-based agentic systems have shown remarkable capabilities in complex, real-world domains, such as software engineering (Jimenez et al., 2024; Deng et al., 2025), scientific discovery (Mialon et al., 2024), and multi-turn customer service workflows (Barrès et al., 2025). These trajectories often span hundreds of reasoning, action, and observation steps, and a terminal failure may trace back to an early mistake that propagates through subsequent decisions, making the trajectory difficult to debug and improve. *Critical error detection* (Zhang et al., 2025b) addresses this difficulty by locating the earliest error step in a failed trajectory that is causally linked to the task failure, thereby supporting trajectory repair and system reliability analysis.

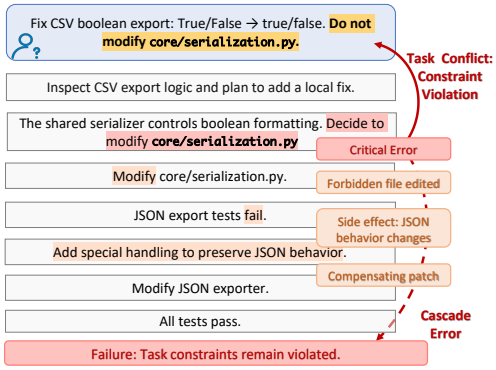

Figure 1: Critical error detection requires grounding errors in long-range context and distinguishing the failure-responsible error from multiple coexisting errors.

Critical error detection faces two key challenges in long, complex agent trajectories. First, *long trajectories make it difficult to identify individual errors.* Complex tasks often require extended execution processes that interleave planning, reasoning, tool use, environment feedback, and repair attempts over many steps (Yao et al., 2023a,b). As a result, judging whether a step is erroneous may require relating it to evidence scattered across distant task instructions, observations, and prior trajectory context (Peng et al., 2023). For example, in Figure 1, the erroneous decision conflicts with a task constraint stated at the beginning of the trajectory, requiring long-range task context to identify the error. Second, *failed trajectories often contain mul-*

[1]https://github.com/THU-KEG/TrajDebug

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 9169–9192
October 24-29, 2026 ©2026 Association for Computational Linguistics

tiple local errors whose downstream effects differ, obscuring which error is critical to the final failure. A failed trajectory may contain many local errors (Cemri et al., 2025a), some are resolved, some are inconsequential, and others are merely downstream symptoms of earlier mistakes (Fan et al., 2026). Thus, the critical error is not necessarily the first local error observed in the trajectory, nor the temporally closest error to the terminal failure.

Existing methods often address only part of this problem. Current taxonomy-based and constraint-based methods focus on identifying candidate erroneous steps in long trajectories (Zhu et al., 2025; Barke et al., 2026), while causal-diagnostic methods use causal structures or counterfactual attribution to filter propagated errors (Wang et al., 2026; Li et al., 2026). However, candidate-error methods often leave criticality to a holistic final judgment, which is brittle when many local errors coexist. Causal attribution methods model dependencies among errors, but their judgments can remain ambiguous when later feedback and repair attempts alter the effects of earlier errors.

To address these challenges, we propose TRAJDEBUG, an evidence-grounded error-lifecycle tracing framework for critical error detection. TRAJDEBUG decomposes the task into three stages: error trigger detection, error state classification, and critical attribution. First, to identify individual errors in long trajectories, TRAJDEBUG uses multi-granularity history compression and detects error triggers as wrong commitments grounded in conflicts with task instructions, trajectory history, environment feedback, or the agent's own reasoning. This turns error discovery from holistic diagnosis into an auditable evidence-verification problem, reducing hallucinated or weakly grounded diagnoses (Gao et al., 2023). Second, to address the ambiguity caused by multiple local errors with different downstream effects, TRAJDEBUG groups related triggers into error instances by their shared violated reference, such as a task constraint or prior observation. It then classifies the state of each instance by tracking whether the wrong commitment is resolved, and whether it leaves an observable terminal footprint, such as an irreversible state change, a persistent task violation, or a substantial recovery cost. These state classifications distinguish errors that are repaired or remain harmless from those that stay relevant to the final failure. Finally, TRAJDEBUG selects the critical error step from evidence-backed instances with terminal relevance, rather than judging over the full trajectory.

To evaluate critical error detection under realistic scenarios and complement existing benchmarks (Zhang et al., 2025b; Zhu et al., 2025), we construct TRAJERRBENCH, a benchmark of 486 manually annotated failed trajectories: 400 from τ^2-Bench (Barrès et al., 2025), covering diverse tool-use and user-interaction scenarios, and 86 from SWE-Bench Pro (Deng et al., 2025), covering long-horizon coding trajectories with an average length of about 119.7 steps.

Experiments on existing agent benchmarks and TRAJERRBENCH show that TRAJDEBUG achieves the best overall performance over advanced LLM prompting baselines and existing diagnostic systems. Length-based analysis shows that TRAJDEBUG remains more robust on long-horizon trajectories. To assess whether critical-error diagnoses can improve future agent behavior, we explore two inference-time application scenarios. In the first, each failed trajectory is diagnosed and converted into targeted guidance before re-executing the same task, improving success by 10.80% on average. In the second, diagnoses from a small set of historical failures are aggregated into reusable failure memory and transferred to held-out tasks, yielding a 5.70% average improvement. These results suggest that TRAJDEBUG can serve not only as a diagnostic tool, but also as a practical interface for converting failed executions into actionable experience for improving long-horizon agents.

2 Pilot Study

This section first formalizes critical error step detection (§ 2.1) and then provides empirical motivation for the two challenges discussed above. We examine how trajectory length is associated with critical error localization difficulty (§ 2.2) and how local errors evolve within failed trajectories (§ 2.3). See Appendix A for experiment and annotation details.

2.1 Task Formulation

Let a trajectory be $\tau = (x_1, x_2, \ldots, x_N)$, where each step x_t contains the agent's reasoning, action, or environment response. A trajectory is *failed* if its final state does not satisfy the task goal. Following Zhang et al. (2025b), a step x_t contains a decisive error if replacing its action with a correct one, while leaving all prior steps unchanged and allowing the remainder of the trajectory to unfold correctly, would have turned the failure into a suc-

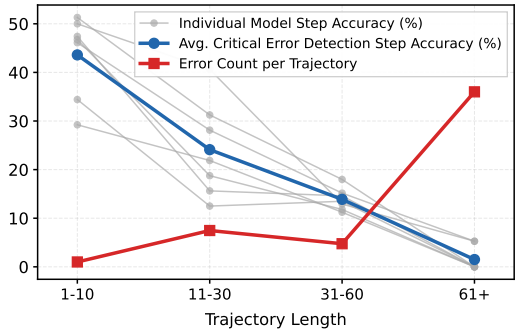

Figure 2: Critical error detection accuracy and local error density across trajectory length buckets.

cess. The **critical error step** is the earliest decisive error step, identifying the origin of the failure rather than its downstream propagation.

2.2 Error Detection under Growing Context

We group trajectories from WhoAndWhen (Zhang et al., 2025b) and AgentDebugBench (Zhu et al., 2025) by trajectory length, and directly prompt LLMs to identify the annotated critical error step in each failed trajectory, and use critical-step detection accuracy as an indicator of the difficulty introduced by growing context. We evaluate seven widely used advanced models and report the average accuracy across models to reveal the overall trend. Figure 2 shows that detection accuracy decreases as trajectory length increases, suggesting that long-horizon context makes critical error detection harder. This trend is consistent with prior observations that longer trajectories are associated with lower task success rates (Qi et al., 2025).

2.3 Trajectory Dynamics of Local Errors

We further sample 50 failed trajectories from these benchmarks and annotate local error steps with three annotators, with details in Appendix A. The sampled trajectories contain 381 local errors in total, averaging 7.62 per failed trajectory, while each trajectory has only one critical error. Figure 2 shows that the average number of local errors increases with trajectory length, indicating that longer trajectories expose a denser set of plausible but non-critical candidates. To understand how non-critical errors evolve, we exclude the critical errors and analyze the remaining 331 local errors. We find that 205 (61.9%) are later repaired by the agent, while 104 (31.4%) persist until the final outcome. Only 22 (6.6%) are unrepaired yet dormant: for example, a step may miscount four search results as two, but no later decision depends on the count. These dynamics show that critical error detection must distinguish local error occurrence from its downstream state and terminal impact, motivating TRAJDEBUG in § 3.3.

3 TRAJDEBUG

Figure 3 shows the overall framework of TRAJDEBUG. Given a failed trajectory τ, TRAJDEBUG first constructs multi-granularity trajectory views to preserve local evidence while compressing distant context (§3.1). TRAJDEBUG then follows an error-lifecycle tracking perspective and decomposes critical error detection into three stages. First, TRAJDEBUG detects evidence-grounded error *triggers* at individual steps (§3.2). Second, it groups related triggers into object-anchored error *instances* and classifies each instance's error state, based on whether the error is resolved or remains active and whether it leaves an observable terminal footprint (§3.3). Finally, it attributes the final failure to the critical candidate through candidate-set-guided causal attribution (§3.4). This turns critical error detection from an end-to-end judgment over the full trajectory into three more controlled steps: finding evidence-backed local errors, classifying their error states, and selecting the causally responsible step from the remaining candidates. More details and examples are in Appendix C and Appendix D.

3.1 Multi-Granularity Compression

To reduce the long-context burden while preserving verifiable evidence, we use LLM to construct three views for each step: (1) *high-detail* view keeps the original instruction, action, observation, and locally relevant reasoning snippets needed for evidence verification; (2) *medium-detail* view summarizes the step's main intent, action, and state update; (3) *low-detail* view records only coarse progress, salient entities, and unresolved commitments. Each stage then retrieves the finest view required for its decision, using high-detail views for local verification and compressed views for distant context.

3.2 Error Trigger Detection

For each step, given the trajectory prefix up to that step, the first stage extracts per-step atomic error triggers. A **trigger** $e = (t, c, p, q_w, q_r)$ denotes a local evidence-grounded mismatch at step t, where q_w is an erroneous commitment expressed in the current step, q_r is the violated reference, c is the reference category, and p is the execution phase. The violated reference q_r may come from the task

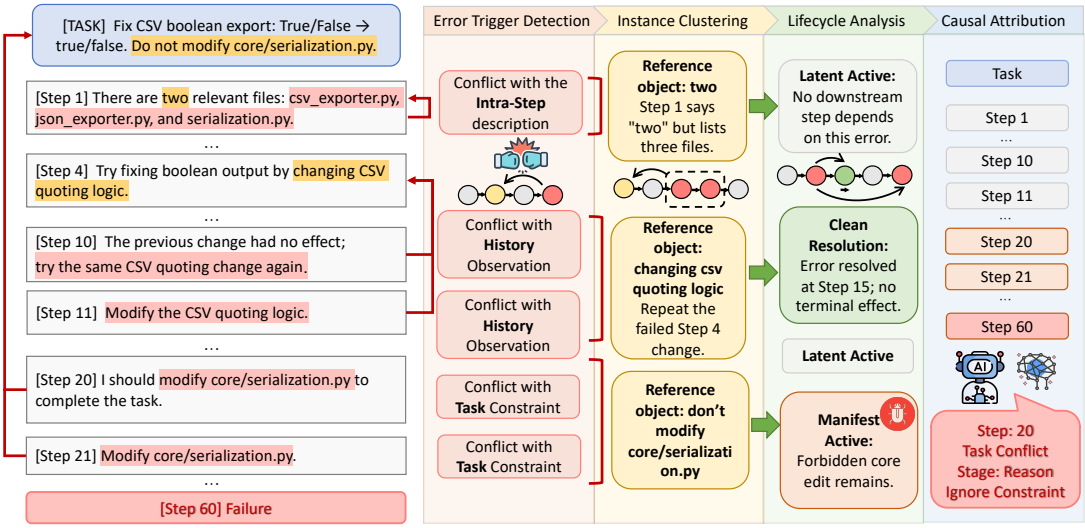

Figure 3: An overview of the framework of TRAJDEBUG.

instruction, prior trajectory, environment feedback, or the same step. To prevent subjective drift and hallucinated diagnoses, each trigger must satisfy a **verbatim evidence** condition: both q_w and q_r must be explicitly citable; otherwise, the trigger is discarded. A step may contain multiple triggers when it violates different references or expresses different erroneous commitments. For each inspected step, the detector uses the high-detail current step and task instruction, medium-detail previous two steps, and low-detail views for the remaining history. This preserves local evidence for verification while keeping long-range context compact.

