TraJudger: Structure-Guided Failure Attribution for LLM-Powered Multi-Agent Systems

Yongxin Zhao
Nankai University
Tianjin, China

Shenglin Zhang
Nankai University
Tianjin, China

Wenwei Gu*
Nankai University
Tianjin, China

Zijing Que
Nankai University
Tianjin, China

Weiwen Qian
Nankai University
Tianjin, China

Yongqian Sun
Nankai University
Tianjin, China

Li Shi
Alibaba Group
Hangzhou, China

Wei Li
Alibaba Group
Hangzhou, China

Liping Zhang
Alibaba Group
Hangzhou, China

Dan Pei
Tsinghua University
Beijing, China

Abstract

LLM-powered multi-agent systems are increasingly deployed in production, yet diagnosing their failures remains a major operational challenge. Unlike failures in cloud architectures, which can often be diagnosed using structured diagnostic information, multi-agent system failures are embedded in unstructured natural-language interactions, cascade silently across semantic boundaries, and defy attribution due to a lack of prior domain knowledge. Existing LLM-as-a-Judge approaches treat execution trajectories holistically, misattributing cascaded failures and fabricating plausible but ungrounded explanations. To overcome the limitations of existing methods, we present *TraJudger*, a structure-guided failure attribution system deployed in production at a global top-tier online shopping service provider. *TraJudger* constructs a Hierarchical Cognitive Graph from raw trajectories to recover explicit inter-step dependencies, deploys four category-specific specialist agents for typed failure detection, and performs graph-constrained backward tracing to ground root-cause identification in verified dependency paths. On 121 production and 126 benchmark failure trajectories, *TraJudger* achieves a relative improvement of at least 15% and 13.3% over state-of-the-art methods at the agent and step levels. *TraJudger* has been deployed at *Company A* for over one month, reducing average time-to-attribution from 30 minutes to under one minute.

*Correspondence to wwgu@nankai.edu.cn

This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857043

CCS Concepts: • **Computing methodologies** → **Multi-agent systems**; • **Computer systems organization** → **Maintainability and maintenance**; **Reliability**.

Keywords: Failure Attribution, LLM-Powered Multi-Agent Systems, Trajectory Analysis

ACM Reference Format:
Yongxin Zhao, Shenglin Zhang, Wenwei Gu, Zijing Que, Weiwen Qian, Yongqian Sun, Li Shi, Wei Li, Liping Zhang, and Dan Pei. 2026. TraJudger: Structure-Guided Failure Attribution for LLM-Powered Multi-Agent Systems. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857043

1 Introduction

Large Language Model (LLM)-powered multi-agent systems (MAS) are increasingly entrusted with complex production tasks [9, 14, 15, 26, 29]. At *Company A*, we operate several such platforms, each serving over ten thousand monthly active users in domains spanning software development, data analysis, and information retrieval. These platforms orchestrate heterogeneous LLM agents (*e.g.*, Orchestrator, WebSurfer) that cooperate through multi-turn natural language interactions to fulfill queries [4, 7]. As deployment scales, reliability has emerged as a first-order concern [5, 37, 38]: a non-trivial fraction of executions yield incorrect or constraint-violating outcomes, imposing a sustained diagnostic burden on on-call engineers (OCEs).

Maintaining reliability hinges on *failure attribution* [19, 44, 50], which identifies the root-cause agent, the root-cause step, and the failure type from a failed trajectory and artifacts (*i.e.*, task description, agent prompts, tool registry), as shown in Figure 1. At *Company A*, this task has long been performed manually. Upon receiving a user-reported failure, OCEs inspect the execution trajectory, defined as the sequence of agent actions and system states recorded during task execution, to localize the root cause. These trajectories

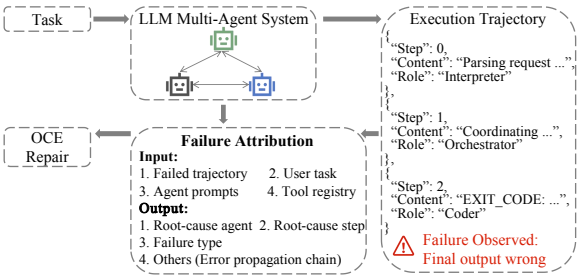


Figure 1. Failure attribution workflow. Given a failed trajectory together with the user task and agent prompts, the attribution module identifies the root causes to guide repair.

typically span tens of thousands of tokens. Such manual diagnosis requires approximately 30 minutes per case, even for experienced OCEs, hindering scalability to production.

Automated failure diagnosis has been studied extensively for cloud services [1, 11], microservices [25, 46, 47], and data center networks [28, 36, 40]. The success of these automated diagnosis approaches rests on two foundational properties: (1) structured diagnostic information, available directly from key performance indicators, distributed traces, and topology-derived dependency graphs, or extracted from semi-structured and unstructured logs [6, 8, 23, 35, 46, 47]; and (2) well-characterized failure taxonomies, in which each category exhibits distinguishable signatures codified into troubleshooting guides and domain expertise [17, 25, 40]. These diagnostic foundations are less readily available in LLM-powered MAS. Interactions within these systems unfold in natural language [4], and structured runtime metadata alone does not expose the semantic dependencies among steps. Failure evidence often resides in the semantic content of trajectories, limiting the effectiveness of conventional telemetry alone for attribution [18, 33].

To bridge the gap, the prevailing paradigm is LLM-as-a-Judge [2, 44, 48], which leverages the understanding and reasoning capabilities of frontier models to analyze raw trajectories directly. While adequate for locally constructed MAS in controlled settings, our empirical analysis of *Company* $\mathcal{A}$'s failures exposes three challenges (Figure 2) that undermine this paradigm in real-world deployments.

Challenge #1: Hidden error cascades across semantic boundaries. In MAS, a subtle upstream error propagates silently through natural-language context [18], accumulating across downstream agents until it surfaces far from its origin. Unlike microservice architectures, where cascades are captured by distributed traces [47] and alert storms are aligned with topology links [40], MAS expose no explicit dependency structure for retracing this path. LLM-as-a-Judge consequently misattributes cascades to the symptom-proximate agent: in our production data, the agent-level accuracy of state-of-the-art judges falls from 0.67 to 0.39 once trajectories exceed ten steps.

Challenge #2: Type-heterogeneous failures requiring divergent detection strategies. MAS failures span orthogonal cognitive categories, including constraint violations, hallucinations, execution loops, and collaboration breakdowns, each demanding a distinct reasoning strategy. Traditional systems surface diverse failures through a shared observability stack and thus admit a unified diagnostic framework [16, 25, 47]; in MAS, by contrast, no single prompt can encode all evidence patterns required across failure categories. A monolithic judge, therefore, conflates failures, with 53.3% of constraint-violation cases in our evaluation misclassified as hallucinations.

Challenge #3: Hallucinated root-cause attribution without structural grounding. Root-cause reasoning in traditional systems is anchored by topologies [40], incident archives [1], and dependency graphs [47]. MAS workflows are dynamically composed, lack precedent, and have no historical failure repository to ground reasoning [12, 20]. Absent such structural anchors, LLM judges drift toward narrative plausibility: only 42.1% of a frontier model's explanations in our verification are grounded in trajectory evidence; the remainder fabricate plausible yet unsupported stories.

To address the above challenges, we propose *TraJudger*, a production-ready failure attribution framework for LLM-powered MAS. Specifically, to mitigate hidden cross-agent error cascades, *TraJudger* first constructs a Hierarchical Cognitive Graph (HCG) from raw trajectories to recover explicit structural dependencies. To handle type-heterogeneous failures, it then employs multiple category-specific specialists that diagnose failures in parallel using tailored reasoning strategies. To reduce hallucinated attribution, it finally performs conduit-aware backward tracing over verified dependency paths, ensuring that root-cause localization is grounded in structural evidence.

