Seeing the Whole Elephant: A Benchmark for Failure Attribution in LLM-based Multi-Agent Systems

Anonymous ACL submission

Abstract

Failure attribution, i.e., identifying the responsible agent and decisive step of a failure, is particularly challenging in LLM-based multi-agent systems (MAS) due to their natural-language reasoning, nondeterministic outputs, and intricate interaction dynamics. A reliable benchmark is therefore essential to guide and evaluate attribution techniques. Yet existing benchmarks rely on partially observable traces that capture only agent outputs, omitting the inputs and context that developers actually use when debugging. We argue that failure attribution should be studied under full execution observability, aligning with real-world developer-facing scenarios where complete traces, rather than only outputs, are accessible for diagnosis. To this end, we introduce TraceElephant, a benchmark designed for failure attribution with full execution traces and reproducible environments. We then systematically evaluate failure attribution techniques across various configurations. Specifically, full traces improve attribution accuracy by up to 76% over a partial-observation counterpart, confirming that missing inputs obscure many failure causes. TraceElephant provides a foundation for follow-up failure attribution research, promoting evaluation practices that reflect real-world debugging and supporting the development of more transparent MASs.

1 Introduction

If there is one constant in the evolution of software, it is the persistent occurrence of failures (Charette, 2005). When it does, the critical first step is to assign responsibility for the failure to a specific component, i.e., failure attribution or localization, which enables developers to focus debugging efforts, guide the design of patches or architectural improvements. Traditional techniques for this task, ranging from statistical debugging (Zheng et al., 2006) and delta debugging (Misherghi and Su, 2006) to more recent learning-based approaches (Wong et al., 2016; Zou et al., 2019), operate under the assumption that system states are discrete, executions are traceable, and component behaviors are largely deterministic.

The rise of LLM-based multi-agent systems (MASs) fundamentally challenges these assumptions and complicates the attribution problem. In these systems, complex tasks are decomposed and coordinated across multiple agents whose primary reasoning and communication medium is natural language (Guo et al., 2024; Li et al., 2024). This introduces two layers of complexity for fault attribution. First, the problem-solving process in MASs often involves complex interactions among multiple LLM-powered agents, between agents and external tools, and within the internal reasoning processes of the LLMs themselves. These interactions complicate system logs, challenging the interpretation of system behavior and hindering rapid root cause identification. Second, the system actions and their resulting states are recorded in natural language within the log. The inherent ambiguity of natural language further impedes the precise characterization of operations and states.

To address the challenge of fault attribution in MAS, several technical approaches have been proposed, such as ECHO (Banerjee et al., 2025), AgenTracer (Zhang et al., 2025a), GraphTracer (Zhang et al., 2025b), and FAMAS (Ge et al., 2025). On another front, to effectively evaluate how these techniques perform in real-world MAS settings, the foundation lies in having benchmarks that accurately reflect authentic fault attribution scenarios.

To the best of our knowledge, Who&When (Zhang et al., 2025c) is currently the only benchmark specifically designed for failure attribution in LLM-based MAS. It provides partially observable execution traces that include only the agents' outputs, without the corresponding inputs such as original task instructions, prompts, or contextual

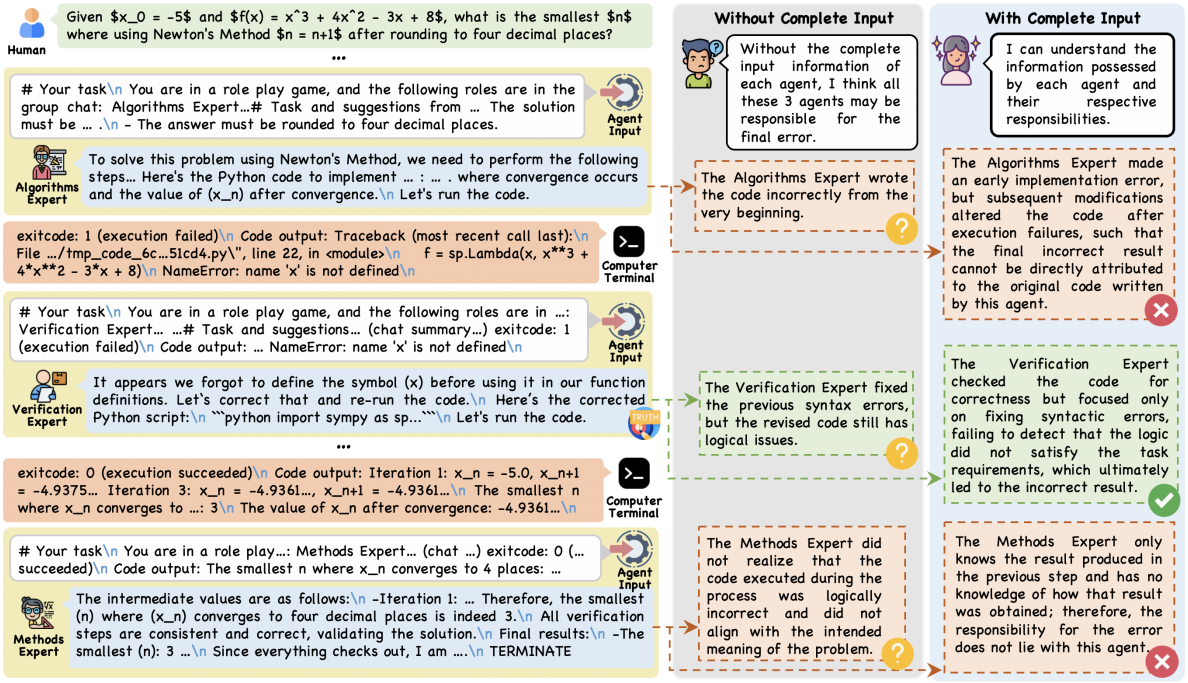

Figure 1: A failure case (from Who&When benchmark) illustrating the limitation of partial observability. When only agent outputs are visible and critical inputs are absent, localizing the decisive failure step becomes difficult.

messages. This restricted setting can be suitable for evaluating attribution methods under certain black-box scenarios, where internal inputs or intermediate states are inaccessible (e.g., debugging deployed agents on external platforms).

However, in many practical developer-facing debugging scenarios, developers typically have access to substantially richer execution information, including task instructions, prompts, intermediate messages, tool invocations, and environment states. More importantly, we view failure attribution as a developer-centric task conducted in the context of system debugging and iterative refinement, with the explicit goal of producing actionable insights for repairing and improving MASs. Achieving this goal critically depends on access to the inputs, intermediate states, and environment interactions that drive agent behavior. In contrast, black-box settings, where only outputs are observable, leave many failures ambiguous and thus offer limited practical value for guiding system fixes. For instance, our analysis of the 184 failure cases in Who&When benchmark shows that in at least 21% of instances, developers cannot reliably perform failure attribution using output-only logs. We illustrate this in Figure 1 with an adapted case from Who&When, where we restored the missing inputs by re-running the original system. Under partial observability, the failure cause remains ambiguous, whereas with full inputs, the ambiguity is largely resolved, enabling precise step-level localization.

To fill this gap, we construct TraceElephant, the first benchmark specifically designed for developer-facing failure attribution in LLM-based MASs by "seeing the whole elephant", i.e., providing access to the complete execution narrative of an MAS. Compared with existing benchmarks in this field (i.e., Who&When), TraceElephant differs in two key aspects: (1) TraceElephant collects step-by-step execution traces from multiple representative MASs, where each trace records agent-level actions, natural language inputs and outputs, tool and environment interactions, agent configurations and system architecture, and (2) it accompanies each trace with a reproducible execution environment, enabling controlled re-execution, state inspection, and interactive hypothetical debugging queries (e.g., "what if this agent had received different input?").