We organize triggers along two axes. The first axis is the **reference category** c, which specifies the source of the violated reference q_r: (1) *Task Conflict*, where q_r comes from the task instruction, such as a constraint; (2) *History Conflict*, where q_r comes from prior trajectory context, such as a tool output, or established trajectory fact; and (3) *Intra-Step Conflict*, where q_r comes from the current step itself, such as an earlier claim. We additionally use (4) *Environment Anomaly* as an exogenous category when the agent action is reasonable but the environment response is abnormal. The second axis is the **execution phase** p, which records where the mismatch surfaces in the agent process (Deshpande et al., 2025; Zhu et al., 2025): *planning*, *reasoning*, *action*, *observation*, or *verification* (detailed in Appendix D). The reference category c, together with the violated reference q_r, determines the reference object O used for instance clustering, while the execution phase p characterizes the agent-side mechanism for fine-grained analysis.

3.3 Error State Classification

The second stage groups per-step triggers into error **instances** and classifies the state of each instance. Since the same wrong commitment may appear across multiple steps (Xie et al., 2026), such as a task-constraint violation surfacing in planning, action, and verification, we define an instance as $E = (\mathcal{E}, O)$, where $\mathcal{E}$ contains triggers that violate the same reference object O. This avoids overcounting repeated manifestations of the same error. Each instance thus represents one continuous episode of holding a wrong commitment to a concrete reference object.

For each instance, the state classifier makes two evidence-backed judgments: whether the wrong commitment is **resolved** or remains **active**, and whether it leaves an observable **terminal footprint**. An instance is resolved only if later steps explicitly revisit O and provide citable evidence that supersedes the wrong commitment; otherwise, it remains active. A terminal footprint is assigned when there is evidence of: (i) an *irreversible* state change, such as submitting a wrong order; (ii) a *semantic* footprint, where the wrong commitment is reflected in later steps or the terminal state, such as a persistent violation of a task constraint; or (iii) *budget debt*, where the error is resolved only after consuming more than $k\%$ of the trajectory. In implementation, we set $k = 50$, as exceeding half the trajectory mostly leaves an insufficient budget for recovery.

Combining these two judgments yields four error states: **Clean Resolution** (resolved, no footprint), **Costly Resolution** (resolved, budget-debt footprint), **Manifest Active** (active, irreversible or

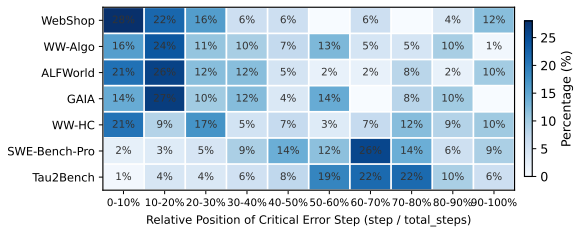

Figure 4: Relative position of the critical error step within each trajectory in TRAJERRBENCH and others.

semantic footprint), and **Latent Active** (active, no footprint). We retain terminal-relevant instances as candidates:

$$\mathcal{F}(\tau) = \{E : \text{state}(E) \in \{\text{CostlyResolution}, \text{ManifestActive}\}\}. \quad (1)$$

Each candidate is passed to the final stage with its origin step, state label, footprint channel, and supporting evidence; final criticality is decided by the attribution stage.

3.4 Candidate-Set-Guided Causal Attribution

This stage executes candidate-set-guided causal attribution. Given the candidate set $\mathcal{F}(\tau)$, this attribution head selects the candidate instance whose first step best explains the terminal failure. A simple earliest-candidate rule is insufficient because temporal order alone does not determine failure responsibility. For example, an earlier budget-debt instance should not be selected over a later semantic instance unless the wasted budget plausibly prevented the agent from avoiding the later failure. Therefore, we use an LLM as the final attribution head. Each candidate is provided with its first step, state label, and supporting verbatim evidence, and the LLM selects the critical error step.

4 TRAJERRBENCH

Existing critical-error benchmarks remain limited in scale, domain diversity, or trajectory complexity (Zhang et al., 2025b; Zhu et al., 2025; Barke et al., 2026; Li et al., 2026). We construct TRAJERRBENCH, a benchmark of 486 manually annotated failed trajectories from two complementary domains: 400 τ^2-Bench trajectories (Barrès et al., 2025) and 86 SWE-Bench Pro trajectories (Deng et al., 2025), averaging 29.3 and 119.7 steps, respectively. Following WhoAndWhen (Zhang et al., 2025b), annotators identify the earliest evidence-grounded mistake responsible for the final failure, while separating locally wrong but later repaired mistakes from failure-responsible ones. Each trajectory is annotated by three annotators, and labels with at least two-way agreement are used as ground truth; this majority-vote protocol yields usable labels for 98.2% of τ^2-Bench trajectories and 91.9% of SWE-Bench Pro trajectories. The critical-error step labels reach almost-perfect agreement on τ^2-Bench (Fleiss' κ=0.91) and substantial agreement on the longer, code-heavy SWE-Bench Pro trajectories (Fleiss' κ=0.67); details are in Appendix B.

Figure 4 shows that critical errors in TRAJERRBENCH often occur in the mid-to-late. This reflects the shared structure of TRAJERRBENCH: agents first gather task-relevant information before making decisive decisions, through user interaction in τ^2-Bench and repository exploration in SWE-Bench Pro. Such extended information gathering shifts many critical errors later and makes localization harder, as relevant evidence is accumulated across earlier context. We also annotate each critical error by contradiction reference category and execution phase, with full statistics in Appendix B. In τ^2-Bench, critical errors are almost evenly split between task conflicts (52.4%) and history conflicts (46.3%), while SWE-Bench Pro is dominated by task conflicts (73.4%) with fewer history conflicts (24.1%). Reasoning is the dominant failure phase in both domains (60.7% and 57.1%), suggesting that failures mainly arise from misinterpreting accumulated context in agentic trajectories.

5 Experiments

5.1 Experimental Settings

Datasets and Evaluation Metric. We evaluate our approach across benchmarks in various domains. **WhoAndWhen** (Zhang et al., 2025b) contains a hand-crafted subset of 58 trajectories and an algorithm-generated subset of 126 trajectories drawn from GAIA and AssistantBench. **AgentDebugBench** (Zhu et al., 2025) provides 100 ALFWorld and 50 each from GAIA and WebShop trajectories. One GAIA trajectory has a null ground-truth critical-error label and is therefore excluded from evaluation, leaving 869 evaluated trajectories in total. **TRAJERRBENCH** (§ 4) adds 400 τ^2-Bench and 86 SWE-Bench Pro trajectories, extending evaluation to substantially longer tool-use and software-engineering rollouts. Following prior benchmark (Zhang et al., 2025b; Zhu et al., 2025), we report **exact critical-step accuracy**, where a prediction is correct only if it matches the annotated

Method	AgentDebugBench			WhoAndWhen		TRAJERRBENCH		AVG
	ALFWorld	GAIA	WebShop	Hand-Crafted	Algorithm	τ^2-Bench	SWE-Bench Pro	
Avg. Trajectory Length	60.00	28.68	51.84	51.60	8.72	29.27	119.70	49.97
Direct Prompting								
GLM-5.1	15.00	26.00	26.00	15.52	42.86	46.75	6.98	25.59
Gemini-3.1-Pro	19.00	30.61	**32.00**	20.69	46.82	52.50	17.44	31.29
DeepSeek-V4-Pro	15.00	34.69	24.00	18.97	42.06	48.25	12.79	27.97
Claude Sonnet 4.6	13.00	30.61	18.00	12.07	29.37	42.75	15.12	22.99
GPT-5.4	11.00	20.41	14.00	13.79	28.57	41.50	10.46	19.96
Claude Opus 4.6	10.00	34.69	24.00	**27.59**	46.83	45.25	17.44	29.40
Qwen3-235B-A22B	9.00	33.33	18.00	17.24	42.06	42.75	17.44	25.69
Multi-Agent System								
AgentDebugger	21.00	36.00	20.00	12.07	40.48	26.00	10.47	23.72
CHIEF	6.00	26.00	10.00	18.97	41.27	26.82	2.32	18.77
AgentRX	8.00	26.53	18.00	25.86	41.27	36.25	5.81	23.10
TRAJDEBUG (Ours)	**26.00**	**38.78**	26.00	22.41	**48.41**	**52.75**	**24.41**	**34.11**

Table 1: Main results on critical error detection across multiple benchmarks. All methods operate without task ground truth and receive only the failed trajectory. **Bold** values indicate the best result per column.

step, and evaluate all methods in a realistic setting where they use only the failed trajectory and failure signal, without success traces, gold outcomes, or auxiliary debugging signals.

Baselines. We compare with two families of baselines. Motivated by the effectiveness of direct prompting reported in prior work (Zhang et al., 2025b; Banerjee et al., 2025), our first family is **Direct Prompting**, which queries an LLM for the critical error step from the full trajectory under a unified instruction; we instantiate it with seven widely-used advanced models: GLM-5.1 (GLM, 2026), Gemini-3.1-Pro (Team, 2023), DeepSeek-V4-Pro (DeepSeek-AI, 2025), Claude Sonnet 4.6 (Anthropic, 2026b), Claude Opus 4.6 (Anthropic, 2026a), GPT-5.4 (OpenAI, 2023), and Qwen3-235B-A22B-Thinking (Team, 2025). The second family is **Multi-Agent Systems**, which decomposes the task across multiple LLM calls: *AgentDebugger* (Zhu et al., 2025) prompts the judge with a phase-level error taxonomy and produces a structured diagnosis; *CHIEF* (Wang et al., 2026) parses the trajectory into a hierarchical causal graph and selects the critical step via counterfactual backtracking against a virtual oracle; *AgentRX* (Barke et al., 2026) first flags steps that violate explicit task or schema constraints and then asks an LLM to select the critical error.

Implementation. We implement TRAJDEBUG and all multi-agent baselines with Qwen3-235B-A22B-Thinking (Team, 2025) for a fair comparison, while direct-prompting baselines use the LLMs listed in Table 1. All methods use temperature 0 for reproducibility, and multi-agent baselines follow official prompts and pipelines where available. Implementation details are in Appendix E.

5.2 Main Results

All experimental results are presented in Table 1. We have the following observations: (1) TRAJDEBUG achieves the best overall average in our benchmark suite. It obtains the highest macro-average accuracy (34.11%), outperforming both direct prompting with frontier models and all multi-agent baselines on average. The absolute accuracies remain low across all methods, highlighting the difficulty of identifying the exact critical step from only the failed trajectory. Under this challenging setting, TRAJDEBUG improves over direct prompting with the same backbone from 25.69% to 34.11% (+8.42), demonstrating the effectiveness of our framework. (2) TRAJDEBUG generalizes well across heterogeneous agent domains. It ranks first on five of seven datasets, spanning embodied decision-making, open-domain information seeking, user-facing tool use, and long-horizon software engineering. In contrast, multi-agent baselines show uneven cross-domain performance(e.g., CHIEF drops to 2.32% on SWE-Bench Pro and AgentRX to 8.00% on ALFWorld), suggesting that their pipelines transfer less consistently across agent scenarios. These results demonstrate that TRAJDEBUG's evidence-grounded trigger detection and error state classification offer robust advantages that generalize across diverse agent scenarios.

Method	τ^2-Bench	SWE-Bench Pro	AVG
TRAJDEBUG (Ours)	**52.75**	**24.41**	**38.58**
w/o Multi-Granularity Compression	20.00	15.12	17.56
w/o Evidence-Grounded Triggers	46.25	20.93	33.59
w/o Error State Classification	46.25	16.28	31.27
w/o All Components	42.75	17.44	30.10

Table 2: Ablation study of TRAJDEBUG.

(3) TRAJDEBUG remains effective on the longest-horizon benchmarks. On ALFWorld (60.00 steps on average) and SWE-Bench Pro (119.70 steps), where evidence is distributed across many steps and multiple errors often coexist, TRAJDEBUG attains the best accuracy, with gains of +13.00% and +6.97% over the corresponding direct baselines. These results suggest that evidence-grounded trigger detection and error state classification help narrow critical-step localization in long trajectories with complex failure dynamics.