Our main contributions are summarized as follows:

- We conduct an empirical study on failures from *Company* $\mathcal{A}$'s LLM-powered MAS platforms, identifying three recurring attribution challenges: hidden cross-agent cascades, type-heterogeneous failures, and ungrounded attribution.
- We design and implement *TraJudger*, a structure-guided failure attribution system that makes inter-agent dependencies explicit, invokes specific specialists for failure detection, and performs graph-constrained tracing for attribution. By separating structural reconstruction, specialized detection, and constrained reasoning, *TraJudger* improves accuracy and interpretability in MAS failure attribution.
- We deploy *TraJudger* in production at a top-tier online shopping service provider, where LLM-powered MAS platforms collectively serve over ten thousand monthly active users and process over ten thousand tasks per month. *TraJudger* reduces time-to-attribution from 30 minutes to under one minute, and outperforms the state-of-the-art methods

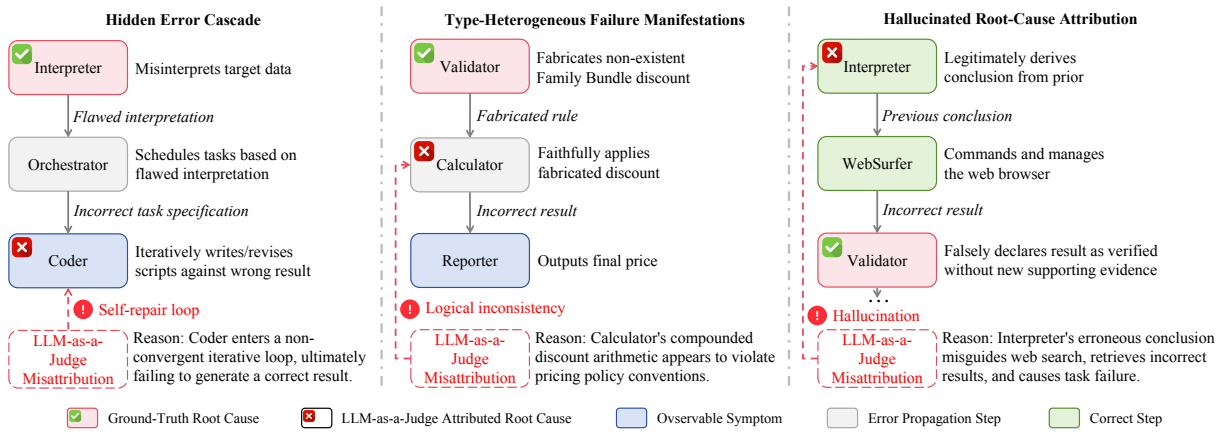


Figure 2. Three systematic challenges observed in production failure attribution. Hidden cascade: the root cause at an upstream agent propagates silently through semantic boundaries; the judge misattributes it to the symptom-proximate step. Type confusion: a monolithic judge conflates heterogeneous failure categories. Hallucinated attribution: the judge fabricates a narrative unsupported by trajectory evidence.

by at least 15% and 13.3% at the agent and step levels, respectively. Beyond the accuracy and efficiency improvements, our deployment distills four operational lessons and highlights limitations that point to open problems for the community.

2 Background

2.1 Failure Landscape

When an end user marks a response as incorrect, the observability platform of the MAS records the full execution trajectory as a failed case for subsequent analysis. We analyze 121 such failure trajectories collected from a production multi-agent system at *Company A* that serves over ten thousand monthly active users. The system comprises four agents: a *Clarifier* (user intent interpretation), a *Leader* (task scheduling), a *Coder* (code generation and execution), and a *WebSearcher* (web retrieval). Agents communicate exclusively through natural-language messages, and the orchestration topology is determined dynamically on a per-query basis.

The dataset exhibits substantial heterogeneity along three dimensions. First, trajectories vary widely in length and size, spanning 2–47 interaction steps and 554 to more than 10,000 tokens. Second, failures are not concentrated in a single mode but instead cover ten fine-grained failure types across four categories (Table 1). Third, many failures unfold across multiple dependent steps rather than at a single isolated point: in 45.5% of cases, the annotated error propagation chain spans at least three interaction steps. Together, these observations indicate that failure attribution in production MAS should handle long contexts, heterogeneous failure modes, and multi-step error propagation.

2.2 Failure Taxonomy

Existing taxonomies for multi-agent failures adopt various perspectives: system-level [5], execution-architecture [10], and agent-environment interaction [31]. Building on these prior efforts and on our production failure patterns, we define the taxonomy shown in Table 1. The ten failure types are grouped into four categories based on the cognitive nature of the failure, enabling category-specific detection strategies (§3.3). This taxonomy is not intended to be exhaustive; as discussed in §6.3, it should evolve with the platform as new agent roles and orchestration patterns introduce previously unseen failure modes.

2.3 Empirical Evidence

§1 identified three challenges of LLM-as-a-Judge attribution. Here we present the quantitative evidence behind each claim, derived from evaluations of state-of-the-art approaches [44] on our production dataset. We report attribution accuracy at two granularities: *agent-level accuracy (Agent Acc)*, which measures whether the predicted root-cause agent matches the human annotation, and *step-level accuracy (Step Acc)*, which measures whether the predicted root-cause step index exactly matches the annotation.

2.3.1 Evidence for Challenge #1: Cascade-Induced Misattribution. Table 2 reports the attribution accuracy of the three LLM-as-a-Judge strategies from Who&When [44], stratified by trajectory length. On trajectories with 10 or fewer steps, the strategies achieve an average agent-level accuracy of 0.67 and step-level accuracy of 0.60. On trajectories exceeding 10 steps, where longer cascade chains are more likely, accuracy degrades to 0.39 and 0.23, respectively. This degradation confirms that holistic judges are systematically misled by cascade depth: without an explicit dependency structure to traverse, they default to attributing fault to the symptom-proximate agent.

Table 1. Taxonomy of failure modes in LLM-powered multi-agent systems.

Category	Failure Mode	Description
Constraint and Role Violations	Task Constraint Deviation	Output violates explicit task requirements or hard constraints.
	Role Deviation	Agent performs actions beyond its assigned functional scope.
Logic and Cognitive Errors	Hallucination	Fabrication of non-existent information or data.
	Logical Inconsistency	Current output contradicts prior statements or established facts.
	Overconfidence	Authoritative but incorrect assertions driven by excessive confidence.
Execution and Flow Breakdowns	Step Looping or Redundant	Trapped in repeated actions or executing unnecessary steps.
	Premature Termination	Task declared complete before objectives are fully met.
	Tool Error	Failures in tool invocation (*e.g.*, param mismatches, malformed input).
Collaboration and Context Failures	Unclear Communication	Passing ambiguous information to downstream agents.
	Context Ignorance	Disregarding prior conversation history or critical corrections.

Table 2. Attribution accuracy of LLM-as-a-Judge strategies, stratified by trajectory length.

Method	Steps ≤ 10		Steps > 10	
	Agent Acc	Step Acc	Agent Acc	Step Acc
All-at-Once	0.7	0.644	0.385	0.308
Step-by-Step	0.644	0.578	0.462	0.308
Binary Search	0.656	0.578	0.308	0.077
Average	0.667	0.6	0.385	0.231

Table 3. Performance analysis of holistic judgment across failure categories.