Building upon this benchmark, we evaluate the automated failure attribution techniques under various configurations. The experimental results demonstrate that with complete trace information, it achieves an average attribution accuracy of 65.9% at the agent level and 30.3% at the step level. This represents an improvement of 22% in agent-level accuracy and 76% in step-level accuracy over the performance on output-only traces (i.e., similar to

Who&When benchmark), underscoring the critical role of full observability. With the running environment, the step-level accuracy can further improve by 10%. Our analysis further reveals that step-level attribution is more sensitive to missing information than agent-level attribution, highlighting the finer-grained, context-dependent nature of localizing the precise failure step. Moreover, performance can also vary across different MAS architectures, agent types, and step positions, indicating the need for architecture-aware attribution approaches. We also provide actionable implications and takeaways for developing more effective attribution techniques and designing more debuggable MASs.

On one hand, this work highlights the necessity of full trace information for reliable fault attribution, indicating that developers should, whenever possible, incorporate all accessible execution details when performing this task. On the other hand, it also suggests that the field would benefit from more diverse benchmarks to evaluate attribution techniques from multiple perspectives, thereby contributing to a more solid and cumulative understanding of failure attribution in MAS.

The contributions of this work are as follows.

- We are the first to study failure attribution of LLM-based MASs from a developer-facing scene, where full execution traces and reproducible execution environment are available.
- We develop TraceElephant[1], a failure attribution benchmark that instantiates the above paradigm by providing a collection of annotated execution traces. It consists of 220 failure traces collected from three representative MASs, with each annotated with the responsible agent and decisive failure step.
- We conduct extensive experiments on TraceElephant, examining how failure attribution behaves under various configurations, and providing related implications.

2 Problem Definition

We consider an LLM-based multi-agent system (MAS) composed of a finite set of agents $\mathcal{A} = \{a_1, \ldots, a_N\}$ collaboratively performing a task τ. The system executes in discrete steps under a turn-based protocol, where exactly one agent is selected to act at each step. At step t, the acting agent $a(t)$ receives input x_t and produces output y_t. The execution trace at step t is recorded as

$$o_t = \big(x_t,\, y_t,\, a(t),\, \textit{step\_id}_t,\, \textit{agent\_id}_t\big).$$

The task outcome is determined by the complete execution trace. In case of failure, we aim to attribute responsibility at both the step and agent level: step-level attribution identifies the earliest point at which failure becomes inevitable, while agent-level attribution identifies the agent responsible for the failure at that step. For the task of failure attribution in MAS, the goal is to determine both the step-level and agent-level responsibility given the full failure trace.

For a full formalization, including definitions, notation, and equations, refer to Appendix A.1.

3 Benchmark Construction

We construct TraceElephant, a benchmark for failure attribution in LLM-based MAS. Each instance is grounded in an executable MAS and is associated with a fully observable execution trace, as well as the annotated failure-responsible agent and decisive failure step.

3.1 Data Sources: Systems and Tasks

TraceElephant collects execution traces from three representative MASs across various tasks, aligning task design with each system's intended capabilities, as demonstrated in Table 1. Detailed descriptions of the systems, task sources, run configurations are provided in Appendix A.2.

System	Task Source	# Traces	# Failed
Captain-Agent	GAIA	126	73
Captain-Agent	AssistantBench	21	12
Magentic-One	GAIA	119	74
Magentic-One	AssistantBench	30	17
SWE-Agent	SWE-Bench	84	44
Total	All	380	220

Table 1: Overview of execution traces in TraceElephant.

3.2 Trace Collection Pipeline

TraceElephant adopts an automated collection pipeline designed to capture complete execution traces while preserving the original structure of agent interactions. At its core, a lightweight LLM API middleware transparently intercepts and captures all LLM requests, responses, and subsequent tool interactions without modifying the original

[1] https://github.com/TraceElephant/TraceElephant

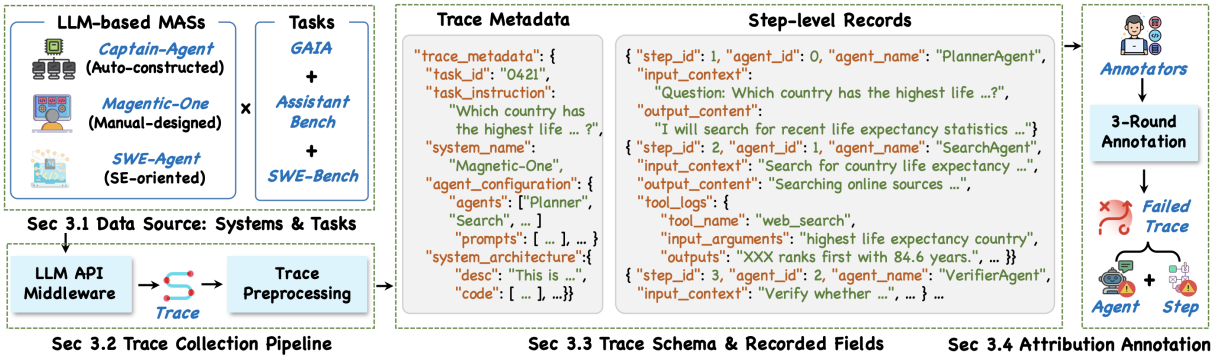

Figure 2: Overview of TraceElephant.

agent implementations. The raw traces then undergo targeted pre-processing, extracting only basic attributes such as agent names and step types, to maintain maximal fidelity to the original execution flow. More details are provided in Appendix A.3.

3.3 Trace Schema and Recorded Fields

Each instance in TraceElephant benchmark consists of a complete execution trace generated by running a multi-agent system on a given task, as well as the executable system associated with the runnable code and configurations for this specific trace instance.

(1) Trace metadata. Each execution trace is associated with a set of trace-level metadata that describes the execution context as a whole. These metadata fields include: (1) *task_id* and *task_instruction*, indicating the task being solved; (2) *system_name*, identifying the multi-agent system that produced the trace; (3) *agent_configuration*, the runtime setup defining the agent roster, prompts, and toolset; and (4) *system_architecture*, the design documentation and implementation code defining the system's structural blueprint.

(2) Step-level records. An execution trace is composed of an ordered sequence of steps, where each step corresponds to a single agent action. For each step, TraceElephant records a set of observable fields, organized into input and output fields, that are directly exposed by the running system.

(i) Step input fields. They capture all information provided to an agent when an action is executed. Specifically, the recorded input fields include: (1) *step_id*, a unique identifier indicating the global execution order of the step; (2) *agent_id* and *agent_name*, identifying the agent responsible for the action; (3) *input_context*, which stores the complete input information supplied to the agent. This field typically includes the original task instruction, intermediate messages exchanged among agents, and any system-constructed contextual information available at execution time.

(ii) Step output fields. They record the observable results produced by the agent at the current step. These fields include: (1) *output_content*, which stores the response generated by the agent. (2) *tool_logs*, which (when available) records raw interaction logs with external tools or environments when such interactions occur. Each tool log typically includes the tool name, input arguments, outputs, and execution status.

All execution traces are stored in a structured JSON format. Each trace is represented as a JSON object containing trace-level metadata and an ordered list of step records, within which input and output fields are stored as nested objects, preserving all content in its raw, unprocessed form. We provide an example trace in Appendix A.4.

3.4 Failure Attribution Annotation

The annotation aims at acquiring failure attribution labels, i.e., (1) the agent primarily responsible for the failure, and (2) the execution step where the failure originates. These labels are obtained through a multi-round expert annotation process designed to ensure reliability. Detailed annotation protocols are provided in Appendix A.5.