5.3 Ablation Study

Table 2 ablates the three key components of TRAJDEBUG on τ^2-Bench and SWE-Bench Pro. Replacing multi-granularity compression with the same hard-truncation strategy used by direct prompting causes the largest drop, reducing AVG by 21.02 points. This shows that preserving evidence from different history granularities is essential for long trajectories, rather than merely increasing the number of LLM calls. Removing error trigger detection or error state classification lowers AVG by 4.99 and 7.31 points, respectively, showing that these two downstream components are also useful and non-redundant. On τ^2-Bench, both ablations reach the same accuracy, suggesting that the trigger taxonomy and state classification stage act as orthogonal filters of comparable strength on tool-use trajectories. On SWE-Bench Pro, removing state classification reduces accuracy to 16.28, below the direct-prompting baseline of 17.44, whereas removing only the taxonomy still achieves 20.93. This pattern supports our motivation in § 2: long-horizon trajectories contain many detected errors that are cleanly resolved or dormant, making final attribution more difficult. State-based filtering narrows attribution to errors with terminal footprint.

5.4 Length Analysis

Figure 5 shows critical-step detection accuracy across trajectory length buckets. All baselines degrade sharply as trajectories grow, dropping from

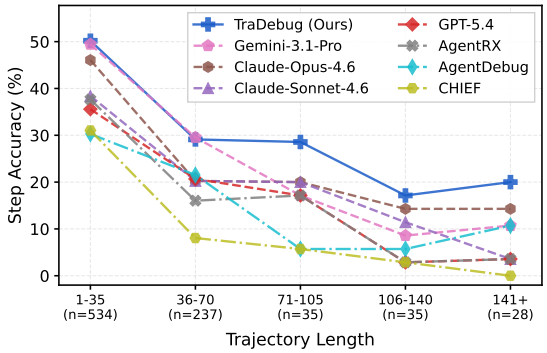

Figure 5: Accuracy across trajectory length buckets.

35–50% on short trajectories to below 15% on long ones. In contrast, TRAJDEBUG remains substantially more stable, retaining over 20% accuracy on the longest bucket where the best baseline drops to around 14%. This suggests that TRAJDEBUG is still affected by increasing trajectory length, but evidence-grounded error detection and state-based candidate filtering help it degrade less sharply and yield larger gains on long-horizon trajectories.

6 Application: Critical Error Detection as Feedback for Agent Improvement

Beyond diagnostic accuracy, a critical error detector is useful only if its outputs translate into measurable agent improvement. We evaluate TRAJDEBUG as a feedback source for the agent in two deployment scenarios on τ^2-Bench (Barrès et al., 2025) and a sampled set of 100 SWE-Bench Verified (Jimenez et al., 2024), covering both an idealized per-trajectory repair setting and a more realistic cross-trajectory transfer setting. In both scenarios, the actor is GLM-5.1, all detectors use Qwen3-235B-A22B-Thinking as the backbone, and detector-produced feedback is injected into the actor's system prompt. We compare TRAJDEBUG against two baselines: *Vanilla*, which directly prompts with the failed trajectory to find the critical error step and write feedback, and *Self-Reflection* (Renze and Guven, 2024), which generates a reflection over the trajectory before producing the feedback. Details are in Appendix E.4.

Per-Trajectory Repair under Oracle Failure. We assume oracle access to the success label of each rollout: for every failed trajectory, the detector produces a feedback, which is injected into the system prompt before re-executing the same task with the same actor. As shown in the upper block of Table 3, TRAJDEBUG delivers the largest gains

Method	Airline	Retail	SWE-Bench
Per-Trajectory Repair (Oracle Failure)			
Initial	78.00	84.21	72.00
Self-Reflection	84.00 (+6.00)	93.86 (+9.65)	78.00 (+6.00)
Vanilla Debug	88.00 (+10.00)	93.86 (+9.65)	80.00 (+8.00)
TrajDebug	**90.00** (+12.00)	**95.61** (+11.40)	**81.00** (+9.00)
Failure-Memory Transfer (Realistic)			
Initial	78.78	84.21	71.64
Self-Reflection	81.81 (+3.03)	82.89 (-1.32)	74.63 (+2.99)
Vanilla Debug	75.75 (-3.03)	88.15 (+3.94)	71.64 (+0.00)
TrajDebug	**84.84** (+6.06)	**90.78** (+6.57)	**76.12** (+4.48)

Table 3: Task success rate (%) when detector outputs are used as feedback for GLM-5.1.

across all three settings, lifting the average success rate from 78.07 to 88.87, surpassing Vanilla Debug and Self-Reflection. The gap shows that more accurate critical-error localization produces more actionable feedback, beyond what holistic prompting or generic reflection can offer.

Failure-Memory Transfer to Held-Out Trajectories. We further test a realistic deployment in which the detector is applied to only a small slice of historical failures and the resulting feedback must generalize to unseen tasks. For each subset, we split the trajectories per-dataset into a $1/3$ *memory-construction split* and a $2/3$ *held-out evaluation split*. On the memory-construction split, the detector produces feedback on each failed trajectory; all feedbacks are aggregated into a memory pool, which is then injected as a whole into the actor's system prompt when evaluating on the held-out split. As shown in the lower block of Table 3, despite no per-instance oracle, TrajDebug still yields the largest improvement, whereas Vanilla and Self-Reflection show less stable transfer and can even reduce performance in some cases. This indicates that the feedback induced by TrajDebug encodes transferable failure patterns rather than instance-specific fixes, suggesting that critical error detection can serve as a low-cost source of reusable experience for long-horizon agents.

7 Related Work

Trajectory analysis and process supervision. Recent work analyzes agent executions beyond final answers, including process reward models that assign step-level scores (Xi et al., 2026; Fan et al., 2026) and studies that summarize recurring failure modes such as coordination, tool-use, and planning errors (Cemri et al., 2025a,b; Lu et al., 2025; Mulian et al., 2026; Ma et al., 2025). However, step-level scores do not isolate the failure-responsible step, and failure-mode taxonomies identify error categories rather than the decisive step in a specific trajectory.

Critical error attribution. A recent line studies fine-grained failure attribution across multi-agent reasoning (Zhang et al., 2025b; Chen et al., 2026; In et al., 2026; Banerjee et al., 2025), embodied and web environments (Zhu et al., 2025), and software engineering (Deshpande et al., 2025; Li et al., 2026). Methodologically, prompt-based approaches inspect raw or compressed trajectories with an LLM judge (Zhang et al., 2025b; Banerjee et al., 2025), but their diagnoses can be weakly grounded in evidence; taxonomy- and constraint-based approaches narrow the search space with predefined error categories or invariants (Zhu et al., 2025; Barke et al., 2026), yet miss errors that fall outside the schema; replay- and spectrum-based methods localize errors by re-executing or perturbing the trajectory (Ge et al., 2025), but require deterministic environments; graph-based methods model dependencies among trajectory elements (Wang et al., 2026; zha, 2025; Zhang et al., 2025a; Li et al., 2026; Wang, 2026), but use connectedness as a proxy for terminal responsibility, leaving each error's resolution status and terminal impact implicit. In contrast, TrajDebug grounds error triggers in verbatim evidence, classifies error states, and performs attribution over terminal-relevant candidates, addressing both long-context error identification and ambiguity among multiple local errors.

Benchmarks for failure attribution. Existing benchmarks are typically tied to a single domain and modest trajectory length: WhoAndWhen (Zhang et al., 2025b), TraceElephant (Chen et al., 2026), and MPBench (In et al., 2026) target multi-agent dialogues; AgentDebug (Zhu et al., 2025) covers embodied and web tasks; and TRAIL (Deshpande et al., 2025) and CodeTracer (Li et al., 2026) focus on code agents. TrajErrBench complements them with 486 manually annotated long-horizon trajectories from customer-service tool use (Barrès et al., 2025) and repository-scale software engineering (Deng et al., 2025), with up to $\sim$120 steps per trajectory, exposing critical errors under long-range evidence dependencies and multiple coexisting local errors.

8 Conclusion

We presented TRAJDEBUG, a three-stage error-lifecycle tracing framework for critical error detection. It combines error trigger detection, error state classification, and candidate-set-guided attribution to identify errors in long trajectories and distinguish failure-responsible errors among multiple errors. We also introduced TRAJERRBENCH, a benchmark of 486 manually annotated failed trajectories from realistic tool-use and coding scenarios. Experiments show that TRAJDEBUG outperforms LLM prompting and diagnostic baselines, and application studies demonstrate its value for trajectory repair and transferable failure memory. These results position critical error detection as a promising interface for inference-time agent improvement.

9 Limitations

We discuss the limitations of our work here, including two main aspects: (1) Although TRAJDEBUG constrains judgments with verbatim evidence and structured candidates, it still relies on LLMs for error interpretation and final attribution. Its predictions may therefore be affected by the model's reasoning ability, domain knowledge, and calibration. To reduce the concern that the gains merely come from using a stronger judge, we implement TRAJDEBUG with an open-source backbone rather than the strongest proprietary models; even under this setting, TRAJDEBUG outperforms direct prompting with frontier models and all multi-agent baselines on average. (2) As a staged pipeline, TRAJDEBUG may inherit false negatives from earlier trigger detection or state classification stages, causing the final candidate set to miss the true critical error. To mitigate this, the attribution stage allows an out-of-set prediction only under strict evidence requirements: the model must explain why the retained candidates do not account for the failure and provide the missing trigger, violated reference, origin step, and terminal footprint.

10 Ethical Considerations

We discuss three ethical considerations. (1) *Intellectual property.* We respect the licenses of all artifacts used in this work, including datasets, models, and code repositories. We will release TRAJDEBUG, the associated code, and TRAJERRBENCH under the MIT license[2]. (2) *Intended use and risk control.* TRAJDEBUG is designed to trace the error lifecycle for critical error detection in failed agent trajectories. The data used in this work is anonymized to the best of our knowledge. Since the framework may still produce incorrect predictions due to model and method limitations, users should manually verify important diagnoses before using them in high-stakes settings. (3) *AI assistance.* We used AI to refine the wording of some sentences.

[2]https://opensource.org/license/mit

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 52539001), the Shenzhen Science and Technology Program (AI 20260020), a grant from the Institute for Guo Qiang, Tsinghua University (2019GQB0003), and the 2025 Tencent Rhino-Bird Joint Research Program (JR2025TEG013). We thank the meta-reviewer and reviewers for their valuable comments.

References

2025. Graphtracer: Graph-guided failure tracing in LLM agents for robust multi-turn deep search. *CoRR*, abs/2510.10581. Withdrawn.

Anthropic. Introducing Claude Opus 4.6 [online]. 2026a.

Anthropic. Introducing Claude Sonnet 4.6 [online]. 2026b.

Adi Banerjee, Anirudh Nair, and Tarik Borogovac. 2025. Where did it all go wrong? A hierarchical look into multi-agent error attribution. *CoRR*, abs/2510.04886.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. 2026. Agentrx: Diagnosing AI agent failures from execution trajectories. *CoRR*, abs/2602.02475.

Victor Barrès, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *CoRR*, abs/2506.07982.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya G. Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025a. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya G. Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025b. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*.

Mengzhuo Chen, Junjie Wang, Fangwen Mu, Yawen Wang, Zhe Liu, Huanxiang Feng, and Qing Wang. 2026. Seeing the whole elephant: A benchmark for failure attribution in llm-based multi-agent systems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, pages 19888–19905. Association for Computational Linguistics.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948.

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

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. TRAIL: trace reasoning and agentic issue localization. *CoRR*, abs/2505.08638.

Shengda Fan, Xuyan Ye, Yupeng Huo, Zhi-Yuan Chen, Yiju Guo, Shenzhi Yang, Wenkai Yang, Shuqi Ye, Jingwen Chen, Haotian Chen, Xin Cong, and Yankai Lin. 2026. Agentprocessbench: Diagnosing step-level process quality in tool-using agents. *CoRR*, abs/2603.14465.

Tianyu Gao, Howard Yen, Jiatong Yu, and Danqi Chen. 2023. Enabling large language models to generate text with citations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 6465–6488. Association for Computational Linguistics.

Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. 2025. Who is introducing the failure? automatically attributing failures of multi-agent systems via spectrum analysis. *CoRR*, abs/2509.13782.

GLM. 2026. GLM-5: from vibe coding to agentic engineering. *CoRR*, abs/2602.15763.

Yeonjun In, Mehrab Tanjim, Jayakumar Subramanian, Sungchul Kim, Uttaran Bhattacharya, Wonjoong Kim, Sangwu Park, Somdeb Sarkhel, and Chanyoung Park. 2026. Rethinking failure attribution in multi-agent systems: A multi-perspective benchmark and evaluation. *CoRR*, abs/2603.25001.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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. *CoRR*, abs/2604.11641.

Jiaying Lu, Bo Pan, Jieyi Chen, Yingchaojie Feng, Jingyuan Hu, Yuchen Peng, and Wei Chen. 2025. Agentlens: Visual analysis for agent behaviors in llm-based autonomous systems. *IEEE Trans. Vis. Comput. Graph.*, 31(8):4182–4197.