Failure Mode	#Cases	Acc	Top Confusion
Task Constraint Deviation	30	0.4	Hallucination (0.533)
Role Deviation	3	0.334	Hallucination (0.667)
Hallucination	32	0.594	Overconfidence (0.188)
Logical Inconsistency	10	0.3	Hallucination (0.3)
Overconfidence	25	0.32	Hallucination (0.48)
Step Looping or Redundant	7	1.0	-
Premature Termination	3	1.0	-
Tool Error	5	0.8	Overconfidence (0.2)
Unclear Communication	2	0.0	Hallucination (1.0)
Context Ignorance	4	0.5	Overconfidence (0.25)

2.3.2 Evidence for Challenge #2: Category Conflation.

Table 3 reports per-type classification accuracy when GPT-5.4 serves as a holistic judge with a unified prompt incorporating our full failure taxonomy. The judge achieves near-perfect accuracy on execution flow failures (*e.g.*, step looping at 1.0) but drops sharply on semantically subtle categories: hallucination detection falls to 0.59, and logical inconsistency to 0.3. The dominant misclassification pattern is systematic rather than random: 53.3% of Task Constraint Deviation instances are misclassified as Hallucination, and overconfidence errors are frequently absorbed into the hallucination label. This confirms that a monolithic judge collapses fine-grained distinctions into coarse catch-all categories. Each failure category demands a distinct verification procedure. A judge that must maintain all procedures simultaneously tends to default to the most salient heuristic, rather than executing category-specific verification at each step.

2.3.3 Evidence for Challenge #3: Ungrounded Root-Cause Reasoning.

We task GPT-5.4 [30] with attributing root causes for failed trajectories where ground-truth error propagation chains are known, requiring the model to explain the propagation mechanism linking the selected root-cause step to the eventual failure. Manual verification by two engineers, each with over five years of operations and maintenance experience, reveals that only 42.1% of the explanations produced by the judge are substantively grounded in trajectory evidence. Among ungrounded attributions, the LLM fabricates facts to force-fit a failure reason onto the selected step, even when that step executes correctly. We further observe cases where the judge arrives at the correct root-cause step yet provides an entirely incorrect propagation mechanism, citing non-existent data dependencies or inventing interactions that never occur in the trajectory.

Taken together, these analyses reveal three distinct failure mechanisms rather than a single generic effect of trajectory complexity. The length-stratified results are consistent with symptom-proximity bias, the taxonomy-based evaluation exposes category conflation, and the explanation analysis reveals ungrounded reasoning. *TraJudger* addresses these mechanisms through separate design components: dependency-graph construction and backward tracing mitigate symptom-proximity bias, category-specific specialists apply distinct verification procedures, and graph-constrained traversal restricts root-cause reasoning to extracted dependency paths. These mappings may also suggest concrete directions for extending prior LLM-as-a-Judge methods.

2.4 Operational Constraints and Design Requirements

Our production deployment imposes three operational constraints that shape the design of *TraJudger* and rule out several families of existing techniques.

Constraint #1: Unstructured-trajectory input. Traditional operation and maintenance tools [8, 16, 47] leverage structured telemetry and event templates extracted from semi-structured or unstructured logs. In multi-agent systems, however, failures are embedded in the semantic content of natural-language interactions rather than in machine-parseable signals; there are no log templates to extract or structured event types to match, making these tools difficult to apply directly.

Constraint #2: Non-intrusive attribution. Intervention-based methods [22, 49] validate causal hypotheses by modifying agent inputs or intermediate states and re-running parts of the workflow. In our production environment, such re-execution is prohibitively expensive due to cost, the non-determinism of LLM outputs, and potential side effects on user-facing state. We therefore require a diagnostic approach that operates on recorded trajectories without perturbing the running system.

Constraint #3: No task-specific training. Prior fine-tuning-based approaches [18, 27, 42, 43] rely on supervised or reinforcement learning to train dedicated failure trackers. We do not adopt this paradigm because our production data does not provide sufficient labeled failure trajectories to support reliable task-specific training. Moreover, our multi-agent platform is under active architectural evolution, with new agents, tools, and orchestration policies introduced frequently; each change would necessitate renewed data collection and retraining. Instead, we build on frontier LLMs accessed through remote APIs, which better match our operational setting by avoiding retraining and the additional infrastructure required to host specialized models. Although fine-tuning may yield higher in-distribution accuracy, an API-based design is more practical given limited labeled data and rapid system change.

These constraints lead us to design a system that operates on raw unstructured trajectories, does not rely on task-specific training, and performs diagnosis without re-execution.

3 TraJudger Design

3.1 Overview

As shown in Figure 3, *TraJudger* adopts a three-stage architecture that progressively transforms unstructured natural-language trajectories into structured diagnoses.

Stage 1: HCG construction. *TraJudger* parses the task description, system prompts, tool registry, and execution trajectory into a hierarchical semantic graph that captures tasks, constraints, agent roles, actions, and tool invocations. The HCG serves as a shared structural evidence base for all subsequent stages.

Stage 2: Multi-specialist failure detection. Four independent specialist agents (Compliance, Logic, Flow, and Context) examine the trajectory from complementary perspectives.

Table 4. HCG node types and their sources in the multi-agent runtime.

Layer	HCG Element	Source in MAS Runtime
Static	Task node	User query (subgoals and constraints)
	Role node	Agent system prompt (role and rules)
	Tool node	Tool registry
Dynamic	Intention node	Trajectory step (extracted intent)
	Action node	Trajectory step (extracted action)

Each specialist consumes HCG-structured information and emits typed failure findings in a unified format.

Stage 3: Conduit-aware failure attribution. *TraJudger* infers inter-step trajectory dependencies and instantiates them as edges on the HCG. It then traces backward from the final failure along these edges, explicitly modeling conduit steps (correct steps that propagate upstream errors), and produces a ranked diagnostic report with propagation chains.

Design principles. *TraJudger* is built on two design principles that address the structural gap identified in §1: (1) Structure before reasoning: the HCG reconstructs the explicit dependency representation that traditional systems obtain from distributed tracing, enabling downstream stages to reason over structural relations rather than raw long text. (2) Detection separated from attribution: failure detection (Stage 2) enumerates candidate failures using category-specific strategies, while failure attribution (Stage 3) localizes root causes via graph-constrained path search. This decoupling prevents a single LLM call from conflating failure identification with structure-guided reasoning.

3.2 Hierarchical Cognitive Graph Construction

The HCG reconstructs the explicit dependency structure that multi-agent systems lack (§1). As shown in Figure 4, it converts unstructured execution trajectories into a two-layer semantic graph: a static cognitive layer encoding task goals, agent roles, and tools fixed before execution, and a dynamic execution layer encoding the thought-action sequence produced at runtime. Intra-layer edges capture temporal and logical dependencies within the dynamic layer; cross-layer edges anchor each dynamic step to the static premises it should obey. Table 4 summarizes the node types and their sources.

HCG construction relies solely on artifacts already available to the MAS runtime, including the user task, agent system prompts, tool registry, and recorded execution trajectory, as summarized in Table 4. It requires no additional configuration files.

Nodes and edges are extracted by an LLM through structured prompting. Each node is grounded in a specific span of the input text, while each candidate edge must be supported by explicit input evidence, with uncertain relations excluded during extraction.