4 Failure Attribution Evaluation

4.1 Evaluation Design

Observability Configurations. We evaluate failure attribution performance under two complementary observability configurations reflecting practical debugging workflows. (1) ***Static***: Attribution is performed using the complete execution trace (including metadata, inputs, and outputs fields as shown in Section 3.3). (2) ***Dynamic***: In addition to the static trace, a replayable execution environment

System	Ground Truth	Static Configurations								Dynamic Config.	
		All-at-Once		Binary Search		Step-by-Step		Static Agentic		Dynamic Agentic	
		Agent	Step	Agent	Step	Agent	Step	Agent	Step	Agent	Step
Captain-Agent	**w/** Ground Truth	64.7	29.4	25.9	14.1	57.7	22.4	67.1	30.6	**68.2**	**32.9**
	w/o Ground Truth	**63.5**	22.4	24.7	9.4	44.7	17.7	58.8	24.7	61.2	**25.9**
Magentic-One	**w/** Ground Truth	58.2	25.2	38.5	13.2	63.7	20.9	67.0	30.8	**68.1**	**33.0**
	w/o Ground Truth	56.0	23.1	37.4	8.8	60.4	15.4	61.5	26.4	**61.5**	**27.5**
SWE-Agent	**w/** Ground Truth	**63.6**	29.6	52.3	11.4	61.4	6.8	**63.6**	29.6	**63.6**	**34.1**
	w/o Ground Truth	54.6	22.7	50.0	9.1	56.8	4.6	56.8	27.3	**59.1**	**29.6**
All-avg	**w/** Ground Truth	62.2	28.1	38.9	12.9	60.9	16.7	65.9	30.3	**66.7**	**33.3**
	w/o Ground Truth	58.0	22.7	37.4	9.1	54.0	12.5	59.1	26.1	**60.6**	**27.6**

Table 2: Performance (i.e., Agent-level accuracy and Step-level accuracy) comparison of failure attribution techniques across different agent systems under *with or without ground truth* scenario. We use bold to indicate the highest value, while underline indicating the second highest value.

Observability Configurations	Agent	Step
All-at-Once	**0.62**	**0.28**
w/o metadata	0.55	0.21
w/o input	0.54	0.18
w/o metadata & input	0.51	0.16
Static Agentic	**0.66**	**0.30**
w/o metadata	0.57	0.23
w/o input	0.56	0.19
w/o metadata & input	0.54	0.17

Table 3: Ablation study on observability configurations.

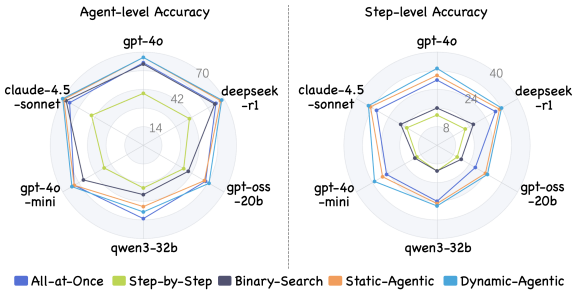

Figure 3: Comparison under different backbone LLMs.

is provided, enabling controlled re-execution and counterfactual probing to verify or refine candidate attributions. We additionally utilize several variants of static configuration to further evaluate the performance under different levels of observability.

Attribution techniques. We utilize five commonly-used techniques. There are three LLM-based prompting techniques, i.e., ***All-at-Once, Binary Search, Step-by-Step***, which differ in how the trace is provided to the LLM. There are two agent-based techniques. ***Static Agentic*** adopts mini-SWE-agent[2], which can navigate the trace information, retrieve related fields as needed, and make a conclusion gradually. ***Dynamic Agentic*** first proposes candidate failure attributions derived based on static agentic technique, with the candidate steps and agents. The method then re-runs the system from the corresponding execution point and issues counterfactual checks. Default setting of the following evaluations is Static Agentic.

Application Scenarios. Following existing studies (Zhang et al., 2025c), experiments are conducted both ***With ground truth*** (use this, if not explicitly stated) and ***Without ground truth*** scenarios, simulating different debugging contexts.

Evaluation Metrics. Predictions are evaluated for both responsible agent and decisive step, and we use ***agent-level accuracy*** and ***step-level accuracy***, following existing studies (Zhang et al., 2025c,b).

More details about the evaluation design are provided in Appendix A.6.

4.2 Results and Analysis

4.2.1 Performance Across Configurations

(1) Static vs. Dynamic Attribution Performance. Generally speaking, dynamic configuration can achieve the highest performance, especially for step-level accuracy, and within static configurations, the agentic technique is the best in most experimental settings.

As demonstrated in Table 2, with ground truth, failure attribution performance can reach an average of 33.3% and 30.3% step-level accuracy respectively in dynamic and static configuration. For agent-level accuracy, these figures are 66.7% and 65.9% respectively. The dynamic method improves step-level attribution by 10% due to actively verifying candidate failure steps through controlled

[2] 2.4k stars as of January 2026. https://github.com/SWE-agent/mini-swe-agent/.

re-execution and counterfactual probing. This process helps filter out spurious candidates identified from static traces, thereby refining the attribution. Agent-level accuracy sees limited gains because it depends more on understanding agent roles and coordination logic, i.e., information already largely available in complete static traces.

Among all static configurations, agentic technique achieves the highest performance in almost all cases, which is likely because it allows to better trace responsibility through the chain of analysis and tool use that characterizes MAS failures. Furthermore, among other static configuration, All-at-Once performs relatively better than others. We conduct further analysis in Section 4.2.3.

(2) Ablation Study. Table 3 presents the ablation results for two representative techniques, while others show similar trend and are omitted for brevity. Full observability is essential for accurate failure attribution, and the absence of either metadata or input fields leads to noticeable performance degradation. Besides, step-level attribution is more sensitive to missing information than agent-level attribution, i.e., 76% vs. 22% accuracy drop, underscoring the finer-grained and more context-dependent nature of identifying the failure step.

(3) Effect of Backbone LLMs. Figure 3 demonstrates the performance in terms of different backbone LLMs. Broadly speaking, Claude-4.5-Sonnet, DeepSeek-R1, and GPT-4o exhibit relatively strong performance in both agent-level and step-level accuracy, likely due to their advanced reasoning architectures and strong contextual understanding. In contrast, Qwen3-32B and GPT-OSS-20B show weaker performance, which may be attributed to their smaller parameter scales and consequently limited capacity for sustained multi-step reasoning and fine-grained causal tracing.

(4) With vs. Without Ground Truth. Performance consistently declines across all systems and techniques when ground truth is unavailable. This confirms that access to a reference outcome (e.g., a correct answer or test pass/fail signal) provides crucial guidance for attribution, especially for finer-grained step attribution. Agentic methods (especially Dynamic Agentic) show relatively smaller performance degradation without ground truth. This suggests their interactive validation (replay and counterfactual checks) can partially compensate for the missing reference signal by testing behavioral hypotheses.

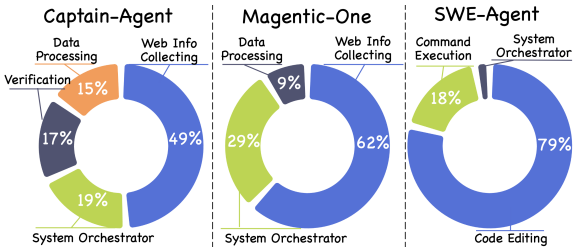

Figure 4: Distribution of failure agent in TraceElephant.

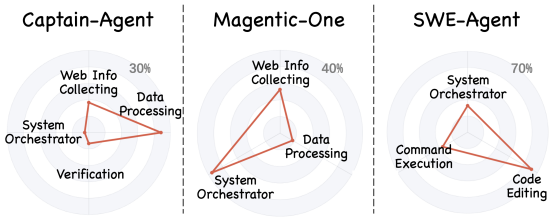

Figure 5: Fine-grained agent-level accuracy.