Xuyan Ma, Xiaofei Xie, Yawen Wang, Junjie Wang, Boyu Wu, Mingyang Li, and Qing Wang. 2025. Diagnosing failure root causes in platform-orchestrated agentic systems: Dataset, taxonomy, and benchmark. *CoRR*, abs/2509.23735.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Hadar Mulian, Sergey Zeltyn, Ido Levy, Liane Galanti, Avi Yaeli, and Segev Shlomov. 2026. Agentfixer: From failure detection to fix recommendations in LLM agentic systems. *CoRR*, abs/2603.29848.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Hao Peng, Xiaozhi Wang, Jianhui Chen, Weikai Li, Yunjia Qi, Zimu Wang, Zhili Wu, Kaisheng Zeng, Bin Xu, Lei Hou, and Juanzi Li. 2023. When does in-context learning fall short and why? A study on specification-heavy tasks. *CoRR*, abs/2311.08993.

Yunjia Qi, Hao Peng, Xiaozhi Wang, Amy Xin, Youfeng Liu, Bin Xu, Lei Hou, and Juanzi Li. 2025. AGENTIF: benchmarking large language models instruction following ability in agentic scenarios. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*.

Matthew Renze and Erhan Guven. 2024. Self-reflection in LLM agents: Effects on problem-solving performance. *CoRR*, abs/2405.06682.

Gemini Team. 2023. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805.

Kimi Team. 2026. Kimi K2.5: visual agentic intelligence. *CoRR*, abs/2602.02276.

Qwen Team. 2025. Qwen3 technical report. *CoRR*, abs/2505.09388.

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 2 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

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. *CoRR*, abs/2602.23701.

Zhaohui Geoffrey Wang. 2026. Agenttrace: Causal graph tracing for root cause analysis in deployed multi-agent systems. *CoRR*, abs/2603.14688.

xAI. Grok 4 Fast [online]. 2025.

Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, Tao Ji, Tao Gui, Qi Zhang, and Xuanjing Huang. 2026. Agentprm: Process reward models for LLM agents via step-wise promise and progress. In *Proceedings of the ACM Web Conference 2026, WWW 2026, Dubai, United Arab Emirates, originally scheduled for April 13-17, 2026, rescheduled for June 29 - July 3, 2026*, pages 4184–4195. ACM.

Yizhe Xie, Congcong Zhu, Xinyue Zhang, Tianqing Zhu, Dayong Ye, Minfeng Qi, Huajie Chen, and Wanlei Zhou. 2026. From spark to fire: Modeling and mitigating error cascades in llm-based multi-agent collaboration. *CoRR*, abs/2603.04474.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. 2025a. Agentracer: Who is inducing failure in the LLM agentic systems? *CoRR*, abs/2509.03312.

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 *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where LLM agents fail and how they can learn from failures. *CoRR*, abs/2509.25370.

A Pilot Study Details

Data sources and annotators. For Pilot Study 2, we randomly sample 50 failed trajectories from WhoAndWhen (Zhang et al., 2025b) and Agent-DebugBench (Zhu et al., 2025), drawing 10 trajectories from each subset of the two benchmarks, with 1,373 steps in total. The critical-error label of each trajectory is inherited from the source dataset, since both benchmarks already provide ground-truth critical-error annotations for their failed trajectories; we therefore only annotate per-step local errors and their downstream states. All annotations are produced by three computer-science undergraduates who completed a calibration training round on a separate set of trajectories before working on the pilot data.

Detector evaluation (§2.2). For the detector comparison in §2.2, we evaluate seven advanced models: GLM-5.1 (GLM, 2026), Gemini-3.1-Pro (Team, 2023), DeepSeek-V4-Pro (DeepSeek-AI, 2025), Claude Sonnet 4.6 (Anthropic, 2026b), Claude Opus 4.6 (Anthropic, 2026a), GPT-5.4 (OpenAI, 2023), and Qwen3-235B-A22B (Team, 2025). All models are queried with the same direct-prompt template used by the direct prompting baselines (Table E.1) at temperature 0, and a prediction is counted as correct only when the predicted step index exactly matches the critical error step provided by the source dataset.

Local error step annotation (§2.2, §2.3). We recruit a pool of 20 annotators in total, all of whom completed a calibration training round on a held-out set of trajectories. Annotators were compensated at rates consistent with the prevailing market standards for comparable annotation work. Three annotators independently inspect each step. Following the annotation principles established in prior process-supervision work (Fan et al., 2026), each annotator is required to (i) judge each step strictly along the agent's own reasoning, action, and observation, without using their own task knowledge to fill in unstated steps; (ii) mark a step as erroneous only when the error can be tied to an explicit, objective trigger, such as a violated task constraint, a contradicted observation, or an internally inconsistent claim; and (iii) write a short, reproducible reason that cites the specific evidence supporting the judgment. Vague reasons (e.g., "the agent seems off-track") are rejected and the step is not added to the error set. A step enters the error set only when all three annotators independently mark it as erroneous and each provides an evidence-grounded reason; disagreements are kept as non-errors to avoid inflating the error set with subjective calls. This protocol yields 381 local error steps. The Fleiss' κ on error identification is 0.538, indicating moderate agreement and confirming that, even under an evidence-grounded protocol, identifying step-level errors in long agent trajectories remains genuinely hard for humans.

Downstream-state annotation of non-critical local errors (§2.3). For the downstream-state analysis in §2.3, the same three annotators independently classify each of the 331 non-critical local errors into one of three outcomes: *repaired*, *persistent*, or *unrepaired yet dormant*. We attach an evidence requirement to each label so that state judgments remain reproducible. A *repaired* label follows the resolution criterion in §3.3: a later step must explicitly revisit the same reference object and supersede the wrong commitment, for example by satisfying the violated constraint, acting on the corrected state, or retracting the prior wrong claim; the annotator must cite the specific repairing step. A *persistent* label requires the annotator to write an explicit reason together with the evidence from a later step that still uses or relies on the wrong commitment introduced at the error step. An *unrepaired yet dormant* label is reserved for cases where no later step depends on the erroneous content, for example a step that miscounts four search results as two when no subsequent reasoning or action references the count; the annotator must state why the wrong content is never used downstream. The boundary between *persistent* and *unrepaired yet dormant* is therefore decided by the presence of a downstream reference: persistent requires a citable later usage of the wrong content, while dormant requires explaining its absence. A label is kept only when all three annotators independently agree, following the same unanimous-agreement criterion used for local error identification. The Fleiss' κ on downstream-state labels is 0.378, lower than the agreement on error identification and consistent with the view that state judgments require additional reasoning over downstream references and are therefore harder to align across annotators.

B TRAJERRBENCH Details

B.1 Dataset Construction

We construct TRAJERRBENCH from two complementary domains: 400 rollouts on the airline and retail tasks of τ^2-Bench (Barrès et al., 2025), produced by DeepSeek-Reasoner, GPT-5, Qwen3-Max and 86 rollouts on SWE-Bench Pro (Deng et al., 2025), produced by Claude-Opus-4-6 (Anthropic, 2026a), Kimi-K2.6-Coding (Team, 2026), Gemini-3.1-Pro (Team, 2023), Grok-4.20-Beta (xAI, 2025), and GPT-5.4 (OpenAI, 2023) from the OpenHands scaffold (Wang et al., 2025). The two subsets average 29.3 and 119.7 steps per trajectory, respectively.

B.2 Annotation Details

Our annotation protocol follows the decisive-error annotation practice of WhoAndWhen (Zhang et al., 2025b). Annotators are given the task instruction, the full failed trajectory, and the final failure outcome, and are asked to identify the earliest step that directly causes the failure under the counterfactual definition in § 2.1. They do not need to execute an actual intervention; instead, they make an evidence-grounded attribution judgment from the trajectory, as in prior failure-attribution annotation.

Since both benchmarks contain long and difficult failure trajectories, we first generate model-assisted pre-annotations to help annotators understand the likely failure points. Specifically, we ask GPT-5.4, Gemini-3.1-Pro, and Claude-Opus-4-6 to annotate each trajectory. Each model is given the full trajectory, the ground-truth task specification, and the failure information returned by the original benchmark evaluation. For τ^2-Bench, this includes the unmet database constraints reported by the task environment; for SWE-Bench Pro, this includes the details of the failing test cases. Annotators then receive the same full information, together with the three model pre-annotations, as reference material during annotation. The pre-annotations are used only to support trajectory understanding; the final labels are determined by human annotators following the guideline below.

Agreement between pre-annotations and final labels. To quantify potential anchoring from model-assisted pre-annotation, Table 4 reports the exact-match rate between each model's proposed critical step and the final human-majority label. Agreement varies substantially across models and subsets, and no single model consistently determines the final labels. Moreover, the pre-annotation setting provides benchmark failure details to aid human annotation, whereas all evaluated direct-prompting baselines receive only the failed trajectory and failure signal, without gold outcomes or auxiliary debugging information.

Table 4: Exact-match agreement between model-assisted pre-annotations and final human-majority labels.

Subset	Pre-annotation model	Exact match
τ^2-Bench (N=400)	Claude Opus 4.6	64.5%
	Gemini 3.1 Pro	76.8%
	GPT-5.4	53.2%
SWE-Bench Pro (N=86)	Claude Opus 4.6	48.8%
	Gemini 3.1 Pro	31.4%
	GPT-5.4	55.8%

The standardized guideline contains three principles. First, an annotated step must contain an explicit mistake supported by the available evidence, such as violating the task instruction, contradicting a previous observation, misusing a tool result, or making an internally inconsistent claim. Second, the mistake must be failure-responsible: annotators should not choose local mistakes that are later corrected or errors whose downstream effect never affects the final outcome. Third, if multiple failure-responsible mistakes are present, annotators choose the earliest such step, following the principal-cause convention used by WhoAndWhen (Zhang et al., 2025b). Each annotator also writes a short explanation for the selected step so that disagreements can be checked against concrete evidence.

Annotator pool and labelling protocol. We recruit a pool of 20 annotators in total, all of whom completed a calibration training round on a held-out set of trajectories covering both τ^2-Bench and SWE-Bench Pro before working on the released data. Annotators were compensated at rates consistent with the prevailing market standards for comparable annotation work. Because decisive-error localization is inherently ambiguous in long agent trajectories, each trajectory is independently annotated by 3 annotators and the final label is taken by majority vote: a label is kept when at least 2 of the 3 annotators agree. We apply the same three-annotator majority-vote protocol to the reference-category labels (*Task Conflict*, *History Conflict*, and *Intra-Step Conflict*) and to the execution-phase labels (*planning*, *reasoning*, *action*, *observation*, *verification*).

Inter-annotator agreement. Table 5 reports inter-annotator agreement for the critical-error step, the execution-phase label, and the reference-category label on each subset of TRAJERRBENCH. On τ^2-Bench, the critical-error step reaches almost-perfect agreement (Fleiss' κ=0.91), while the execution-phase and reference-category labels reach substantial and fair agreement respectively. For SWE-Bench Pro, we initially annotate 110 trajectories, including ambiguous cases on which no two annotators select the same critical step. The released benchmark and all experiments use only the 86 trajectories for which at least two of three annotators agree on the critical step; Table 5 reports agreement on this retained set. Agreement is lower than on τ^2-Bench, reflecting the longer and more code-heavy trajectories: the critical-error step reaches substantial agreement (Fleiss' κ=0.674), while execution-phase and reference-category agreement is moderate and fair, respectively. Despite this, 98.2% of τ^2-Bench trajectories and 91.9% of SWE-Bench Pro trajectories receive a reference-category label on which at least 2 of 3 annotators agree, indicating that the majority-vote protocol yields a usable label on the overwhelming majority of trajectories.

Table 5: Inter-annotator agreement on TRAJERRBENCH. "Pairwise" denotes pairwise agreement.

Subset	Label	Fleiss' κ	Pairwise
τ^2-Bench	Critical-error step	0.91	91.2%
	Execution phase	0.73	84.0%
	Reference category	0.41	68.9%
SWE-Bench Pro	Critical-error step	0.67	66.7%
	Execution phase	0.42	61.8%
	Reference category	0.38	52.4%

B.3 Distribution of Critical Errors

Figures 6 and 7 show the joint distribution of critical errors over the execution phase, error subtype, and reference category for the τ^2-Bench and SWE-Bench Pro subsets of TRAJERRBENCH respectively. The flows in each diagram are read from left (execution phase) to right (reference category), with the middle column showing the fine-grained error subtype.