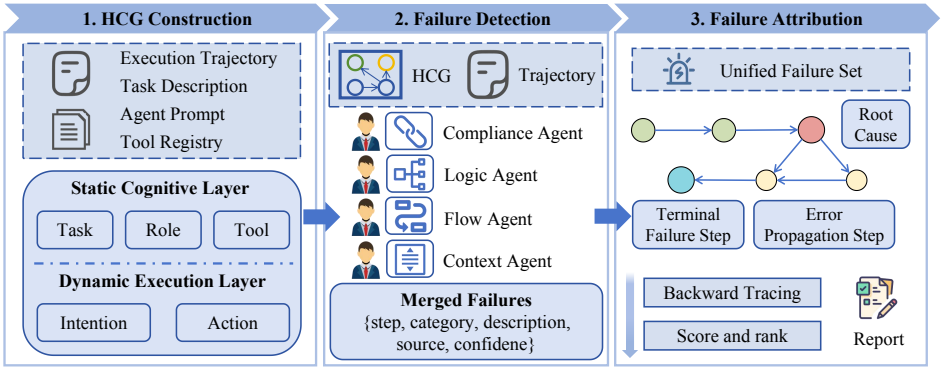


Figure 3. Architecture of *TraJudger*. Stage 1 constructs an HCG from the raw trajectory and task specification. Stage 2 dispatches four category-specific specialists in parallel over the HCG. Stage 3 infers trajectory dependencies, traces backward from the final failure, and produces a ranked diagnostic report with evidence chains.

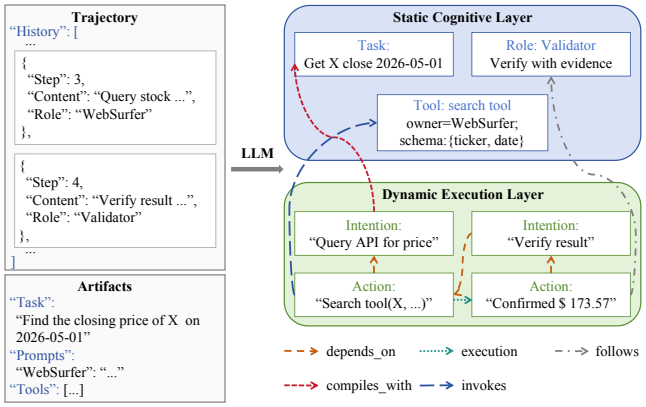


Figure 4. The process of HCG construction.

3.2.1 Static Cognitive Layer.

The static layer captures three categories of premises.

Task nodes. An LLM is invoked with a task-decomposition prompt to parse the task description provided by the user into a list of structured items, each with an identifier and a natural-language description. Items include both subgoals and hard constraints; both become task nodes, since downstream stages (Compliance Specialist, cross-layer linking) treat them uniformly as premises that dynamic steps should satisfy. The identifier of each node serves as an anchor for cross-layer links.

Role nodes. For each agent in the system, the role description and behavioral rules are extracted from its system prompt and represented as role nodes. The role description and its constraints are co-located in the same node group so that the Compliance Specialist can check the behavior of an agent against the full specification in a single lookup. Since these definitions are stable within a given MAS, role nodes are constructed once and reused across trajectories.

Tool nodes. Each available tool is represented as a node recording: tool name, owning agent, input schema (parameter names and types), per-invocation arguments, return value summary, invocation status (success or failure), and error message if applicable. These records enable tracing how tool outputs propagate through the trajectory, but do not support diagnosis of failures internal to the external source.

3.2.2 Dynamic Execution Layer.

Upon task completion, an LLM constructs the dynamic execution layer from the recorded trajectory by extracting two components for each step t: what the agent plans to do and what it actually does. These become a pair of nodes, Intention_t and Action_t, connected by two types of intra-layer edges:

Execution edges. $\text{Action}_{t-1} \rightarrow \text{Action}_t$ for adjacent turns, capturing temporal ordering.

Depends_on edges. $\text{Action}_t \rightarrow \text{Intention}_t$ (action predicated on intention) and $\text{Intention}_t \rightarrow \text{Action}_{t-1}$ (intention derived from the previous outcome).

3.2.3 Cross-layer Linking.

Cross-layer edges anchor dynamic behavior to static premises, making deviation judgments both retrievable and refutable. During dynamic-layer construction, the LLM produces candidate links for each step, materialized as:

Complies_with edges. $\text{Intention}_t \rightarrow \text{Task}_j$ and $\text{Action}_t \rightarrow \text{Task}_j$, indicating that the step serves subgoal j or respects constraint j.

Follows edges. $\text{Intention}_t \rightarrow \text{B}(a)_*$, indicating that the intention is derived from the role definition or behavioral constraint of agent a.

Invokes edges. $\text{Action}_t \rightarrow \text{Tool}(\cdot)$ for each tool called in step t.

Together, these three edge types anchor every dynamic action to the corresponding task goal, role definition, and tooling, providing a structurally aligned scope for the subsequent stages.

3.3 Multi-Specialist Failure Detection

To address the category-conflation problem identified in Challenge #2 (§1), *TraJudger* decomposes failure detection

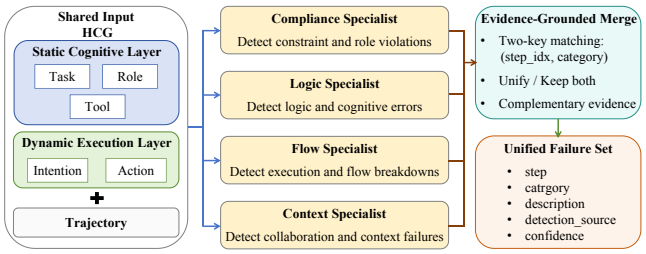


Figure 5. Multi-specialist failure detection workflow.

across four focused specialists rather than relying on a single monolithic judge, as shown in Figure 5. Each specialist attends to a single cognitive dimension, reducing per-call complexity and improving coverage.

3.3.1 Four Category-Specific Specialists. Following the failure types in Table 1, *TraJudger* deploys four LLM-driven specialist agents. Each specialist takes the HCG and the trajectory as input but attends to different node and edge types.
Compliance specialist (covers *Task Constraint Deviation* and *Role Deviation* in Table 1). Using task nodes and role nodes as the reference, this specialist checks, step by step, whether each action node complies with the task constraints and with the role definition (and its constraints) of the acting agent.
Logic specialist (covers *Hallucination*, *Logical Inconsistency*, and *Overconfidence* in Table 1). This specialist focuses on the consistency between intention and action nodes within a step and on the reasoning coherence between adjacent steps. It targets three classes of issues: intention-action inconsistency, logical discontinuity, and factual hallucinations.
Flow specialist (covers *Step Looping or Redundant*, *Premature Termination*, and *Tool Error* in Table 1). This specialist identifies redundant actions or loops, premature termination before task completion, and mishandling of tool-call return values. Omitted steps are identifiable only when they manifest as unmet task obligations or premature termination; otherwise, unobserved steps are not reconstructed.
Context specialist (covers *Unclear Communication* and *Context Ignorance* in Table 1). This specialist follows the depends_on edges to trace information flow across agents, checking whether critical information is correctly propagated and whether previously confirmed key facts are subsequently forgotten.

3.3.2 Evidence-Grounded Annotation. After the four specialists complete their analyses independently, each emits, for every finding, the failure category, the associated step index, a failure description, and the detection source. The system then deduplicates and merges findings across specialists using a two-key matching criterion: two findings are considered duplicates if they share the same step index and the same failure category from Table 1. When a match is found, the system retains a single entry and appends the description and detection source from each specialist as complementary evidence. The merged results form a unified failure set $\mathcal{F}$ in a normalized format for the next stage. The four specialists have no execution dependencies on one another and run in parallel to reduce end-to-end latency.

3.4 Conduit-Aware Failure Attribution

This stage addresses Challenge #1 (hidden error cascade) and Challenge #3 (hallucinated attribution) jointly. To trace cascades that propagate silently through natural-language content, we construct an explicit dependency graph on top of the HCG; to reduce hallucinated attribution, we constrain all attribution reasoning to verified paths in this graph.