4.2.2 Failure & Attribution Patterns

(1) Failure Responsible Agents. Figure 4 illustrates the distribution of failure-prone agent types in our benchmark. Agents responsible for interacting with external environments or performing concrete operations are most prone to errors (almost over 50% of the failures), i.e., agents handling web information collection and browsing in CaptainAgent and Magentic-One, agent directly editing code in SWE-Agent. This might be because these actions depend on dynamic, often noisy external systems (APIs, websites, file systems), where malformed requests, parsing errors, or unexpected outputs can easily occur. The orchestrator/planner agents also represent a non-negligible (18-29%) source of failures. Their mistakes often stem from incorrect task decomposition, sub-optimal agent selection, or flawed coordination logic-errors that may not manifest immediately but can propagate and amplify throughout the execution.

(2) Agent-level Accuracy. Figure 5 further breaks down the predicted agent-level accuracy for these key agent types. For Captain-Agent, data processing agents show high prediction accuracy (53%), likely because their errors (e.g., CSV parsing or script execution failures) leave clear, tool-mediated traces, and web-related agents achieve moderate accuracy (22%), as search API outputs are often ambiguous in natural language. For Magentic-One, the Orchestrator attains higher accuracy (38%) than in Captain-Agent, possibly because its failures frequently arise during visible mid-process interventions, such as re-planning after a browser search stalls, making its responsibility more discernible from output logs.

(3) Decisive Failure Steps. Figure 6 shows the distribution of failure steps over the execution timeline in our benchmark. Failures in the automatically generated system (i.e., Captain-Agent) are dispersed. Since the system constructs and coordinates agent teams on-the-fly for each task, errors can be introduced at multiple points: during agent selection, iterative planning, inter-agent coordination, or tool-calling. Failures in manually crafted systems (i.e., Magentic-One and SWE-Agent) are heavily concentrated in the early steps, tightly linked to the initial task planning and routing decisions made by their central orchestrators.

(4) Step-level Accuracy. Figure 7 demonstrates the step-level accuracy for the ground-truth failure steps divided equally into three segments based on their chronological order. In Captain-Agent and SWE-Agent, step-level accuracy remains relatively stable across all phases, while for Magentic-One, the accuracy is remarkably low in the early phase (8%), moderate in the middle (19%), and high in the late phase (52%). This suggests that failures occurring early in the execution are especially difficult to attribute, whereas those in later stages are more easily identified. A plausible explanation lies in Magentic-One's system design, which often involves extended exploratory and re-planning cycles. Early-phase errors, such as incorrect task decomposition, improper subtask assignment, or flawed initial assumptions, may not manifest immediately and only become visible after subsequent execution fails. In contrast, mid- to late-phase failures often involve tangible inconsistencies, evidence conflicts, or clear execution failures (e.g., mismatched web information), which provide stronger, more localized signals for attribution.

4.2.3 TraceElephant vs. Who&When

Performance under the Output-only Setting. Table 3 demonstrates the performance under *All-at-Once w/o metadata&input*, i.e., only utilizing the step output fields for failure attribution, which is exactly the case as Who&When benchmark (Zhang et al., 2025c). We can see that, solely relying on the output fields, the performance suffers a notable degradation, from 62% to 51% in agent-level accuracy and from 28% to 16% in step-level accuracy. These results are largely consistent with those reported in Who&When (with the same backbone LLM), i.e., agent-level accuracy being 51.1% to 54.3% and step-level accuracy being 12.5% to 13.5%.

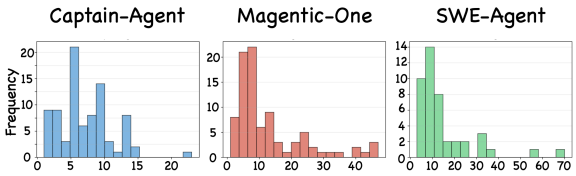

Figure 6: Distribution of failure step in TraceElephant.

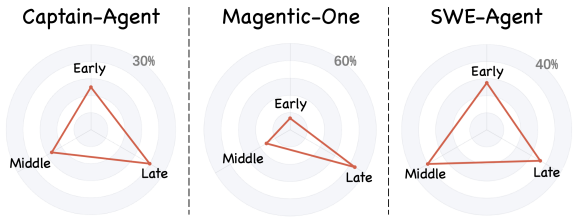

Figure 7: Fine-grained step-level accuracy.

Variation in Prompting Strategies. A key difference lies in the comparison across prompting strategies: in our experiments, both agent-level and step-level accuracy achieve the best results under the All-at-Once setting, whereas in Who&When, step-level accuracy peaks under the Step-by-Step setting. This discrepancy may stem from the fact that our execution traces contain more steps (average LLM invocations: 20.5 for Captain-Agent, 29.3 for Magentic-One vs. 9.6 and 28.8 in Who&When), leading to overly long contextual inputs in later interactions of incremental prompts, i.e., exceeding the effective context window of the LLM and thus degrading performance.

We provide more analysis on Appendix A.7.

4.2.4 Implications and Takeaways

Our benchmark and experimental findings offer several actionable insights for the future of failure attribution in MAS.

Architecture-Aware Attribution. As revealed in Section 4.2.2, the types of failure responsible agents and the distribution of decisive failure steps vary significantly across different MAS architectures. The performance of automated attribution itself is highly dependent on agent types and step positions. In practice, this calls for architecture-aware attribution methods that leverage prior knowledge about an MAS's design (e.g., centralized vs. dynamic team formation, tool-heavy vs. planning-heavy workflows) to focus attention on its most vulnerable components and interaction patterns, thereby improving both the efficiency and accuracy of failure attribution.

Enhancing Static Attribution with Advanced Reasoning. Our *Static Agentic* method, while

effective, currently employs only basic tool use (e.g., inspecting step I/O, see details in Appendix A.6). Future work can integrate more sophisticated reasoning strategies, such as explicit hypothesis generation and testing cycles, or graph-based reasoning over extracted agent interaction networks. This could help the agent better synthesize long-horizon dependencies and ambiguous failure patterns present in static traces.

Leveraging Dynamic Environment for Deeper Analysis. The current dynamic configuration primarily performs single-step replay and counterfactual checks, which already improve step-level accuracy. The provided executable environment enables more advanced analysis techniques, such as reading code to infer potentially faulty nodes in the system design, reconstructing the actual control-flow structure from execution traces, conducting systematic state-space exploration around failure points, testing system robustness through automated fault injection, or applying causal discovery algorithms that actively intervene on multiple variables to isolate root causes. These measures allow researchers to better understand how error traces arise. As a result, the benchmark is transformed from a passive dataset into an active laboratory for debugging research.

Specializing Models for Attribution via Fine-Tuning. The strong correlation between model reasoning capability (e.g., Claude-4.5, DeepSeek-R1) and attribution accuracy also highlights the need for specialized models. Promisingly, recent work such as GraphTracer (Zhang et al., 2025b) and AgenTracer (Zhang et al., 2025a) shows that task-specific fine-tuning can enable smaller models to surpass larger base models. Future efforts can fine-tune compact models utilizing TraceElephant's traces and running environments, explicitly incorporating structural features (e.g., agent graphs, tool-call sequences) and temporal dependencies as auxiliary training signals to create efficient, attribution-specialized models.

Toward Integrated Debugging Tools. Our findings underscore the practical value of full observability. This motivates the development of integrated debugging tools that automatically capture rich execution traces, visualize agent interaction flows, and suggest potential failure points using attribution models. Embedding such capabilities into MAS development frameworks can significantly reduce debugging overhead.