Execution phase. Among critical errors with an agreed execution-phase label, τ^2-Bench is heavily skewed toward reasoning (60.7%), followed by planning (14.8%) and observation (13.5%), with action (6.1%) and verification (4.8%) contributing the rest. This indicates that on tool-use tasks, critical errors more often arise when the agent interprets accumulated context than when it issues an isolated action. In SWE-Bench Pro, reasoning is still the largest phase (57.1%), followed by planning (15.6%), verification (14.3%), and action (13.0%). This shift is consistent with the patch-and-test nature of software-engineering trajectories: the agent spends comparatively more time editing code and validating it, and a larger share of failures appears in those phases.

Error subtype. The middle column reveals which fine-grained agent behaviour is most failure-prone within each phase. On τ^2-Bench, the top three subtypes `reason.InvalidInference` (22.8%), `reason.WrongChoice` (22.2%), and `reason.MissingAssumptionCheck` (14.5%) jointly account for nearly 60% of critical errors, indicating that the dominant failure mode is inference over already-observed context rather than acting on incorrect observations. On SWE-Bench Pro, the distribution becomes more peaked: `reason.WrongChoice` alone accounts for 40.7% of critical errors, followed by `plan.BadDecomposition` (14.0%) and `verify.NoVerification` (9.3%), suggesting that failures cluster around selecting the wrong implementation path and skipping verification before submission.

Reference category. On the right column, both subsets are dominated by external references rather than intra-step contradictions: among critical errors with an agreed reference-category label, *Task Conflict* and *History Conflict* together cover 98.7% of τ^2-Bench and 97.5% of SWE-Bench Pro critical errors, while *Intra-Step Conflict* accounts for only 1.3% and 2.5%, respectively. The two subsets differ in which external reference is violated more often: τ^2-Bench is balanced between *Task Conflict* (52.4%) and *History Conflict* (46.3%), reflecting failures that violate either explicit user constraints or earlier tool observations; SWE-Bench Pro is more strongly skewed toward *Task Conflict* (73.4% vs. 24.1% *History Conflict*), consistent with code-task failures where the violated reference is most often the task instruction or expected behaviour rather than an intermediate observation.

Joint patterns. Reading the diagrams jointly, two patterns stand out. First, the reasoning phase is the dominant entry channel into both *Task Con-*

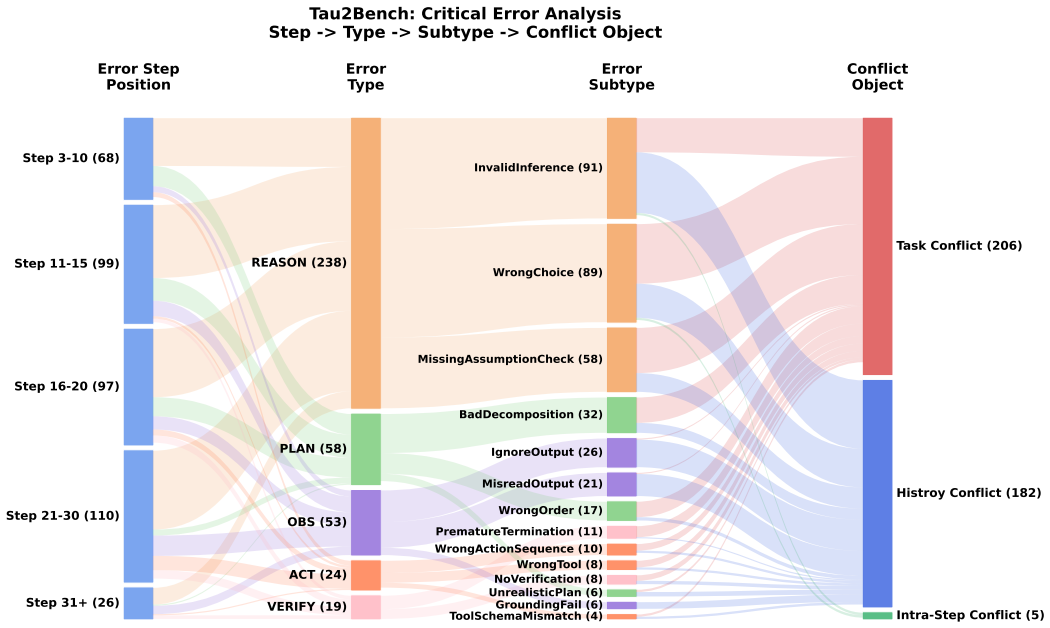

Figure 6: Distribution of critical errors in τ^2-Bench. Flows go from execution phase (left) to error subtype (middle) to reference category (right).

flict and *History Conflict* in both subsets, suggesting that strengthening evidence-grounded reasoning is the single highest-leverage intervention for critical-error reduction. Second, the planning phase flows almost exclusively into *Task Conflict*, whereas observation-phase failures flow almost exclusively into *History Conflict*; this clean separation reflects the underlying definitions and supports using the reference category as a complementary axis to the execution phase when analysing failure mechanisms.

Table 6: Overview of the case study trajectory.

Field	Value
Dataset	τ^2-Bench(airline)
Agent model	`qwen3-max`
Trajectory length	49 steps
Final outcome	Failed (transferred to human)
Detected triggers	15
Detected instances	9
Candidate instances	4 (in $\mathcal{F}(\tau)$)
TRAJDEBUG critical step	25
Human-annotated step	25

C Case Studies

We present an end-to-end case to illustrate how TRAJDEBUG localizes the critical error within a long failed trajectory. The example is drawn from the airline subset of τ^2-Bench, where `qwen3-max` acts as the agent and the user requests several modifications to an existing reservation under a budget cap of $200. The trajectory contains 49 steps and ends with the agent transferring the conversation to a human operator. TRAJDEBUG selects step 25 as the critical error, matching the human annotation exactly.

C.1 Case Overview

Task. User `yara_garcia_1905` asks the agent to modify reservation `HXDUBJ`: change the outbound flight to a direct flight one day later, push the return flight back by one day, and, within a $200 budget, consider upgrading to business class and adding checked baggage. The system policy explicitly states that all flights in the same reservation must share the same cabin class.

Trajectory summary. The trajectory can be decomposed into three phases. *Flight modification (steps 0–18).* The agent retrieves the reservation, searches for direct flights, and successfully updates the booking to the new outbound (`HAT072`) and return (`HAT278`) flights in economy. *Upgrade negotiation (steps 19–24).* The agent computes the cost of upgrading both flights to business as $408, which exceeds the user's $200 budget; the user then asks whether only the outbound flight can be

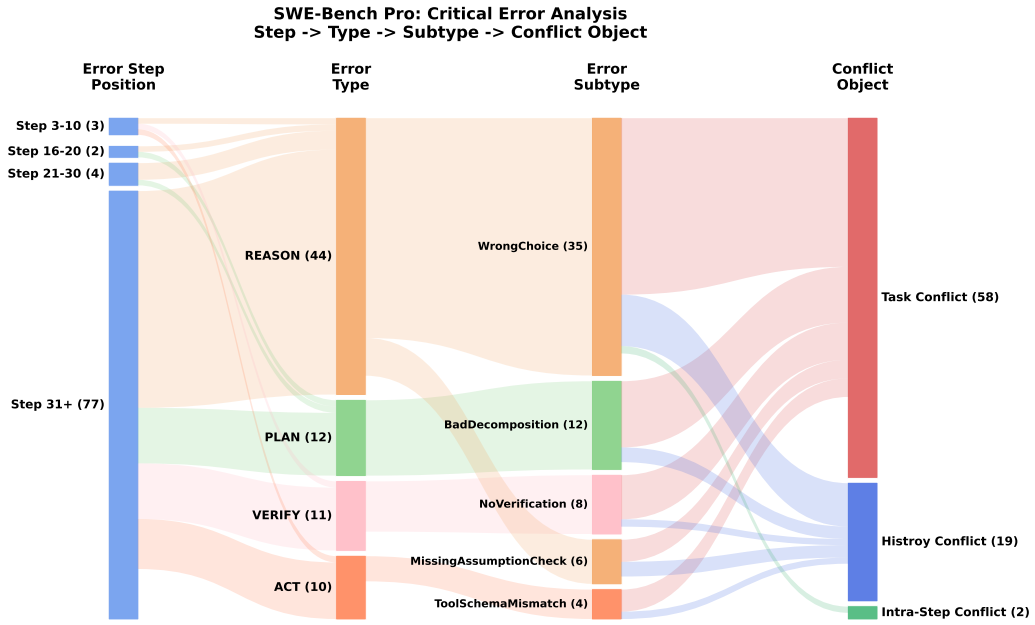

Figure 7: Distribution of critical errors in SWE-Bench Pro. Flows go from execution phase (left) to error subtype (middle) to reference category (right).

upgraded. *Cascading failure (steps 25–48).* The agent commits to the partial-upgrade plan, executes the corresponding API call, and enters a chain of payment failures and repeated retries that finally end in a hand-off to a human agent.

C.2 The Critical Error

Step 25 (assistant) — Critical Error

Upgrading only the outbound flight (HAT072) to business class costs $282. Since your current outbound flight costs $130, the upgrade requires an additional $152. Given your budget of $200, this upgrade is within your budget. As a gold member, you will still enjoy two free checked bags on both flights. Please confirm whether you would like to proceed with upgrading only the outbound flight to business.

The agent commits to a plan that violates the reservation-level uniform-cabin policy stated in the system prompt. The trigger detector flags this step as a *Task Conflict* in the *planning* phase with the wrong commitment "upgrade only the outbound flight to business" and the violated reference being the policy "all flights in the same reservation must share the same cabin class". Because this commitment cannot be repaired without retracting the plan, the resulting error instance has a *semantic* terminal footprint and the state label *Manifest Active*, which keeps it in the candidate set $\mathcal{F}(\tau)$.

C.3 Detected Error Instances

TRAJDEBUG clusters the 15 per-step triggers into 9 object-anchored error instances. Table 7 lists them with origin step, state label, and a short description. Four instances (origin steps 13, 25, 33, 39) survive state-based filtering and form the candidate set $\mathcal{F}(\tau)$.

C.4 Error Propagation and Attribution

Although several earlier instances contain genuine local mistakes, only the critical instance at step 25 has a wrong commitment that directly causes the terminal failure. The propagation path is: (i) at step 27, the agent issues `update_reservation_flights` with `cabin=business` but provides the return flight at its economy price, so the system charges both flights as business ($282+$443=$725), exceeding the gift-card balance; (ii) at step 31, the agent switches to another gift card and reissues the same invalid call, which again fails; (iii) the agent then misattributes the failure to a payment-side problem and enters a payment-retry chain (instances 5 and 8, steps 33–43), exhausting the remaining budget before transferring to a human at step 45.

Within the candidate set $\mathcal{F}(\tau)$, the causal attribution stage compares: (a) instance 1 (budget debt, origin step 13): a financial-calculation error that

Table 7: Detected error instances in the case-study trajectory. Instances marked with $\star$ form the candidate set $\mathcal{F}(\tau)$, and the row marked † is the critical instance selected by TRAJDEBUG.

ID	Origin	Triggers	Reference category	State label	Description
0	9	9	Task Conflict	Clean Resolution	Flight search ignores the user-specified 8 am–9 pm time window, but the next turn still yields a feasible option.
$1^\star$	13	13, 19, 23	History Conflict	Costly Resolution (budget debt)	Financial computations omit the non-refundable $30 insurance fee, propagating through the refund and upgrade-cost estimates.
2	15	15	Task Conflict	Clean Resolution	The agent incorrectly claims that travel insurance waives change fees.
3	21	21	Task Conflict	Clean Resolution	The agent queries user details when the budget is already obviously insufficient.
$4^{\star\dagger}$	25	25, 27, 31	Task Conflict	Manifest Active (semantic)	Commits to upgrading only the outbound flight, violating the reservation-level uniform-cabin policy.
$5^\star$	33	33, 41	Task Conflict	Manifest Active (semantic)	Repeatedly proposes split payment in a single `update_reservation` call, which the API does not support.
6	35	35	History Conflict	Clean Resolution	Attempts to pay with a certificate whose balance ($150) is below the required amount.
7	37	37	History Conflict	Clean Resolution	Reuses a payment method that has already failed in a prior step.
$8^\star$	39	39, 43	History Conflict	Manifest Active (semantic)	Repeatedly retries the same failed `gift_card_1646646` payment without addressing the underlying cause.

wastes budget but does not by itself prevent task completion; (b) instance 4 (semantic, origin step 25): a plan that violates a system constraint and makes the requested upgrade infeasible regardless of payment choice; (c) instances 5 and 8 (semantic, origin steps 33 and 39): downstream payment-side errors that occur only because the agent is already trying to execute the impossible plan committed at step 25. Removing instance 4 would eliminate instances 5 and 8 and let the agent either accept the over-budget upgrade or decline it cleanly, while removing instance 1 would still leave the partial-upgrade commitment unresolved. Step 25 is therefore selected as the critical error step $t^\star$.