3.4.1 Trajectory Dependencies and Conduit Steps. The HCG's dynamic layer already captures sequential (execution) and local (depends_on) edges between adjacent steps. In this stage, *TraJudger* augments the dynamic layer with longer-range dependency edges that capture semantic dependencies between non-adjacent steps. Given the HCG and the unified failure set $\mathcal{F}$, a single LLM call analyzes the full trajectory and outputs all dependency edges. Specifically, for each pair of steps $t' < t$, if the later action depends on the result of the earlier one (such as a tool output or a reasoning conclusion), the call emits a directed edge $\text{Action}_{t'} \rightarrow \text{Action}_t$. During dependency extraction, the LLM is instructed to retain an edge only when the trajectory provides explicit evidence of data or control flow between the corresponding steps, while omitting uncertain relations. These constraints are designed to reduce unsupported dependencies. The resulting edges, together with the existing edges, form a trajectory dependency graph $\mathcal{D}$, whose nodes are action steps and whose edges are directed dependencies.

As shown in §2.1, 45.5% of production failures involve error propagation chains spanning three or more steps, where correct intermediate steps faithfully forward upstream errors without being flagged as faulty themselves. A key property of $\mathcal{D}$ is that it explicitly models this phenomenon. We term such a step a *conduit step*: it is not flagged as faulty by any specialist, yet it propagates an upstream error downstream. Conduit steps are essential for complete tracing: if backward search were restricted to only faulty steps in $\mathcal{F}$, the root cause and the final symptom might have no connecting path in $\mathcal{D}$ whenever non-faulty propagation steps lie between them. By retaining conduit steps in the dependency graph, *TraJudger* preserves the full error propagation chain from root cause to observable symptom.

3.4.2 Backward Tracing. Given $\mathcal{D}$ and the failure set $\mathcal{F}$, *TraJudger* performs backward tracing to identify root causes. Because all dependency edges are already established during graph construction, backward tracing reduces to a deterministic graph-reachability computation. First, *TraJudger* follows edges backward from the terminal failure step T in

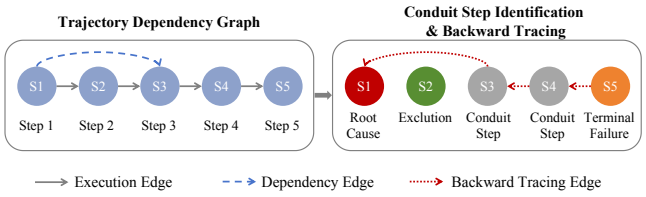


Figure 6. Backward ctracing workflow.

$\mathcal{D}$ to collect all ancestor nodes. It then filters these ancestors to retain only those flagged as faulty in $\mathcal{F}$. Among the faulty ancestors, a step is confirmed as a root cause if none of its own ancestors are themselves faulty. Finally, for each root cause and terminal failure, the path between them in $\mathcal{D}$ forms a error propagation chain, and non-faulty nodes on these paths are marked as conduit steps. The resulting set of faulty ancestors is denoted F_A.

This design decouples LLM reasoning from graph traversal over the dependency structure. The LLM is invoked once to construct $\mathcal{D}$ by determining which dependency edges exist; all subsequent backward tracing and root-cause identification are deterministic graph operations. This separation ensures that reasoning is constrained to the dependency structure established during graph construction, reducing attribution from open-ended narrative generation to structured graph traversal.

3.4.3 Root Cause Scoring and Output. As shown in Figure 6, *TraJudger* first infers inter-step dependencies and encodes them as edges in the HCG. It then performs backward tracing from the final failure along these edges, explicitly identifying conduit steps that propagate the error without originating it. Based on the traced error propagation chain, *TraJudger* finally scores each candidate root cause s_i (with failure type f_i and LLM-reported confidence $c_i \in [0, 1]$) by combining three factors: (1) *severity* of the candidate's own failure, (2) *impact scope* across downstream steps, and (3) *propagation depth* along the error propagation chain:

$$\text{score}(s_i) = w(f_i) \cdot c_i + \sum_{s_j \in \text{reach}(s_i, T)} \frac{w(f_j) \cdot c_j}{\text{pathlen}(s_i, s_j)} \quad (1)$$

where $w(f) \in [0, 1]$ is a severity weight assigned to failure type f based on operational impact, $\text{reach}(s_i, T) = \{s_j \in F_A \mid s_j \text{ is a descendant of } s_i \text{ on paths to } T\}$ is the set of faulty steps that lie between s_i and the terminal failures, and $\text{pathlen}(s_i, s_j)$ is the shortest-path distance between them in $\mathcal{D}$. The base term captures the candidate's own severity; the summation captures impact scope through linear accumulation over reachable downstream failures, with propagation depth acting as an inverse-distance attenuation. Scores are normalized by the maximum across all candidates, and candidates are ranked in descending order.

The final output is a structured diagnostic report containing: the ranked root causes with evidenced propagation chains, secondary failures labeled as propagated or independent, the error propagation path with conduit steps explicitly marked, and a repair priority recommendation that assigns the highest priority to root causes whose fix is expected to simultaneously resolve downstream propagated failures. Notably, automatic repair is beyond the scope of failure attribution. Instead, *TraJudger* reports the identified failure type and error propagation path to help OCEs prioritize remediation targets.

4 Implementation

TraJudger is implemented in approximately 3,400 lines of Python. The HCG is materialized as a `networkx.DiGraph` [13], constructed once in Stage 1 and shared with the specialists and the dependency analyzer. Intermediate results are represented as plain Python dataclasses, allowing any stage to be cached, replayed, or inspected without re-issuing LLM calls. The same entry point accepts the trajectory formats of *Company* $\mathcal{A}$ and Who&When [44].

All LLM traffic is routed through a single client built on the OpenAI Python SDK, with a configurable base URL that directs requests to Qwen3.7-Max [39] or GPT-5.4 [30]. All calls pin the temperature to 0 and request native JSON-mode output; malformed responses are caught and treated as empty findings, ensuring that a single bad response cannot abort a diagnosis. The four specialists share no execution dependencies and are dispatched concurrently through `ThreadPoolExecutor`, keeping end-to-end latency under one minute. The attribution stage constructs the dependency graph and the self-corrected-step map in a single LLM call, invokes `nx.ancestors` for backward tracing, and falls back to deterministic score-based ranking whenever the abductive call fails JSON parsing.

Production integration. *TraJudger* is integrated with the observability data platform as an asynchronous post-mortem diagnostic service that operates on recorded trajectories. *TraJudger* performs no inline diagnosis and invokes no LLMs during MAS execution. When an end user marks a response as incorrect, the platform records the trajectory as a failed case indexed by task ID. Diagnosis is user-initiated: the user locates the failed trajectory on the platform and triggers *TraJudger* with a single click. OCEs can either look up a specific trajectory via the task ID reported by the user or periodically collect all failed trajectories for batch diagnosis.

5 Evaluation

Our evaluation answers the following research questions:

- **RQ1**: How does *TraJudger* compare to state-of-the-art baselines in attribution accuracy?
- **RQ2**: How much does each component of *TraJudger* contribute to overall performance?
- **RQ3**: Are inference latency and token cost acceptable for production deployment?

• **RQ4**: How accurate are the generated error propagation chains?

5.1 Experimental Setup

5.1.1 Datasets. *TraJudger* performs root-cause attribution on trajectories that are already known to have failed; determining whether a trajectory succeeds or fails is outside the scope of this work. We evaluate on two failure datasets:
D1: Production failure trajectories. We collect 121 failure trajectories from the production MAS described in §2.1. Specifically, we submit questions from the GAIA benchmark [24] to the system, execute them end-to-end, and retain only those trajectories whose final outputs disagree with the ground-truth answers.
D2: Who&When benchmark. We adopt the algorithm-generated subset of the Who&When benchmark [44], which provides 126 failure trajectories with fine-grained failure annotations.