5 Related Work

LLM-based Multi-Agent Systems. LLM-based Multi-Agent Systems (MASs) have been widely studied as a paradigm for solving complex tasks, where agents interact through natural language, planning modules, and tool usage to perform tasks such as software engineering, question answering, and decision making (Chen et al., 2024; Maldonado et al., 2024; Zhang et al., 2025d). Existing work has explored different system architectures, including centralized planning, decentralized coordination, and hybrid designs, as well as mechanisms for role assignment and communication (Wu et al., 2024; Li et al., 2023; Fourney et al., 2024; Song et al., 2025; Team, 2025; Jin et al., 2025). As MAS are deployed in increasingly complex and long-horizon tasks, failures become more difficult to diagnose and failure attribution is even challenging.

Failure Attribution in MAS. There are several techniques for failure attribution in MASs. For example, FAMAS (Ge et al., 2025) conducted spectrum analysis on multiple execution trajectories collected by repeatedly replaying a failed task. AgentTracer (Zhang et al., 2025a) proposed a lightweight model trained via multi-granular reinforcement learning to jointly optimize step-level and agent-level attribution accuracy. GraphTracer (Zhang et al., 2025b) modeled agent interactions as information dependency graphs, and traced causal information flows. ECHO (Banerjee et al., 2025) combined hierarchical context representation, objective analysis-based evaluation, and consensus voting to improve failure attribution accuracy. To reliably evaluate these existing techniques and support the development of new ones, high-quality benchmarks like ours are essential.

6 Conclusion

The inherent complexity and non-deterministic interactions in LLM-based MASs make failure attribution a core challenge for ensuring operational reliability and facilitating targeted debugging. To support this, we introduce TraceElephant, a benchmark for failure attribution of LLM-based MASs under full execution observability. Experiments show that full observability significantly improves attribution performance, and dynamic replay further enhances attribution capability. This work lays a practical foundation for future failure attribution research and promotes evaluation practices that mirror real-world debugging.

7 Limitations

This work focuses on failure attribution under developer-facing settings using execution traces collected from a limited set of representative multi-agent systems. While this scope does not cover all possible system architectures or black-box usage scenarios, it reflects realistic debugging conditions in which full execution traces are available. As this study is conducted on only three MASs, some of our findings may not generalize to all existing or future systems. However, the selected systems, i.e., Captain-Agent, Magentic-One, and SWE-Agent, are intentionally diverse in their design paradigms, covering dynamic team assembly, centralized orchestration, and specialized software engineering workflows. This deliberate diversity enhances the representativeness of our benchmark and mitigates the risk of architecture-specific bias.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Anthropic. 2025. Introducing Claude Sonnet 4.5. 2025-01-05.

Adi Banerjee, Anirudh Nair, and Tarik Borogovac. 2025. Where did it all go wrong? a hierarchical look into multi-agent error attribution. *arXiv preprint arXiv:2510.04886*.

Robert N Charette. 2005. Why software fails [software failure]. *IEEE spectrum*, 42(9):42–49.

Shuaihang Chen, Yuanxing Liu, Wei Han, Weinan Zhang, and Ting Liu. 2024. A survey on LLM-based multi-agent system: Recent advances and new frontiers in application. *arXiv preprint arXiv:2412.17481*.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang, Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. 2024. Magentic-One: A generalist multi-agent system for solving complex tasks. *Preprint*, arXiv:2411.04468.

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. *Preprint*, arXiv:2509.13782.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. DeepSeek-R1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Sheng Jin, Haoming Wang, Zhiqi Gao, Yongbo Yang, Bao Chunjia, and Chengliang Wang. 2025. Evolution in simulation: Ai-agent school with dual memory for high-fidelity educational dynamics. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5843–5857.

Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9.

Diego Maldonado, Edison Cruz, Jackeline Abad Torres, Patricio J Cruz, and Silvana del Pilar Gamboa Benitez. 2024. Multi-agent systems: A survey about its components, framework and workflow. *IEEE Access*, 12:80950–80975.

Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151.

OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. 2025-01-05.

Linxin Song, Jiale Liu, Jieyu Zhang, Shaokun Zhang, Ao Luo, Shijian Wang, Qingyun Wu, and Chi Wang. 2025. Adaptive in-conversation team building for language model agents. *Preprint*, arXiv:2405.19425.

MiroMind AI Team. 2025. MiroFlow: A high-performance open-source research agent framework. https://github.com/MiroMindAI/MiroFlow.

W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. AutoGen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-Agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. 2025a. AgenTracer: Who is inducing failure in the llm agentic systems? *arXiv preprint arXiv:2509.03312*.

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

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and 1 others. 2025c. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. *ICML 2025 Spotlight*.

Xueqiang Zhang, Xiaofei Dong, Yiru Wang, Dan Zhang, and Feng Cao. 2025d. A survey of multi-ai agent collaboration: Theories, technologies and applications. In *Proceedings of the 2nd Guangdong-Hong Kong-Macao Greater Bay Area International Conference on Digital Economy and Artificial Intelligence*, pages 1875–1881.

Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112.

Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347.

A Appendix

A.1 Detailed Problem Definition

Multi-Agent Execution and Observability. We consider an LLM-based multi-agent system (MAS) $\mathcal{M}$ composed of a finite set of agents $\mathcal{A} = \{a_1, a_2, \ldots, a_N\}$ that collaboratively perform a task τ. The system executes in discrete steps. At each step, a single agent is selected to act, possibly through a dispatcher, router, or planning component that is part of the system. At step t, the acting agent is selected as $a(t) = g(h_{t-1})$, where $g(\cdot)$ represents the system-level agent selection mechanism and h_{t-1} denotes the system context at step $t-1$, summarizing the execution history. The selected agent receives an input $x_t = \phi(h_{t-1})$, and produces an output $y_t = \pi_{a(t)}(x_t)$, where $\pi_{a(t)}$ denotes the policy of agent $a(t)$. The system context is then updated as $h_t = u(h_{t-1}, a(t), x_t, y_t)$. The execution terminates after T steps, producing an execution trace $\mathcal{T} = \langle s_1, s_2, \ldots, s_T \rangle$. Each execution step s_t is associated with a structured observable record $o_t = \big(x_t,\ y_t,\ a(t),\ \textit{step\_id}_t,\ \textit{agent\_id}_t\ \big)$.

Task Outcome and Failure Attribution. Given a completed execution trace $\mathcal{T}$, the system produces a task outcome. We define a task outcome function $\Omega(\mathcal{T}) \in \{0, 1\}$, where $\Omega(\mathcal{T}) = 1$ indicates successful task completion and $\Omega(\mathcal{T}) = 0$ indicates failure. Failure attribution aims to identify the agent and the execution step responsible for a task failure based on observable execution traces. We define failure attribution at two levels. **Step-level attribution.** Let $\mathcal{T}_{\le t}$ denote the prefix of the execution trace up to step t. We define the set of feasible continuations from $\mathcal{T}_{\le t}$ under system $\mathcal{M}$ as $\mathcal{C}(\mathcal{T}_{\le t}) = \{\mathcal{T}' \mid \mathcal{T}' \text{ is a valid continuation from } \mathcal{T}_{\le t}\}$. Failure is said to become inevitable at step t if all feasible continuations result in failure: $\forall \mathcal{T}' \in \mathcal{C}(\mathcal{T}_{\le t}),\ \Omega(\mathcal{T}_{\le t} \oplus \mathcal{T}') = 0$. The failure step is defined as the earliest inevitable step $t^* = \min \Big\{ t \in \{1, \ldots, T\} \mid \forall \mathcal{T}' \in \mathcal{C}(\mathcal{T}_{\le t}),\ \Omega(\mathcal{T}_{\le t} \oplus \mathcal{T}') = 0 \Big\}$. **Agent-level attribution.** Given an execution trace $\mathcal{T}$ and its failure step t^*, agent-level attribution identifies the agent whose action at the failure step gives rise to the task failure. Formally, the failure-responsible agent is defined as $a^* = \arg\max_{a \in \mathcal{A}} \mathbb{I}[\exists\, s_{t^*} \in \mathcal{T} \text{ s.t. } a(t^*) = a]$, where $a(t^*)$ denotes the agent selected by the system to act at step t^* in the execution trace $\mathcal{T}$.