C.5 Discussion

The case illustrates how the three stages of TRAJDEBUG support long-horizon failure analysis. First, error trigger detection grounds local mistakes in explicit conflicts, such as the uniform-cabin policy violation at step 25. Second, error state classification groups repeated triggers into object-anchored instances and distinguishes corrected errors (e.g., instances 0, 2, 3) from terminal-relevant states such as budget debt and manifest active commitments. Third, causal attribution compares the remaining candidates by their role in the failure rather than by temporal order alone, which is why TRAJDEBUG selects the later step 25 instead of the earlier financial error.

D TRAJDEBUG Implementation Details

All experiments use Qwen3-235B-A22B-Thinking (Team, 2025) as the base model, with temperature fixed at 0 for reproducibility. Following the error-lifecycle tracking perspective in §3.3, TRAJDEBUG operationalizes critical error detection through three stages: error trigger detection, error state classification, and causal attribution. To handle long trajectories, we use the multi-granularity compression scheme in §3.1. The high-detail view (th1) keeps up to 3000 characters, the medium-detail view (th2) keeps up to 1200 characters, and the low-detail view (th3) keeps up to 600 characters.

Each downstream stage uses these views according to its own evidence requirements. In the *error trigger detection* stage, the current step and his-

tory within two steps are rendered with th1, history three to five steps away is rendered with th2, and more distant history is rendered with th3. Only the history before the current step is included in this view; the current step is provided separately in th1 for local verification. In the *error state classification* stage, trigger steps are rendered with th1 and all other steps are rendered with th3 as background context for instance clustering; for state classification, the origin step, the last trigger step, and a fixed number of following steps are always rendered with th1. The remaining context is rendered by falling back in the order th1 → th2 → th3 → raw, with inline annotations indicating the associated error instance. For *budget debt*, we use a Python script to check whether the distance between the origin step and the repair step exceeds 50% of the trajectory length. In the *causal attribution* stage, the full trajectory is rendered with the same fallback order th1 → th2 → th3 → raw, with an additional 800-character truncation limit.

We list the complete system prompts for the compression step and the three stages of TRAJDEBUG: error trigger detection, error state classification, and causal attribution.

Multi-Granularity Compression Prompt.

Stage A — Trajectory Compression

```
You are compressing a single trajectory step at THREE
      different compression levels for downstream agent error
       diagnosis. Produce a detailed version (th1), a
      moderate version (th2), and a concise version (th3) of
      the same step content.

1. PRESERVE SEMANTIC CORE: Compression removes ONLY redundant
      phrasing, verbose boilerplate, repetitive expressions,
      and filler words. The core meaning of EVERY sentence
      must survive. When in doubt, keep more rather than less.


2. CONSTRAINTS AND PLANS ARE CRITICAL: If the content contains
       ANY of the following, their key information MUST be
      preserved across ALL three tiers:
   - Constraints (time limits, budget limits, "do NOT do X"
      prohibitions, "must do Y first" preconditions, required
       formats, allowed/disallowed actions)
   - Plans (specific action items, ordered steps, goals, sub-
      goals, milestones)
   - Requirements for subsequent steps (instructions that
      govern future behavior, expected outputs, acceptance
      criteria)
   - Conditional logic ("if X then Y", fallback strategies,
      error handling rules)
   Drop surrounding prose before dropping any constraint or
      plan item.

3. THREE-TIER COMPRESSION LEVELS:
   - th1 (detailed, <={th1_max} chars per field): Preserve all
       meaningful details. Remove only obvious redundancy,
      repeated phrasings, and decorative formatting. Keep
      every distinct fact, constraint, action, and
      observation.
   - th2 (moderate, <={th2_max} chars per field): Merge
      related information. Omit secondary details and
      elaborations. But KEEP all constraints, key decisions,
      action outcomes, error messages, and critical
      observations.
   - th3 (concise, <={th3_max} chars per field): Retain only
      the most essential actions, results, errors, and
      constraints. Use minimal wording. Still preserve all
      hard constraints and critical factual anchors.

================ PRESERVE VERBATIM =============

Preserve the FOLLOWING classes of tokens VERBATIM (copy them
      exactly as written - do not translate, paraphrase,
      reorder, round, or abbreviate):

- Proper nouns and named entities (people, places, brands,
      franchise/universe names such as "Disney", "Marvel", "
      MCU", book/paper/file names).
- Product / part / SKU / ASIN / ISBN / DOI / arXiv identifiers
       and any alphanumeric IDs the environment returned.
- Numbers with their units exactly as given (prices like "$22
      .50", quantities like "3 items", measurements like "45
      min", percentages, counts).
- Years and dates (including citation years like "(1976)", ISO
       dates, "Q3 2023"-style period labels).
- URLs, file paths, CLI flags, API parameter names and their
      literal values.
- Quoted constraint phrases from the task statement (e.g. "
      avoid Tue", "refundable", "under $25", "first page only
      ") - keep them inside quotation marks if the original
      had them.
- Tool / function names and their exact argument keys.
- Error messages, status codes, truncation markers ("has_more:
       true", "output truncated", "403 Forbidden").

You MAY rewrite the connective prose between these tokens to
      shorten the field; you MAY NOT rewrite the tokens
      themselves. If preserving them verbatim would push a
      field past the length cap, DROP the least-important
      surrounding prose first and keep the verbatim tokens.
```

Error-Trigger Annotation Prompt.

Stage 1 — Error Trigger Detection

```
You are an error-trigger annotator for one step of an LLM-
      agent trajectory. For the CURRENT STEP, decide which
      trigger tags (if any) fire. Each fired trigger falls
      into one of four categories:

  cat-1 = the step conflicts with the TASK
  cat-2 = the step conflicts with VISIBLE CONTEXT (env / tool
      output / system feedback / prior environment-rendered
      fact)
  cat-3 = the step is INTERNALLY INCONSISTENT - its claim
      contradicts something the same message states verbatim,
       or contradicts the agent's own prior plan / reflection
       / memory
  env   = the agent's tool call at THIS step is correct but
      the environment response is anomalous;


=== CHECKLIST (A) - conflict with TASK (cat-1) ===

  [ ] The step's plan permanently drops a sub-task / sub-goal
      / deliverable the TASK explicitly requires, with no
      TODO / "later" / sequencing intent in the step itself,
      and HISTORY does not already cover it. Phased execution
       that still keeps deferred items in scope is NOT a
      trigger.
     -> plan.BadDecomposition

  [ ] The step's plan schedules a sub-task before another sub-
      task whose output it depends on, and TASK itself
      dictates this ordering.
     -> plan.WrongOrder

  [ ] The step depends on a tool / API / permission / data
      source that TASK or the environment does not provide.
     -> plan.UnrealisticPlan

  [ ] Double-anchor required. (i) HISTORY contains a verbatim
      line where the agent was previously on-constraint; (ii)
       THIS step contains a verbatim line that explicitly
      narrows / widens / replaces a TASK constraint. Both
      anchors must be in reference_quote. Gradual multi-step
      drift without an explicit single-step swerve is NOT a
      trigger here.
     -> plan.GoalDrift

  [ ] TASK explicitly requires verify / check / confirm of X,
      and THIS step delivers the final answer without
      performing that check. Implicit verification
      expectations are NOT a trigger here.
     -> verify.NoVerification

  [ ] The step declares completion / returns the final answer
      / stops the trajectory while a TASK-required
      deliverable is still unmet.
     -> verify.PrematureTermination

  [ ] The step performs an action that TASK or environment
```

```
      rules explicitly forbid (safety policy, permission
      boundary, forbidden endpoint, forbidden operation).
     -> act.UnsafeOrForbiddenAction

=== CHECKLIST (B) - conflict with VISIBLE CONTEXT (cat-2) ===

  [ ] The step makes an explicit decision and cites supporting
       evidence, but the citation is distorted (wrong value,
      wrong field, mis-paraphrased), and the wrong reading
      drives the decision.
     -> reason.WrongChoice

  [ ] The step cites visible evidence correctly, but
      generalises the conclusion beyond what the evidence
      supports.
     -> reason.InvalidInference

  [ ] The step relies on a prior tool call / observation /
      fact that does NOT actually appear in HISTORY (
      fabricated cross-message reference into a context that
      should exist).
     -> reason.MissingAssumptionCheck

  [ ] (a) The chosen tool or data source is unsuited to the
      established sub-goal; OR (b) plan-action mismatch: this
       step commits to plan A but issues action B.
     -> act.WrongTool

  [ ] A tool call is syntactically malformed against the
      visible schema or prior successful calls.
     -> act.ToolSchemaMismatch

  [ ] The action order violates a workflow established by
      HISTORY or the visible UI / env state machine.
     -> act.WrongActionSequence

  [ ] The step misreads its own immediate tool / env output:
      treats an error as success, swaps fields, mis-
      identifies the returned entity.
     -> obs.MisreadOutput

  [ ] Subject to (C4): information visible in current or prior
       tool / env output is silently ignored where it is
      clearly needed, OR a summary/memory block drops load-
      bearing facts.
     -> obs.IgnoreOutput

  [ ] The step treats a partial / pending / truncated response
       as final.
     -> obs.TimingIssue

  [ ] The step binds intent to the wrong visible entity.
     -> obs.GroundingFail

  [ ] The step is performing verification but the criterion
      does not match the sub-goal in HISTORY.
     -> verify.WrongVerification

  [ ] The step retries the same tool / query / action that
      already produced bad / empty / error in HISTORY, with
      no relevant env state change and no strategy change.
      Subject to (C4).
     -> verify.InfiniteRetry

=== CHECKLIST (C) - internally inconsistent (cat-3) ===

  [ ] The conclusion contradicts something the SAME message
      states verbatim, OR contradicts a prior agent
      reflection / memory / claim.
     -> reason.InvalidInference

  [ ] The step picks the wrong row / column / item from a list
       / table that is verbatim visible in the CURRENT
      MESSAGE itself.
     -> obs.MisreadOutput

  [ ] The step asserts a concrete factual claim from the agent
       's parametric knowledge, where the claim is concrete/
      falsifiable, NOT supported by TASK/HISTORY/CURRENT
      MESSAGE, clearly wrong on widely-known ground truth,
      and load-bearing for this step's conclusion.
     -> reason.MissingAssumptionCheck

=== CHECKLIST (D) - environment-side anomaly (env) ===

env triggers fire when the agent's tool call at this step is
      otherwise correct, but the immediately following
      environment response shows the anomaly. Set attribution
       = "env".

  [ ] The search / page response is hijacked by an ad vignette
       / overlay.
     -> env.AdOverlayHijack

  [ ] The model API / platform content filter rejects a
      syntactically well-formed query.
     -> env.ContentFilterBlock

  [ ] The tool call is admissible but the returned observation
       is structurally insufficient because the tool is
      degenerate or blind on this target.
     -> env.ToolExtractorDegenerate

  [ ] The observation contains verbatim evidence of HTTP 429 /
       5xx / timeout / connection reset / rate limited.
     -> env.RateLimitOrTransient

  [ ] The tool returns structurally empty / constant / payload
       identical to a prior call.
     -> env.EmptyOrRepeatedPayload

=== CONFIDENCE ===

For each trigger fill `confidence`:
  - high   : (C1)+(C2) are both crisp; wrong_content_quote and
       reference_quote are unambiguous and the conflict is
      direct.
  - medium : both quotes exist verbatim, but reference_quote
      needs light alignment.
  - low    : inferring from a pattern; verbatim anchors are
      weak or partial.
Always fill `confidence_reasoning` with 1-2 sentences.

 Two verbatim quotes per trigger:
 - wrong_content_quote : verbatim substring of the CURRENT
      STEP carrying the wrong content;
 - reference_quote     : verbatim substring of TASK / HISTORY
      / CURRENT STEP that is the I being conflicted.
 Paraphrase, summary, or composite quotes are not acceptable.
      If either quote cannot be cited verbatim, do NOT emit
      the trigger.

=== OUTPUT FORMAT ===

Emit a single JSON object:

{
  "triggers": [
    {
      "step": <int>,
      "category": "cat-1" | "cat-2" | "cat-3" | "env",
      "taxonomy_tag": "<one of the 25 tags listed in the
       checklists above>",
      "attribution": "agent" | "env",
      "wrong_content_quote": "<verbatim from CURRENT STEP, or
       for the upstream-env back-reference verbatim from the
       earlier HISTORY step>",
      "reference_quote": "<verbatim from TASK / HISTORY /
       CURRENT STEP>",
      "confidence": "high" | "medium" | "low",
      "confidence_reasoning": "<1-2 sentences>"
    }
  ]
}
```

Error-Instance Clustering Prompt.