5.1.2 Evaluation Metrics. We measure attribution accuracy at two granularities, along with failure-type accuracy: *Agent-level accuracy, Step-level accuracy,* and *Failure-type accuracy.* Failure-type accuracy is computed over the subset of cases with correct step-level and agent-level attribution. All trajectories are labeled by two engineers independently, and inconsistent labels are resolved through discussion. Failure types are mapped to the ten modes listed in Table 1.

5.1.3 Baselines. We compare *TraJudger* against three categories of baselines. Since none of these methods was originally designed to classify failure types, we augment each with the taxonomy in Table 1 as additional input to ensure a fair comparison on failure-type accuracy.
Who&When evaluators [44]. We adopt the three evaluators proposed in Who&When: *All-at-once*, which identifies the responsible agent and the decisive failure step from the full trajectory in a single pass; *Step-by-step*, which examines the trajectory sequentially and terminates once a failure is detected; and *Binary-search*, which recursively narrows the failure localization by bisecting the trajectory.
ECHO [2]. ECHO improves attribution accuracy by integrating hierarchical context modeling, objective-based evaluation, and consensus voting.
RAFFLES [48]. RAFFLES is an iterative evaluation framework in which a central Judge investigates failures with support from specialized Evaluators that assess both system components and the reasoning of the Judge, thereby maintaining a history of hypotheses.

Model fine-tuning methods [18, 27, 42, 43] are excluded because a fair comparison is infeasible. Neither D1 nor D2 provides sufficient labeled data to train these models, and no publicly available pre-trained checkpoints exist. Training on such undersized datasets would produce an unfairly weak baseline that misrepresents the capability of each method.

5.1.4 Evaluation Implementation. To assess stability across model families, we conduct all experiments with two representative LLMs: Qwen3.7-Max [39] and GPT-5.4 [30], both accessed via official APIs. For each combination of dataset and base model, we run *TraJudger* and every baseline independently; all modules within a single method share the same base model to ensure internally consistent comparisons. We set the temperature to T=0 to maximize reproducibility and process every trajectory three times, averaging the results to mitigate residual stochasticity.
Diagnosis without ground-truth answers. To match production conditions, neither *TraJudger* nor any baseline is provided with the ground-truth answer of the underlying task; diagnosis relies solely on the trajectory content.

5.2 Overall Performance (RQ1)

Table 5 reports the agent-level, step-level, and failure-type accuracy of *TraJudger* and five baselines on D1 and D2 under two base models. Across all datasets, base models, and metrics, *TraJudger* consistently achieves the highest accuracy, surpassing the strongest baseline in each setting.
Agent- and step-level accuracy. With Qwen3.7-Max [39], *TraJudger* achieves agent- and step-level accuracy of 0.868 and 0.802 on D1, outperforming the best baseline (RAFFLES [48]) by 14.9 and 14.1 percentage points. On D2, *TraJudger* reaches 0.714 and 0.619, while the strongest baseline (All-at-Once [44]) attains 0.587 and 0.413, yielding gains of 12.7 and 20.6 points. Under GPT-5.4 [30], all methods improve, yet *TraJudger* retains the lead with 0.884 and 0.843 on D1 and 0.786 and 0.691 on D2, exceeding the best baseline in each setting by 9.9 to 12.0 percentage points. The larger gains at the step level suggest that accurate localization of the root-cause step is more challenging than identifying the responsible agent.
Failure-type accuracy. *TraJudger* also achieves the highest failure-type accuracy, reaching 0.773 and 0.755 on D1 and 0.859 and 0.828 on D2 under the two base models, exceeding the strongest baseline by 10.5 to 19.2 points. The advantage is marked on D2, where *TraJudger* outperforms RAFFLES [48] and Binary Search [44], the best baselines under Qwen3.7-Max [39] and GPT-5.4 [30], respectively. This indicates that the error propagation chain produced by *TraJudger* captures not only where the failure occurs but also how it arises, enabling more accurate failure-type identification.
Robustness across datasets and base models. Baselines degrade noticeably from D1 to D2. For example, the step-level accuracy of All-at-Once [44] with Qwen3.7-Max [39] drops from 0.645 to 0.413, a relative decrease of 36.0%, whereas *TraJudger* declines from 0.802 to 0.619, a smaller relative decrease of 22.8%. A similar pattern holds under GPT-5.4 [30]. Moreover, *TraJudger* with Qwen3.7-Max [39] outperforms all GPT-5.4 [30] baselines on both datasets across all three reported metrics. Within the evaluated model settings, these

Table 5. Overall performance comparison.

Seed LLM	Method	D1					D2				
		Agent Acc	Step Acc	Type Acc	Avg. Time	$/case	Agent Acc	Step Acc	Type Acc	Avg. Time	$/case
Qwen3.7-Max	All-at-Once [44]	0.694	0.645	0.603	**3.26 s**	**0.01**	0.587	0.413	0.539	**2.42 s**	**0.01**
	Step-by-Step [44]	0.661	0.603	0.575	13.12 s	**0.01**	0.437	0.31	0.513	13.97 s	**0.01**
	Binary Search [44]	0.661	0.57	0.536	8.53 s	**0.01**	0.397	0.198	0.64	8.79 s	**0.01**
	ECHO [2]	0.512	0.388	0.489	49.01 s	0.02	0.302	0.159	0.4	47.13 s	0.02
	RAFFLES [48]	0.719	0.661	0.613	70.75 s	0.02	0.564	0.357	0.667	72.57 s	0.03
	TraJudger	**0.868**	**0.802**	**0.773**	47.34 s	0.03	**0.714**	**0.619**	**0.859**	45.84 s	0.03
GPT-5.4	All-at-Once [44]	0.769	0.744	0.611	**3.11 s**	**0.01**	0.675	0.54	0.706	**2.36 s**	**0.01**
	Step-by-Step [44]	0.744	0.719	0.612	12.21 s	**0.01**	0.659	0.564	0.69	11.12 s	**0.01**
	Binary Search [44]	0.727	0.661	0.65	8.85 s	**0.01**	0.675	0.571	0.722	8.48 s	0.02
	ECHO [2]	0.512	0.503	0.492	44.84 s	0.05	0.468	0.318	0.575	39.01 s	0.05
	RAFFLES [48]	0.744	0.711	0.616	39.37 s	0.04	0.643	0.468	0.695	38.24 s	0.06
	TraJudger	**0.884**	**0.843**	**0.755**	38.91 s	0.05	**0.786**	**0.691**	**0.828**	32.05 s	0.06

results suggest that the structured analysis pipeline provides gains complementary to base-model capability.

Comparison among baselines. No single baseline is consistently the strongest across all settings. ECHO [2] yields the lowest accuracy in most cases, as it does not explicitly model dependencies across steps. RAFFLES [48] outperforms the Who&When evaluators [44] on D1 but loses this advantage on D2, where trajectories involve more cross-step error propagation. These results confirm that failure attribution in multi-agent trajectories demands structured representations and explicit root-cause reasoning.

5.3 Contribution of Key Modules (RQ2)

We isolate the contribution of each major component by constructing four ablated variants of *TraJudger*, each removing exactly one design element while keeping the rest unchanged:

- ***TraJudger*-NoHCG**: removes the HCG from the attribution stage; specialist findings are passed directly to the LLM alongside the raw trajectory without graph-based traversal or scoring.
- ***TraJudger*-NoSpecialist**: replaces the four specific specialists with a single generic agent that performs diagnosis in one pass over the same input scope.
- ***TraJudger*-NoConduit**: removes the conduit-step abstraction; backward tracing treats all intermediate steps uniformly without distinguishing error propagation steps from ordinary ones.
- ***TraJudger*-NoDepGraph**: removes the trajectory dependency graph from the attribution prompt; the LLM reasons over specialist findings and the linear trajectory without structural constraints.