A.2 Details about Data Sources

LLM-based Multi-Agent Systems. We collect execution traces from three representative MASs:

Captain-Agent, *Magentic-One*, and *SWE-Agent*. Captain-Agent (Song et al., 2025) follows an automated system construction paradigm, where a meta-controller dynamically assembles and coordinates a team of agents for each task through iterative planning and execution. In contrast, Magentic-One (Fourney et al., 2024) adopts a manually specified architecture, in which an orchestrator governs a fixed set of specialized agents. SWE-Agent (Yang et al., 2024) specializes in software engineering tasks, where agents interact with code repositories and development environments to navigate, understand, and edit codebases. For each system, we use the official implementation and retain the original agent definitions, prompts, and tool interfaces.

Task Sources. For both *Captain-Agent* and *Magentic-One*, execution traces are collected on tasks from *GAIA* and *AssistantBench*, reflecting general-purpose agentic problem solving involving multi-step reasoning, information gathering, and tool use. *GAIA* tasks typically require agents to decompose a question, retrieve information from multiple sources, and integrate intermediate results through a sequence of reasoning steps. *AssistantBench* tasks further emphasize interactive tool use and sequential decision making, where errors may arise from incorrect tool selection, incomplete information propagation, or early planning mistakes. For *SWE-Agent*, traces are collected on *SWE-Bench Verified* tasks, which focus on software engineering scenarios such as code navigation, modification, and validation. These tasks often involve long execution traces with repeated interactions between language model agents and external development environments. Failures in this setting commonly manifest as incorrect localization of code changes, incomplete fixes, or mismatches between generated patches and test requirements, resulting in qualitatively different failure patterns compared to general-purpose reasoning tasks.

Run Configuration. For each system-task pair, we execute the system under a fixed configuration θ and assign a unique run identifier ρ. To ensure stable tool invocation behavior while retaining a controlled degree of stochastic diversity, we set the decoding temperature as 0.3 across all runs and repeat each system-task pair for multiple trials.

Trace Statistics. Across all systems and task sources, a total of 380 execution traces are collected after data cleaning, among which 220 traces result in task failures and are included as benchmark instances. The number of failed traces varies across systems and task sources, reflecting differences in task difficulty, execution length, domain constraints, and system specialization. Rather than enforcing a uniform failure rate, TraceElephant preserves the naturally occurring failure distributions observed during execution. This design choice allows failure attribution methods to be evaluated under realistic conditions, where failures may stem from diverse causes and arise at different stages of execution. Each failed run is treated as an independent attribution instance, capturing variability introduced by language model decoding and tool interactions.

A.3 Details about Trace Collection

To enable systematic failure attribution across heterogeneous MAS, TraceElephant adopts an automated trace collection pipeline that adapts different systems into a unified execution logging interface without modifying their original implementations.

LLM API Middleware. At the core of the trace collection pipeline is a lightweight LLM API middleware that mediates all interactions between a running MAS and its underlying LLMs. Instead of directly communicating with an external LLM service, the MAS is configured to send LLM requests through a middleware. It serves as a transparent proxy between a running multi-agent system and its underlying large language model. All LLM requests issued by agents are intercepted by the middleware, forwarded to a user-specified backend model, and logged together with the corresponding responses. This design enables complete observability of LLM-mediated decision making without modifying agent logic, prompts, or system control flow. Instead of directly communicating with an external LLM service, the multi-agent system is configured to route all LLM requests through the middleware. For each request, the middleware records the request payload (including messages and decoding parameters), forwards it to the backend model, receives the response, and returns the response to the calling agent. Since tool invocations in modern MASs are typically triggered by LLM-generated outputs, the middleware additionally captures LLM-mediated tool interactions when such information is observable at runtime. This includes the selected tool name, tool input arguments, and execution results. As a result, both language-model reasoning and subsequent tool-mediated actions are recorded in a unified, system-agnostic manner.

Trace Pre-processing. We apply lightweight pre-processing to raw traces to expose necessary information in a structured manner while preserving their original form as much as possible. Specifically, we use regular-expression matching to perform two annotations: (1) extracting and tagging agent names in the traces, and (2) categorizing step outputs as either plain LLM responses or tool-mediated interactions.

Agent Name Extraction. Agent names are extracted directly from execution-time observations using pattern matching, without introducing additional semantic inference. In many MAS implementations, agent identities are explicitly referenced in message headers or system-generated markers. For example, given the message:

```
Planner: We need to search for
relevant documentation.
```

The agent name `Planner` is extracted and associated with the current execution step.

Output type categorization. Each step output is categorized as either a plain LLM response or a tool-mediated interaction. This distinction is determined by matching tool invocation patterns in the output text. For instance, the output

```
Action: search_web
Input: "Python regex example"
```

is classified as a tool-mediated step.

Step and Agent Indexing. Each recorded trace is assigned a monotonically increasing *step_id* according to its chronological order in the execution trace. Agent identifiers (*agent_id*) are assigned based on the first occurrence of each agent name and remain consistent within a trace. Each step is linked to both the corresponding *agent_id* and the *agent name*.

A.4 Example Execution Trace

We provide a concrete example to illustrate how execution traces are recorded in TraceElephant. The example corresponds to a general travel-planning task executed by a multi-agent system.

Task. *"How should a first-time visitor plan a 3-day trip to Walt Disney World in Florida?"* This task requires multi-step reasoning, external information retrieval, and verification across multiple constraints, making it representative of realistic agentic workloads.

Trace Metadata. Each execution trace begins with trace-level metadata that provides global context for the execution.


```
{
  "trace_metadata": {
    "task_id": "WDW-TRAVEL-001",
    "task_instruction": "How should a
        first-time visitor plan a 3-
       day trip to Walt Disney World
        in Florida?",
    "system_name": "Magentic-One",
    "agent_configuration": {
      "agents": [
        "PlannerAgent",
        "SearchAgent",
        "ItineraryAgent",
        "VerifierAgent"
      ],
      "prompts": ["..."],
      "tools": ["web_search"]
    },
    "system_architecture": {
      "description": "A centrally
          orchestrated multi-agent
          system with role-
          specialized agents.",
      "code": ["orchestrator.py", "
          agents/*.py"]
    }
  }
}
```


The metadata records the task, system identity, agent composition, and architectural information. Besides for failure attribution, these fields can also provide essential context for interpreting execution behavior and comparing traces across systems.

Step-level Records. The execution trace consists of an ordered sequence of step-level records, each corresponding to a single agent action. *Step 1: Task Decomposition*


```
{
  "step_id": 1,
  "agent_id": 0,
  "agent_name": "PlannerAgent",
  "input_context": "Task: Plan a
      three-day Walt Disney World
      visit for a family.",
  "output_content": "I will break the
       task into park selection,
      ticket options, and visit order
      ."
}
```


The planner interprets the user request and decomposes it into actionable sub-tasks.