Stage 2 — Error State Classification: Instance Clustering

```
You are an error-instance clustering annotator. You receive a
      list of per-step error TRIGGERS detected on one
      trajectory. Group those triggers into ERROR INSTANCES,
      where one instance corresponds to "the same underlying
      erroneous content / behavior" repeated or reflected
      across one or more steps.

=== TERMINOLOGY ===

Each trigger contradicts a "conflict object" - the thing the
      wrong content is conflicting with. Following the spec
      we call this object I. Concretely, I can be one of:
  - a TASK clause (sub-task, deliverable, constraint);
  - a HISTORY object (a specific physical / virtual location,
      a specific tool output, a specific observation, a
      specific memory entry, a specific prior tool-call
      signature);
  - a SAME-MESSAGE premise (operands, listed items, restated
      premise);
  - a prior agent-self statement (plan, reflection, memory).

Two triggers are about the SAME I when they are conflicting
      with the same underlying thing.
Two triggers are about DIFFERENT I's when they conflict with
      different observations / different TASK clauses /
      different self-claims, or happen to share the
      taxonomy_tag string but the underlying object is
      different.

=== CLUSTERING RULES ===
```

```
(I-a) Same step, same I, multiple modules -> ONE instance.
(I-b) Same step, DIFFERENT I -> MULTIPLE instances.
(I-c) Across steps, same I, never repaired in between -> ONE
       instance with origin_step = first occurrence.
(I-d) Across steps, same I, but repaired and then re-fires
       later -> SPLIT into TWO instances. If you cannot
       determine whether a repair happened, conservatively
       keep them in ONE instance.
(I-e) "Fabricated I" (an I the agent invented out of thin air)
        is identified by the verbatim claim that introduced it.


CLUSTERING PRINCIPLE: cluster by the SPECIFIC CONSTRAINT
       OBJECT I being violated, NOT by "same type of erroneous
        behavior".

=== PRECISION REQUIREMENTS FOR `what_is_being_violated` ===

Write `what_is_being_violated` as a CONCRETE pointer to the
       exact object being contradicted.
  - For TASK-clause violations: quote or paraphrase the
       specific constraint / deliverable / format requirement.
  - For HISTORY-object violations: quote or paraphrase the
       specific observation / tool output / location / URL,
       and reference the step where it first appeared.
  - For SAME-MESSAGE premise violations: name the operands /
       listed items being misused.
  - For prior-self violations: quote or paraphrase the prior
       plan / reflection / memory statement being contradicted.


=== ONE-SHOT PROBE GUIDANCE ===

If a trigger fires at exactly ONE step and the agent at later
       steps neither references the same wrong belief nor
       takes another action against the same I, keep it as a
       STANDALONE singleton instance. Do NOT merge it with
       other unrelated triggers.

=== HARD CONSTRAINT ===

Triggers with attribution="agent" and triggers with
       attribution="env" CANNOT be merged into the same
       instance.
Every input trigger index must end up in EXACTLY ONE instance'
       s trigger_indices.

=== ENV INSTANCE MERGING ===

For triggers with attribution="env", apply same-source merging
        when two env triggers share at least one of:
  - Same registrable domain;
  - Same tracking parameter key+value;
  - Same HTTP error code;
  - Same failure keyword: "rate limit", "timeout", "content
       filter", "empty payload", "captcha", "access denied", "
       not found";
  - Same taxonomy_tag.

Do NOT merge env triggers that share only a generic/CDN domain
        (google.com, bing.com, youtube.com, etc.).

=== TRAJECTORY CONTEXT ===

You will also receive a TRAJECTORY STEP CONTENT block. Steps
       that fired at least one trigger are shown at HIGH
       detail (th1 compression); steps that did not fire any
       trigger are shown at LOW detail (th3 compression) as
       background for later state classification.

=== OUTPUT FORMAT ===

{
  "instances": [
    {
      "instance_id": <int, 0-indexed across this trajectory>,
      "trigger_indices": [<int>, ...],
      "origin_step": <int, smallest step among trigger_indices
       >,
      "last_trigger_step": <int, largest step among
       trigger_indices>,
      "attribution": "agent" | "env",
      "category": "cat-1" | "cat-2" | "cat-3" | "env",
      "error_content": "<one-sentence natural-language
       description of the same erroneous content/behavior>",
      "what_is_being_violated": "<one-sentence description of
       the I being violated>",
      "merged_reasoning": "<2-3 sentences explaining why these
        triggers share the same I>"
    }
  ]
}

If the input triggers array is empty, emit  {"instances": []}.
```

Error State Classification Prompt.

Stage 2 — Error State Classification: State Labeling

```
You are an instance-state annotator. You receive ONE error
       instance and the FULL trajectory. Upstream trigger
       detection is high-recall and may flag exploratory or
       weakly committal steps; your job is to perform the
       commitment-strength recheck, the repair / state
       determination, and the terminal-connection audit
       defined in the methodology document.

=== KEY DEFINITIONS ===

  origin_step       = the first trigger step selected for this
        instance.
  qualified_origin_step = the FIRST step in [origin_step, T]
       at which the agent makes an OBSERVABLE WRONG COMMITMENT
        for this instance's violated_object. May equal
       origin_step, may be a later step, may be null.
  last_trigger_step = the latest step at which the same
       instance re-fired upstream.
  terminal step T   = the last message index in the trajectory.

  I (violated_object) = the concrete object the instance is
       violating (a TASK clause / a HISTORY object / a same-
       message premise / a prior agent-self statement).

=== TASK 1 - COMMITMENT-STRENGTH RECHECK ===

Upstream trigger detection is high-recall: a trigger may fire
       at an early probe or initial plan step even when the
       agent had no way yet to know the action was wrong. Your
        job is to classify each origin_step into one of three
       commitment strengths so that `qualified_origin_step`
       reflects the FIRST step at which the agent observably
       persisted in the wrong belief despite already-visible
       evidence.

A commitment to error requires BOTH:
  (a) the agent takes an observable binding action or states a
        non-hedged belief that contradicts the violated_object
        I, AND
  (b) by that step, the evidence contradicting I is already
       visible in the prior trajectory.

Classify the upstream origin into ONE of:

  origin_commitment_status = "explicit_wrong_commitment"
    BOTH (a) and (b) hold at origin_step. Set
       qualified_origin_step = origin_step.

  origin_commitment_status = "weak_signal"
    EITHER (a) holds at origin_step but (b) does NOT (
       conflicting evidence not yet observable), OR (a) is
       only hedged at origin_step.
    There MUST be a clearly LATER step M (origin_step < M <= T
       ) at which the agent, now able to see the conflicting
       evidence, still takes a binding action that contradicts
        I.
    Set qualified_origin_step = M, set origin_relocated = true.


  origin_commitment_status = "pure_exploration"
    origin_step is plain exploration / setup / a reasonable
       first guess, AND no step in (origin_step, T] satisfies
       the weak_signal relocation threshold.
    Set qualified_origin_step = null, set
       exploration_suppressed = true.

SINGLE-TRIGGER SHORTCUT. If the instance has exactly ONE
       trigger (origin_step == last_trigger_step) AND the
       instance is cat-2 or cat-3 (NOT cat-1 or env), the
       default label is `pure_exploration`, UNLESS a verbatim
       quote from a step strictly after origin_step shows the
       agent referencing or committing to the same wrong
       object.

  HARD CAT-1 RULES:
    1. cat-1 instance MUST NOT have origin_commitment_status =
        pure_exploration.
    2. cat-1 instance MUST NOT have state = dormant.

  HARD ENV RULES:
    1. An env instance MUST NOT have origin_commitment_status
       = pure_exploration UNLESS BOTH: (a) the environment
       self-recovers on the next judgable step, AND (b) the
       agent's next action is consistent with the recovered
       output.
    2. For env instances, wasted steps should be counted
       against the VIOLATED sub-goal object remaining
       unresolved.

=== TASK 2 - REPAIR / STATE DETERMINATION ===
```

```
  fixed_at_step_<N>
    There is a step N with anchor < N <= T at which the agent'
       s behavior satisfies the repair criterion for this
       instance's category.

  cat-1 : at step N the agent touches the violated TASK clause
        AND the behavior is consistent with that constraint.
  cat-2 : at step N the agent touches the violated CONTEXT
       object AND the behavior is consistent with the latest
       visible state.
  cat-3 : at step N the agent explicitly retracts or corrects
       the prior wrong claim.
  env   : (env-fix-1) the agent detects the env anomaly and
       switches to a MATERIALLY DIFFERENT strategy; OR (env-
       fix-2) the env recovers AND the agent's subsequent
       behavior is consistent with the recovered output.

  HARD ENV-FIX RULES:
    (R1) The following are NOT a strategy switch: clicking
       back; retrying the same URL/query/tool; reloading the
       same hijacked/errored page.
    (R2) Same-source RE-FIRE invalidates a prior fix.
    (R3) A genuine env-fix-1 requires BOTH (a) the agent
       explicitly recognises the env anomaly, AND (b) the next
        observable action targets a DIFFERENT path.

=== TASK 3 - TERMINAL-CONNECTION AUDIT (FAILURE-PATH TEST) ===

Decide whether THIS instance lies on the actual observable
       path that took the trajectory to its terminal failure.

  FAILURE-PATH MEMBERSHIP. Does any of the following appear on
        the path that produced the observed terminal failure?
         (i)   the instance's wrong commitment is still active
        at T, OR re-fires after qualified_origin_step;
         (ii)  the instance is reflected in the terminal
       answer, terminal action, terminal failure state;
         (iii)  this instance produced an irreversible state
       change visible at T.
       If ANY of (i)-(iii) holds -> the instance IS on the
       failure path.

Decide terminal_connection, one of:
  "irreversible"    - Q1(iv) holds.
  "semantic"        - Q1(i) or Q1(ii) holds.
  "none"            - Q2 fully passes, or Q1 has no evidence
       at all.

=== TASK 4 - RESOURCE EFFECT ===

Fill `resource_effect` whenever the instance wasted ANY steps
       in (qualified_origin_step, T].

Resource effect schema:
  {
    "wasted_steps": [<list of step indices wasted on I>],
    "wasted_step_count": <int>,
    "last_referencing_step": <int or null>
  }

=== OUTPUT FORMAT ===

Return STRICT JSON with EXACTLY these keys:

{
  "instance_id": <int, copy from input>,
  "origin_commitment_status": "explicit_wrong_commitment" | "
       weak_signal" | "pure_exploration",
  "qualified_origin_step": <int or null>,
  "fix_status": "active" | "fixed_at_step_<N>",
  "fix_evidence_quote": "<verbatim quote or null>",
  "terminal_connection": "irreversible" | "semantic" | "
       budget_debt" | "none",
  "chain_membership": <bool>,
  "resource_effect": <object or null>,
  "chain_explanation": "<2-4 sentences. MUST include: (1) the
       trajectory's terminal failure mode in one phrase, (2)
       which of Q1(i)-(iii) holds OR the named dominating
       cause (3) one verbatim evidence quote.>"
}
```

Causal Attribution Prompt.