Figure 7 summarizes the results. All four components prove essential. Removing any one of them consistently degrades performance on both D1 and D2, with drops of at least 17 percentage points in agent-level accuracy and 24 percentage points in step-level accuracy. This result suggests

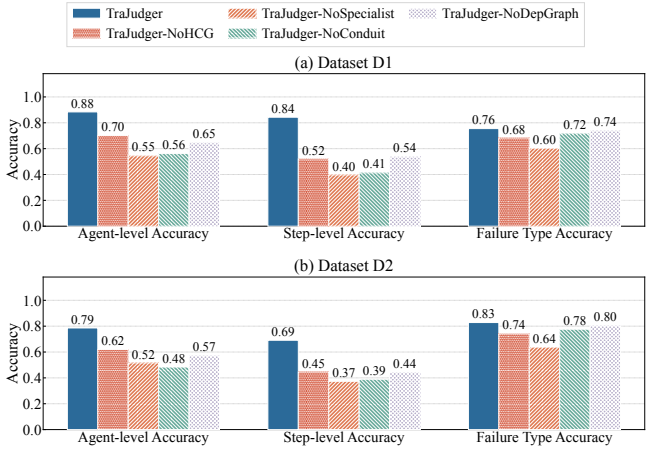


Figure 7. Ablation study results on D1 and D2. The LLM backbone we use is GPT-5.4.

that the gains of *TraJudger* do not stem from a single dominant module but rather depend on the interplay between specialized diagnosis and structured root-cause reasoning.

Among the four variants, removing the specialist module causes the largest overall degradation, particularly at the step level and in failure-type prediction. This indicates that category-specific diagnosis is the primary driver of fine-grained attribution accuracy. In contrast, removing the conduit-step abstraction leads to especially large localization errors while affecting failure-type accuracy to a much lesser extent, suggesting that conduit steps are primarily important for identifying where an error originates rather than what type of failure it represents.

The two graph-based components play complementary but secondary roles. Removing either the Hierarchical Cognitive Graph or the dependency graph consistently reduces localization accuracy, confirming that explicit structure helps the model trace trajectory dependencies more reliably than reasoning over a linearized trajectory alone. The impact of

these removals is smaller than that of removing specialists or conduit steps, yet it remains substantial across both datasets.

5.4 Computational Overhead (RQ3)

Table 5 reports the average end-to-end inference time and API cost per failure case ($/case) for *TraJudger* and five baselines on D1 and D2 under both base models. These measurements assess whether the overhead introduced by *TraJudger* is practical for production deployment.

Inference latency. Across all dataset–model settings, *TraJudger* completes diagnosis in 32.05–47.34 s per case on average, remaining within one minute and demonstrating modest end-to-end overhead for offline failure analysis. Compared with the strongest accuracy-oriented baselines, *TraJudger* consistently achieves lower diagnostic latency than both RAFFLES [48] and ECHO [2]. In contrast, the Who&When evaluators are substantially faster, but this speed advantage comes at a clear cost in attribution accuracy: as shown in Table 5, they trail *TraJudger* by a large margin on both agent- and step-level accuracy. Overall, these results suggest that *TraJudger* achieves a favorable latency–accuracy trade-off.

API cost. The API cost of *TraJudger* ranges from $0.03 to $0.06 per failure case across all settings, comparable to that of ECHO [2] and RAFFLES [48] under the same base model and only $0.02 to $0.05 higher than the cheapest baselines. To contextualize the cost at scale, diagnosing a hypothetical workload of 1,000 failure trajectories per day would incur a daily API cost of at most $60, which is negligible relative to the engineering effort required for manual root-cause analysis.

Implications for production deployment. *TraJudger* operates as an asynchronous post-mortem tool off the online serving path. Sub-minute latency is compatible with the operational workflow at *Company* $\mathcal{A}$, and the cost applies only to failed trajectories, which constitute a small fraction of total executions. Combined with the accuracy gains reported in §5.2, *TraJudger* is both accurate enough for actionable attribution and efficient enough for production deployment.

5.5 Error Propagation Chain Accuracy (RQ4)

Beyond identifying the correct root-cause agent and step, a practically useful attribution system should produce faithful error propagation chains that explain how the root-cause error propagates to the observed failure. We adopt a human evaluation protocol following [45].

We select the 102 trajectories from D1 (GPT-5.4 [30]) for which *TraJudger* correctly identifies both the root-cause agent and the root-cause step. For each trajectory, two engineers, each with at least one year of experience in agent system development and operations, independently examine the error propagation chain produced by *TraJudger* and judge whether it correctly captures the error propagation path. When the two evaluators disagree, a third senior engineer adjudicates to ensure consistency. Of the 102 cases, 83 error propagation chains are judged correct by the evaluators, yielding a propagation-path accuracy of 81.4%. The 19 incorrect cases predominantly involve implicit information transfer through shared context rather than explicit data flow, where the single-call dependency extraction occasionally omits intermediate propagation steps. These results indicate that, conditional on correct root-cause localization, *TraJudger* provides correct propagation explanations in 81.4% of cases. By tracing how the identified root cause propagates to the terminal failure, this evaluation captures the diagnostic evidence presented to OCEs beyond localization alone. It assesses the correctness of the reported propagation paths rather than the accuracy or completeness of the entire HCG.

6 Production Experience

6.1 Deployment

TraJudger has been deployed in the production environment at *Company* $\mathcal{A}$ and piloted on two multi-agent systems: a general-purpose multi-agent assistant and an operations-oriented multi-agent system used by internal operational teams.

Data collection and report delivery. When an end user marks a response as incorrect, the observability data platform of the MAS records the corresponding trajectory as a failed case indexed by task ID. Diagnosis is initiated on demand: the user locates the failed trajectory on the observability data platform and triggers *TraJudger* with a single click. OCEs can similarly retrieve a specific trajectory via the task ID reported by the user, or periodically collect all accumulated failed trajectories for batch analysis. In both cases, *TraJudger* generates a structured diagnostic report containing the root-cause agent, root-cause step, failure type, and error propagation chain.

Diagnostic accuracy and efficiency. Over a continuous deployment period exceeding one month, *TraJudger* processed 63 failure trajectories. A subsequent joint review by users and OCEs verified that *TraJudger* correctly attributed 52 of the 63 cases. Internal measurements show that *TraJudger* completes end-to-end attribution in less than one minute on average, compared to approximately 30 minutes for manual diagnosis, yielding a 30× reduction in time to attribution.

Closed-loop remediation. The on-demand diagnostic reports drove concrete remediation actions on both the user side and the system side. Among the 52 correctly attributed cases, 17 were resolved by users themselves. After triggering diagnosis and reviewing the report, these users refined their task descriptions by clarifying ambiguous requirements or providing more specific constraints. They subsequently obtained correct answers without any system-side change. On the system side, engineers used the reports to perform 5 agent prompt fixes and 9 tool schema corrections. OCEs reported that the error propagation chains were particularly

valuable for remediation: by presenting the full error propagation path with conduit steps explicitly marked, the chains helped engineers identify the true repair point rather than applying fixes at downstream symptom locations, reducing the wrong-fix rate observed prior to deployment.

Infrastructure. Because all LLM-based modules are accessed through remote APIs, *TraJudger* does not require substantial local computational infrastructure and does not affect the serving latency of the underlying multi-agent system. To support broader adoption within *Company* $\mathcal{A}$, we have packaged *TraJudger* as an internal diagnostic service that can be connected to the observability data platform of any multi-agent application with minimal integration effort, enabling both on-demand single-trajectory diagnosis and periodic batch analysis.

6.2 Case Study

We present two production cases: one where *TraJudger* correctly identified a hidden cascade that baselines missed, and one where it failed and drove a subsequent improvement.