Step 2: Identifying Disney Theme Parks

```
{
  "step_id": 2,
  "agent_id": 1,
  "agent_name": "SearchAgent",
  "input_context": "Find the main
      theme parks at Walt Disney
      World.",
  "output_content": "Searching
      official Disney sources.",
  "tool_logs": {
    "tool_name": "web_search",
    "input_arguments": "Walt Disney
        World theme parks",
    "outputs": "Magic Kingdom, EPCOT,
         Hollywood Studios, and
        Animal Kingdom.",
    "execution_status": "success"
  }
}
```


This step involves an external tool invocation. The raw tool interaction is preserved verbatim in *tool_logs*. Please note that for each step of the output, there must be an *output_content*, and only when there is a tool call will there be a *tool_logs*.

Step 3: Retrieving Ticket Options


```
{
  "step_id": 3,
  "agent_id": 1,
  "agent_name": "SearchAgent",
  "input_context": "Find ticket
      options for a three-day Disney
      World visit.",
  "output_content": "Searching ticket
       types and pricing.",
  "tool_logs": {
    "tool_name": "web_search",
    "input_arguments": "Disney World
        three-day ticket options",
    "outputs": "Standard tickets
        allow one park per day; park-
        hopper tickets allow multiple
         parks per day.",
    "execution_status": "success"
  }
}
```


Here, the same agent performs another retrieval step, producing additional evidence required for planning.

Step 4: Drafting an Initial Visit Plan


```
{
  "step_id": 4,
  "agent_id": 0,
  "agent_name": "PlannerAgent",
  "input_context": "Theme parks and
      ticket options have been
      retrieved.",
  "output_content": "Draft plan: Day
      1 Magic Kingdom, Day 2 EPCOT,
      Day 3 Hollywood Studios."
}
```


The planner synthesizes retrieved information into a tentative plan.

Step 5: Feasibility Verification


```
{
  "step_id": 5,
  "agent_id": 2,
  "agent_name": "VerifierAgent",
  "input_context": "Verify whether
      the proposed three-day plan is
      feasible for a family.",
  "output_content": "The plan is
      feasible. Park-hopper tickets
      are optional but not required."
}
```


The verifier checks internal consistency and practical feasibility.

Step 6: Final Recommendation


```
{
  "step_id": 6,
  "agent_id": 0,
  "agent_name": "PlannerAgent",
  "input_context": "Verification
      completed.",
  "output_content": "Final
      recommendation: Visit Magic
      Kingdom, EPCOT, and Hollywood
      Studios on separate days with
      standard three-day tickets."
}
```


This final step produces the consolidated answer returned to the user.

This example demonstrates how TraceElephant records: (1) rich trace metadata, (2) fine-grained step-level inputs and outputs, and (3) raw tool interactions, all in a unified JSON format.

A.5 Failure Attribution Annotation

This section describes the human annotation process used to obtain failure attribution labels, i.e., (1)

the agent primarily responsible for the failure, and (2) the execution step where the failure originates.

Annotators. The annotation is conducted by three human annotators with at least one year of experience in developing and debugging MASs. The annotators are familiar with agent pipelines, tool-mediated execution, and trace-based analysis.

Annotation principle. The annotation process is conducted in three rounds. In the first round, failed execution traces from all multi-agent systems are evenly distributed among the three annotators. Each annotator independently inspects the complete execution trace, including agent actions and tool interactions. For each trace, the annotator identifies the failure-responsible agent and decisive failure step. Annotators additionally indicate whether they are confident or uncertain about each annotation.

In the second round, all traces marked as uncertain by at least one annotator are jointly reviewed. Annotators collaboratively examine the execution timeline, discuss alternative interpretations of the trace, and reconcile differences through discussion. Consensus is reached through mutual agreement rather than majority voting, and annotations are finalized only when all annotators agree on the attribution outcome.

In the final round, each annotator reviews a subset of annotations produced by the other annotators to assess consistency with the shared annotation standards. When inconsistencies or ambiguities are identified, the corresponding traces are revisited and re-annotated through further discussion until a consistent labeling standard is achieved across. This multi-round procedure follows established practices for aggregating expert judgments to approximate reliable ground truth (Krippendorff, 2018).

A.6 Evaluation Design

A.6.1 Evaluation Metrics.

Given an execution trace $\mathcal{O}(\mathcal{T})$, the failure attribution method is tasked with predicting the failure-responsible agent $\hat{a}$ and the decisive failure step $\hat{t}$. Predictions are evaluated against the ground-truth labels (a^*, t^*) annotated in our benchmark. Let $\mathcal{D}$ denote the set of failed execution traces in the benchmark. We report ***agent-level accuracy***, defined as the proportion of traces for which the predicted agent matches the ground truth: $\text{Acc}_{\text{agent}} = \frac{1}{|\mathcal{D}|} \sum_{(\mathcal{T},a^*,t^*)\in\mathcal{D}} \mathbb{I}[\hat{a} = a^*]$. Similarly, we report ***step-level accuracy***, defined as: $\text{Acc}_{\text{step}} = \frac{1}{|\mathcal{D}|} \sum_{(\mathcal{T},a^*,t^*)\in\mathcal{D}} \mathbb{I}[\hat{t} = t^*]$. Following prior work, we additionally report ***step-level accuracy with tolerance***, where a prediction is considered correct if $|\hat{t} - t^*| \leq \delta$, with δ fixed across all experiments. All results are averaged over 3 independent runs.

A.6.2 Observability Configurations.

We evaluate two categories of observability configurations. For ***Static*** configuration, we provide the overall trace information as demonstrated in Section 3.3 for failure attribution. For ***Dynamic*** configuration, besides the static information, we also provide a replayable execution environment for each trace. This setting reflects an interactive debugging workflow where static inspection is complemented by execution-based validation.

We additionally utilize ***Static w/o metadata***, ***Static w/o input***, and ***Static w/o metadata&input*** to further evaluate the necessity of different types of information for failure attribution.

A.6.3 Failure Attribution Techniques.

For above mentioned static configuration, we evaluate three commonly-used LLM-based failure attribution techniques (Zhang et al., 2025c,b; Ge et al., 2025) and two popular agent-based techniques (Yang et al., 2024). The first three techniques present the failure trace to the LLM, and let the LLM identify the responsible agent and decisive step. They differ in how the trace is provided. ***All-at-once*** provides the full trace in a single context window and asks the LLM to output the failure-responsible agent and decisive failure step. ***Step-by-step*** reveals the trace incrementally and asks the LLM to decide at each step whether it contains the failure origin, terminating when a step is selected. ***Binary search*** repeatedly asks the LLM to localize the failure to a sub-range of steps, halving the search space until a single step is identified.

For the first agent-based technique (short for ***Static Agentic***), we adopt mini-SWE-agent, which can navigate the trace information, retrieve related fields as needed, and make a conclusion gradually. Specifically, we modify the codebase of *mini-SWE-agent* by adding three tools: (1) inspecting the outputs of all executed steps; (2) inspecting the input and output of a specific step; and (3) submitting the final answer. These extensions enable the LLM

agent to flexibly and dynamically switch between global information and fine-grained local details during reasoning.

For ***Dynamic Agentic***, it first proposes candidate failure attributions derived based on static agentic technique, with the candidate steps and agents, the method then re-run the system from the corresponding execution point and issue counterfactual checks. Specifically, we first leverage the Static Agentic method to infer up to n ($n \leq 3$) candidate *mistake agent/step/reason* triples, together with their corresponding *expected oracle*. The expected oracle represents the anticipated output of the identified step if the specified mistake reason had not occurred. We then replay the task trajectory, and when execution reaches the identified mistake step, we intervene by modifying the input request of that step through an LLM API middleware. This intervention guides the agent to avoid the error induced by the mistake reason. We subsequently observe whether the next $k = 3$ steps satisfy the expected oracle and whether the previously observed failure modes reoccur or the agent deviates from the task objective.