Stage 3 — Causal Attribution

```
You are an expert at identifying the SINGLE critical step that
        caused a failed agent trajectory.

An error is critical if:
- It represents the ROOT CAUSE that made task success
       impossible
- It caused a cascade of subsequent errors
- The trajectory could have succeeded if THIS specific error
       had not occurred
- IMPORTANT: Correcting this specific error would
       fundamentally change the trajectory toward success

# Hints
1. Consider the ENTIRE trajectory from a global perspective -
       understand the task goal and how the agent's path
       diverged from success.
2. Early exploration steps (steps 1-3) are often normal and
       should NOT be marked as critical unless they clearly
       introduce the root cause.
3. Prefer the first step where the agent had enough reasonable
        information to proceed correctly but nevertheless
       introduced the error locally.
   This includes producing an incorrect output,
       misinterpreting or misusing prior information, or
       turning a correct intermediate state into an incorrect
       one.
4. Do not choose a later step merely because the failure
       becomes more visible there.
   If a later step only repeats, propagates, or amplifies an
       earlier mistake, select the earlier step where the
       mistake originated.
5. If a flawed strategy is repeated across multiple steps,
       attribute the failure to the step where that strategy
       was first introduced. If the strategy was reasonable
       but the execution result was wrong, attribute the
       failure to the first execution step that produced the
       incorrect result.
6. Use recoverability only as a tie-breaker: among otherwise
       similar candidates, prefer the earliest step whose
       error made recovery unlikely or blocked the trajectory
       from returning to a successful path.

You are given:
- the TASK,
- the FULL TRAJECTORY,
- a list of SUSPICIOUS STEPS that an upstream annotator
       flagged as containing some erroneous content (they are
       hints; you may pick any other step from the trajectory
       if you disagree).

=== WHAT TO OUTPUT ===
Output ONE JSON object with these keys and nothing else:
- "critical_step": <int>  - the step number of the critical
       error.
- "rationale": <string, 1-3 sentences> explaining why this
       step is critical.

=== REQUIRED OUTPUT FORMAT (single JSON object, no prose
       around it) ===
{
  "critical_step": <int>,
  "rationale": "<1-3 sentences>"
}

Return ONLY the JSON object.
```

E Experiment Details

All experiments were conducted with temperature set to 0 to ensure reproducibility.

E.1 Direct Prompting Baselines

To satisfy context-length constraints, we truncate each trajectory step to 2,048 characters and set the maximum context length to 16,384 tokens for each trajectory. Table 8 evaluates whether this per-step limit disadvantages direct prompting. Increasing the limit substantially lowers the proportion of truncated steps but does not improve accuracy, suggesting that retaining more raw detail can also introduce irrelevant content that interferes with critical-step judgment.

Table 8: Sensitivity of direct prompting to the per-step character limit.

Per-step limit	Avg. accuracy	Truncated steps
2,048 characters	**26.00**	27.1%
4,096 characters	20.02	14.5%
8,192 characters	24.73	3.5%

Direct Prompting Baseline — Critical Error Detection

```
You are an expert at analyzing failed agent trajectories and
       providing actionable debugging insights.

=== CRITICAL ERROR DEFINITION ===
A "critical error step" is the EARLIEST error instance that
       remains ACTIVE (unrepaired) at the terminal step of a
       failed trajectory -- the first error whose effects were
        never truly repaired, making it the root cause of the
       failure chain.

An error is "critical" if:
- It represents the ROOT CAUSE that made task success
       impossible
- It caused a cascade of subsequent errors
- The trajectory could have succeeded if THIS specific error
       had not occurred
- Correcting this specific error would fundamentally change
       the trajectory toward success

=== TASK ===
{task_description}

TRAJECTORY OUTCOME: FAILED (reward=0, reported by the task
       environment).
LAST STEP INDEX: {last_step}

=== FULL TRAJECTORY (one entry per message) ===
{trajectory_block}

Your task: find the critical error step of the trajectory.

=== REQUIRED OUTPUT FORMAT (single JSON object, no prose
       around it) ===
{
  "critical_error_analysis": {
    "step": <int>,
    "reason": "<1-3 sentences explaining why this step is a
       strong root-cause candidate>"
  }
}

Return ONLY the JSON object.
```

E.2 Multi-Agent Systems Baselines

We reproduced each multi-agent baseline using its official codebase and default configuration. To ensure a fair comparison with our implementation of TRAJDEBUG, we used Qwen3-235B-A22B-Thinking as the base model and fixed the temperature at 0 for reproducibility.

E.3 Additional Evaluation Results

Relaxed critical-step localization. Exact-step accuracy is deliberately strict, although predictions near the annotated critical step can still be useful in practice. For automated intervention, a prediction after the critical error may arrive too late; we therefore additionally evaluate whether the prediction falls in the asymmetric window $[\mathrm{GT}-3, \mathrm{GT}]$. As shown in Table 9, TRAJDEBUG reaches the best average accuracy of 50.2% and remains strongest on ALFWorld, τ^2-Bench, and SWE-Bench Pro. For human-in-the-loop debugging, where nearby later steps can also guide inspection, TRAJDEBUG reaches 72.8% under the symmetric $[\mathrm{GT}-5, \mathrm{GT}+5]$ window.

Inference cost and compute-matched comparison. Table 10 reports average token consumption per trajectory using Qwen3-235B-A22B-Thinking as the common backbone. To control for inference budget, we also run direct prompting 40 times at temperature 1 and use majority voting. This baseline consumes slightly more tokens than TRAJDEBUG but reaches only 29.56% accuracy, showing that TRAJDEBUG's gain over direct prompting cannot be explained by additional token consumption alone.

E.4 Application Experiment Details

Application — Vanilla Prompt

```
Trajectory:
{trajectory}

Your task:
1. Identify the earliest step which directly leads the agent
       off
   track or repeats ineffective behaviour.
2. Reference that exact step number as shown in the trajectory.

   Do not shift to later steps of that error.
3. Explain why the chosen step is wrong, citing relevant
   observation/action details.
4. Suggest a concrete alternative for that same step that
       would
   move the agent toward success
   (e.g., a specific action to take instead).

Respond strictly in the following format (single spaces around
colons, no extra text):
step:<number>
reason:<one concise, specific sentence>
suggestion:<one actionable suggestion for that step>
```

Application — Self-Reflection Prompt

```
Current result: {trajectory}

Why is this trajectory not finished the task?

Feedback:
```

Format-matched feedback comparison. The main application experiment compares complete debugging pipelines, whose feedback formats may differ. To isolate the effect of critical-step localization, we additionally use the same repair-hint generator and the same injection procedure for Vanilla Debug and TRAJDEBUG; the two conditions differ only in the critical step supplied to the hint generator. TRAJDEBUG's intermediate trigger, state, and attribution signals are hidden from both the

Method	ALFWorld	GAIA	WebShop	W&W-HC	W&W-Alg.	τ^2-Bench	SWE-Bench Pro	AVG
GLM-5.1	39.0	42.9	42.0	22.4	73.8	51.2	12.8	40.6
Gemini-3.1-Pro	38.0	34.7	40.0	29.3	62.7	65.5	23.3	41.9
DeepSeek-V4-Pro	35.0	63.3	40.0	29.3	66.7	56.8	17.4	44.1
Claude Sonnet 4.6	44.0	42.9	**48.0**	20.7	54.0	53.0	23.3	40.8
GPT-5.4	38.0	61.2	38.0	22.4	54.8	51.5	15.1	40.1
Claude Opus 4.6	36.0	44.9	**48.0**	32.8	69.0	53.5	19.8	43.4
Qwen3-235B-A22B	37.0	46.9	46.0	29.3	69.0	56.5	25.6	44.3
AgentDebugger	45.0	**64.0**	44.4	29.3	**72.2**	52.5	16.3	46.2
CHIEF	12.0	48.0	22.0	**34.5**	65.1	42.6	19.8	34.8
AgentRX	19.4	32.7	26.5	31.0	59.2	52.2	8.3	40.8
TRAJDEBUG (Ours)	**50.0**	59.2	40.0	29.3	69.0	**69.0**	**34.9**	**50.2**

Table 9: Critical-step localization accuracy (%) under the automated-intervention window $[\mathrm{GT}-3, \mathrm{GT}]$. AVG is the macro-average over the seven subsets.

Method	AVG Accuracy	Avg. Tokens
Direct Prompting	25.69	34,530 (1.0×)
+ Majority Voting	29.56	1,403,850 (40.6×)
AgentDebugger	23.72	1,398,333 (40.5×)
CHIEF	18.77	307,786 (8.9×)
AgentRX	23.10	218,862 (6.3×)
TRAJDEBUG (Ours)	**34.11**	1,382,243 (40.0×)

Table 10: Accuracy (%) and token consumption. Majority Voting is a compute-matched direct-prompting baseline.

hint generator and the actor. Table 11 shows that TRAJDEBUG remains stronger across all three settings, providing a controlled comparison of the downstream value of its localized critical step.

Method	Airline	Retail	SWE-Bench
Initial	78.00	84.21	72.00
Vanilla Debug	86.00 (+8.00)	90.35 (+6.14)	78.00 (+6.00)
TRAJDEBUG	**88.00 (+10.00)**	**92.98 (+8.77)**	**80.00 (+8.00)**

Table 11: Task success rate (%) in the format-matched per-trajectory repair experiment.

Iterative multi-round repair. We further extend the per-trajectory repair setting to multiple rounds under the same oracle access to rollout success. After each unsuccessful re-execution, we run TRAJDEBUG on the newly failed trajectory to generate another repair hint, retain all hints from previous rounds, and inject the accumulated hints into the actor's system prompt for the next attempt. Once a task succeeds, we stop iterating on that task; otherwise, we repeat this procedure for up to five rounds, yielding at most five cumulative hints. Table 12 reports the cumulative task success rate over the original evaluation set after each round.

The first repair round brings the largest improvement. Subsequent iterations can still resolve some persistent failures, but with diminishing returns.

Round	Airline	Retail	SWE-Bench Verified
0 (Initial)	78.00	84.21	72.00
1	90.00	95.61	81.00
2	92.00	96.49	81.00
3	92.00	97.37	81.00
4	92.00	97.37	81.00
5	92.00	98.25	83.00

Table 12: Cumulative task success rate (%) over five rounds of iterative per-trajectory repair with accumulated hints.

E.5 TRAJDEBUG Error Analysis

Candidate retention and conditional attribution. We first stratify all 869 evaluated trajectories according to whether the human-annotated critical error is retained in the candidate set passed to final attribution; one of the 50 GAIA trajectories is excluded because its ground-truth critical-error label is null. As shown in Table 13, TRAJDEBUG retains the ground-truth step in 42.0% of trajectories, compared with an expected 14.2% coverage from a size-matched random candidate set. When the ground truth is retained, candidate-guided attribution reaches 45.5% micro-average accuracy, substantially outperforming direct holistic prompting at 29.6%. When it is excluded upstream, TRAJDEBUG's evidence-constrained out-of-set fallback outperforms direct holistic prompting (39.1% versus 35.1%). The fallback must identify an evidence-backed missing trigger together with its violated reference, origin step, and terminal footprint. The overall 41.8% reported here is a trajectory-level micro-average; Table 1 reports a macro-average across the seven benchmark subsets.

Stage-wise failure decomposition. We decompose the failure cases of TRAJDEBUG according to the stage at which the ground-truth critical error is

GT status before attribution	Cases (%)	Random coverage	TRAJDEBUG	Direct holistic	Δ
Retained in candidate set	365 (42.0%)	14.2%	166 (45.5%)	108 (29.6%)	+15.9 pp
Excluded upstream	504 (58.0%)	85.8%	197 (39.1%)	177 (35.1%)	+4.0 pp
Overall	869	–	363 (41.8%)	285 (32.8%)	+9.0 pp

Table 13: Candidate-set coverage and conditional exact-step accuracy. Accuracy is micro-averaged within each stratum. Random coverage is the expected coverage of a uniformly sampled candidate set with the same size as TRAJDEBUG's candidate set.

Dataset	Trigger (%)	State (%)	Attribution (%)
ALFWorld	32.4	6.8	60.8
GAIA	20.0	6.7	73.3
WebShop	18.4	0.0	81.6
WhoAndWhen-HC	33.3	2.2	64.4
WhoAndWhen-Algo	62.1	12.1	25.9
SWE-Bench Pro	45.2	24.2	30.6
τ^2-Bench	49.2	30.7	20.1
Overall	**42.1**	**17.7**	**40.1**

Table 14: Failure decomposition of TRAJDEBUG.

lost: **Trigger Miss**, where the critical error is not identified as an error trigger; **State Miss**, where the critical error is detected but filtered out from $\mathcal{F}(\tau)$ during error state classification; and **Attribution Miss**, where the correct candidate enters the final attribution stage but is not selected. Table 14 shows that Trigger Misses (42.1%) and Attribution Misses (40.1%) are the two dominant failure modes, while State Misses are less frequent (17.7%).

This decomposition characterizes where the remaining failures occur. First, Trigger Misses (42.1%) account for the largest share of failures. Although our trigger detector inspects each step under multiple compressed trajectory views and identifies evidence-grounded conflicts, some errors or deviations may be subtle, which can lead the detector to miss the critical error. Second, the relatively low share of State Misses (17.7%) suggests that error state classification usually preserves the critical error once it has been detected as a trigger. Attribution Misses (40.1%) therefore correspond to harder residual cases after candidate filtering, where multiple retained errors are plausibly related to the final failure and the model must distinguish the earliest failure-responsible step. These patterns suggest that further gains may come from improving recall for subtle trigger errors and refining attribution among already filtered, terminal-relevant candidates.