We do not execute the entire task to completion; instead, we restrict our observation to $k = 3$ steps. This design choice is motivated by our empirical finding that many task failures stem from systemic design issues, where even corrective interventions may not guarantee success over long horizons. By focusing on a limited window of $k = 3$ steps, we assess local behavioral improvements within a bounded range, which provides a more reliable basis for diagnosis and judgment.

Note that there are other techniques for failure attribution in MASs (i.e., ECHO (Banerjee et al., 2025), AgenTracer (Zhang et al., 2025a), GraphTracer (Zhang et al., 2025b), and FAMAS (Ge et al., 2025)). However, they don't release runnable implementations or source code. We attempt to reproduce them, but couldn't obtain performance consistent with the numbers reported in original works. Therefore, we don't include them in evaluation.

A.6.4 Application Scenarios.

We evaluate attribution under two settings that simulate realistic debugging demands. In the ***With ground truth*** scenario, attribution methods are provided with task-level reference information, which is commonly available in development or diagnostic scenarios. Specifically, for GAIA and AssistantBench, we use the official task ground-truth answers as references. For SWE-Bench-Verified, where explicit reference answers are not available, we instead treat the observed test outcomes of the evaluation instances (i.e., pass/fail signals from the test suite) as ground-truth supervision. In contrast, in the ***Without ground truth*** scenario, the attribution method relies solely on the execution trace (and optional replay information), reflecting practical debugging scenarios where no definitive reference outcome is accessible.

A.6.5 LLMs Under Evaluation.

We evaluate failure attribution performance across a set of representative LLMs, covering both closed-source and open-source models. We use GPT-4o (Hurst et al., 2024) as the default LLM in the main experiments. We additionally compare against GPT-4o-mini (OpenAI, 2024), Claude-4.5-Sonnet (Anthropic, 2025) (closed-source) and Qwen3 (Yang et al., 2025), GPT-OSS-20B (Agarwal et al., 2025), DeepSeek-R1 (Guo et al., 2025) (open-source), to assess whether the findings are consistent across model families and capability levels.

A.7 Analysis on Who&When Benchmark

We report the distribution of failure agent and failure step in Who&When Benchmark in Figure 8 and 9. These trends align closely with those observed in TraceElephant (see Figure 4 and 6).

The slight difference lies in the average number of LLM invocations: in Who&When, CaptainAgent and Magentic-One respectively have an average of 9.6 and 28.8 calls, while these figures for TraceElephant are 20.5 and 29.3.

It is also worth noting that the step count for Magentic-One appears significantly higher (almost 80 steps) in Who&When as shown in Figure 9; this discrepancy stems from differences in step-counting strategies between the two benchmarks. For example, CaptainAgent omits the scheduling process among speaking agents, and ComputerTerminal executes code from the previous step directly via the command line without invoking an LLM. In Magentic-One, the process of reasoning about and selecting the next speaking agent is decomposed into multiple steps. We reconstructed the actual number of LLM invocations in the Who&When dataset: Algorithm-Generated (Captain-Agent) averages 9.6 invocations, Hand-Crafted (Magentic-One) averages 28.8, and the overall average is 15.7.

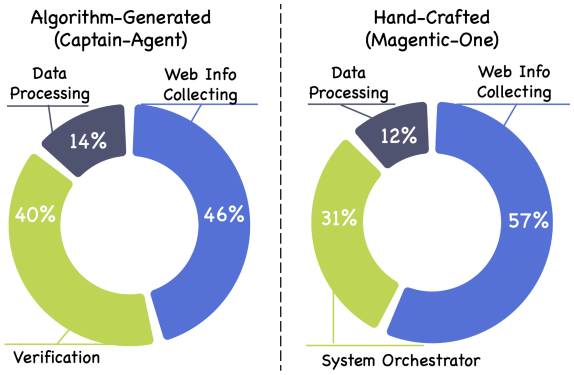

Figure 8: Distribution of failure agent in Who&When.

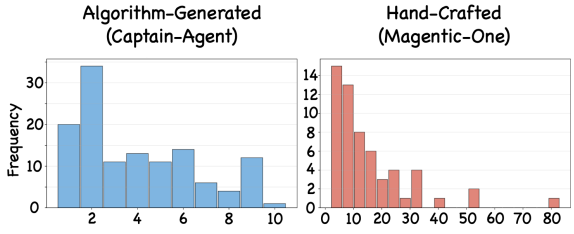

Figure 9: Distribution of failure step in Who&When.

A.8 Privacy Considerations

TraceElephant is constructed using execution traces that include step-level inputs and outputs of agents. While such information is essential for fine-grained failure attribution, it may contain sensitive content, depending on the task domain and system configuration. For example, execution traces may include user-provided instructions, intermediate reasoning artifacts, or tool invocation details that could expose proprietary logic or private information.

To reduce potential privacy risks, we adopt several conservative design choices during benchmark construction. First, all traces are collected from controlled task executions rather than from real user interactions. Tasks are predefined and do not involve personal user data. Second, the benchmark representation focuses on step-level abstractions and does not require storing raw system states or external environment snapshots. Third, when releasing the benchmark, sensitive fields can be selectively anonymized or transformed, for example by replacing raw inputs or outputs with hashed or redacted variants when appropriate.

A.9 Human Annotation Protocol and Ethics Statement

We employed three expert annotators with prior experience in developing and debugging LLM-based multi-agent systems to provide failure attribution labels for the benchmark. Each annotator was instructed to identify (i) the failure-responsible agent and (ii) the earliest execution step at which task failure becomes inevitable, based on the complete execution trace, including agent inputs, outputs, and tool interactions.

The annotation guidelines emphasized consistency, trace-based reasoning, and adherence to the formal definition of step-level and agent-level failure attribution described in Appendix 2. Annotators conducted their analysis independently in the first round, followed by joint discussion and reconciliation for cases marked as uncertain, until consensus was reached.

All annotators participated voluntarily and were fully informed of the purpose of the study and the intended use of the annotated data for research and benchmarking purposes. The annotation task involved only synthetic tasks or publicly available benchmark data (GAIA, AssistantBench, and SWE-Bench) and did not include any personal, sensitive, or user-generated data.

No external recruitment or crowdsourcing platforms were used, and no demographic data were collected. As the study involved only technical annotation of non-sensitive data by informed research participants, it did not require approval from an institutional ethics review board.

A.10 Artifact License and Usage Terms

TraceElephant including execution traces, annotations, and associated metadata, is released for research purposes only. All benchmark artifacts will be made publicly available upon publication through an open-access repository.

The execution traces and annotations are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license, which permits use, distribution, and adaptation for research purposes with appropriate attribution. The released artifacts do not contain any personal, sensitive, or proprietary information.

For third-party components referenced in the benchmark, including task sources such as GAIA, AssistantBench, and SWE-Bench, we follow their original licenses and terms of use. Users of TraceElephant are responsible for complying with the licenses of these upstream resources when using or redistributing the benchmark.

The benchmark code, including trace collection scripts and evaluation utilities, will be released under a permissive open-source license, and detailed

licensing information will be provided in the accompanying repository.

A.11 Use of AI Assistants

We used AI assistants to support limited aspects of this research, primarily for language polishing and code-related debugging during the development process. Specifically, AI tools were occasionally used to improve clarity and grammar in early drafts and to assist with debugging auxiliary scripts that do not affect the core experimental logic or results. All scientific contributions of this work, including the benchmark design, data collection, annotation protocol, experimental setup, analysis, and conclusions, were conceived, implemented, and verified by the authors. AI assistants were not used to generate experimental results, annotations, evaluation metrics, or substantive scientific claims. The authors reviewed, verified, and take full responsibility for all content in the final manuscript.