6.2.1 Case 1: Hidden Error Cascade in a Diagnosis Task. A user reported a disk-usage spike and invoked the production diagnosis multi-agent system at *Company* $\mathcal{A}$. The system concluded that the spike was a benign fluctuation, but the user confirmed that this conclusion was incorrect. Three LLM-as-a-Judge baselines [44] all attributed the failure to the metric analysis agent, because its output was semantically consistent with the final incorrect conclusion and closest to the visible symptom.

TraJudger traced the error propagation chain further and identified the true root cause at an earlier log analysis stage: the log analysis agent incorrectly clustered abnormal log event templates into a normal cluster. This erroneous result propagated as a trusted context to subsequent agents, biasing the metric analysis agent toward the benign-fluctuation explanation. Operations engineers confirmed this attribution.

6.2.2 Case 2: Scalability Limitation on Long Trajectories. A code review task produced a 191-step trajectory. During HCG construction, as the accumulated context approached the context-window limit of the LLM, parsing quality degraded on later steps and some long-range dependencies were omitted, causing attribution to terminate prematurely.

This case motivated a chunked processing strategy: long trajectories are partitioned along semantic boundaries, local dependency parsing is performed within each chunk, and cross-chunk dependencies are reconciled through overlapping regions and structured step summaries.

6.3 Operational Lessons

Running *TraJudger* in production for over one month has yielded practical insights that were not apparent from benchmark evaluation alone.

Lesson #1: Explanatory structure is as important as attribution accuracy. In practice, engineers need more than a root-cause label; they also require evidence showing how an error propagates and which upstream location should be inspected first. During deployment, OCEs consistently identified error propagation chains and conduit-step annotations as particularly actionable components of the diagnostic report. Using these reports, the average attribution time decreased from approximately 30 minutes under manual diagnosis to under one minute. OCEs further reported that propagation evidence helped distinguish upstream repair points from downstream symptoms. These observations suggest that failure-attribution outputs should be designed as engineer-facing diagnostic interfaces rather than presented solely as classification results.

Lesson #2: The failure taxonomy should remain extensible. Production multi-agent systems evolve continuously, and so do their failure modes. As new agents, tools, and orchestration patterns are introduced, previously unseen failures inevitably emerge. We therefore treat the taxonomy as a living artifact: cases with low-confidence attribution are routed for periodic expert review, and the resulting feedback is used to extend the taxonomy and refine specialist prompts. In practice, this review loop proved necessary for keeping the diagnosis pipeline aligned with the system under observation. During the first month of deployment, this process surfaced 12 low-confidence cases. Review of these cases by OCEs led to the addition of two new failure categories to the taxonomy and three rounds of specialist prompt refinement.

Lesson #3: Diagnosability should be designed into the MAS runtime. Trajectory quality dominates diagnosis quality. Early applications at *Company* $\mathcal{A}$ were instrumented with generic log-style capture, and a non-trivial fraction of runs went unrecorded because the collection path was ad hoc. Under this regime, even a strong base model could not reliably recover error propagation chains, because the evidence it needed was never recorded. As the observability stack matured, trajectories are now collected in a uniform JSON schema with structured per-step metadata: input/output token counts, execution latency, tool-call arguments and return values, and explicit references to upstream steps. We define this schema as a contract between the multi-agent runtime and *TraJudger*, and adherence is part of the onboarding checklist for new applications. Structured logging may also benefit other attribution methods. However, invocation references alone may not capture semantic dependencies; structured logging thus complements, rather than replaces, semantic dependency analysis.

Lesson #4: In offline attribution, quality matters more than raw latency. The structured pipeline of *TraJudger* inevitably issues more LLM calls than a one-shot baseline. In our production setting, attribution runs as an asynchronous post-mortem task off the user-facing critical path, so per-case latency on the order of tens of seconds is well within

the operational envelope. The failure mode we guard against is the opposite scenario: a faster but less accurate attribution misdirects engineers to the wrong agent, incurring additional investigation rounds. The cost gap between a structured pipeline and a one-shot baseline can be narrowed through engineering rather than by sacrificing structure: the HCG is cached across specialists, the four specialists are dispatched in parallel, and short trajectories with tool errors are routed to a lightweight path that bypasses inference.

6.4 Limitations

Limitation #1: Context-window pressure on extremely long trajectories. Several stages consume the full trajectory in a single LLM call. For pathologically long trajectories spanning hundreds of turns, our production deployment employs a sliding-window strategy that segments the trajectory along semantic boundaries and reconciles cross-window dependencies through overlapping regions.

Limitation #2: Dependence on LLM reasoning correctness. HCG construction and dependency-edge verification rely on the semantic judgment of the LLM. The structural graph constraint and four-specialist redundancy partially mitigate this risk, but systematic LLM biases that survive both safeguards can still affect attribution. Introducing lightweight formal verification on top of the structural constraint layer is a direction we are actively pursuing.

Limitation #3: Single-root-cause evaluation protocol. Our evaluation assumes a single dominant root cause per trajectory, although independent failures may co-occur in practice. This assumption does not constrain *TraJudger*, which can identify and rank multiple faulty ancestors across observed dependency branches. Evaluating multi-root attribution, however, requires datasets with explicit multi-root annotations, which we leave to future work.

Limitation #4: Efficiency under growing scale and latency-sensitive deployments. *TraJudger* currently targets offline post-hoc analysis, where tens-of-seconds latency is acceptable. Larger trajectory volumes and latency-sensitive settings impose stricter efficiency requirements. Promising directions include native reimplementation of performance-critical components and efficient processing of compressed trajectories, which we leave to future work.

7 Related Work

Research on failure attribution for LLM-powered multi-agent systems spans four categories.

Pattern-based methods. CORRECT [41], FAMAS [12], and Sun et al. [32] localize failures by mining recurrent signals across execution trajectories. These methods rely on statistical regularity and are sensitive to the non-determinism inherent in LLM-driven execution paths.

Intervention-based methods. Dover [22] and AgentDebug [49] validate causal hypotheses through targeted re-execution from specific states. These methods are inherently intrusive and incur a high computational cost that scales poorly to production workloads.

LLM reasoning-based methods. These methods apply foundation models to infer root causes through structured prompting, including multi-strategy evaluation [44], hierarchical context with consistency voting [2], Shapley-based responsibility assignment [21], specialized evaluator frameworks [48], oracle-guided backtracking [34], and constraint-based step-wise validation [3]. However, these methods remain susceptible to hallucination and lack structural grounding for root-cause reasoning.

Model fine-tuning methods. AgenTracer [42], Aegis [18], VerifyMAS [27], and GraphTracer [43] train task-specific failure trackers via supervised or reinforcement learning. These methods achieve strong in-distribution performance but rely on task-specific training sets that limit generalization to unseen architectures.

TraJudger falls within the LLM reasoning-based category but addresses two limitations of existing methods. First, a conduit-aware dependency graph with backward tracing provides structural grounding against hidden cascades. Second, a structure-constrained architecture restricts LLM inference to verified dependency paths, thereby reducing hallucinated attribution.

8 Conclusion

We present *TraJudger*, a structure-guided failure attribution system for LLM-powered multi-agent systems deployed in production at *Company A*. *TraJudger* constructs explicit dependency structures from unstructured trajectories through three decoupled stages: an HCG for structured representation, category-specific specialist agents for typed failure detection, and structure-constrained backward tracing that constrains root-cause reasoning to verified dependency paths. Evaluated on production and benchmark data, *TraJudger* consistently achieves a relative improvement of at least 15% and 13.3% over state-of-the-art methods at the agent and step levels, respectively, and delivers a 30× reduction in time-to-attribution in production deployment.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62272249, 62302244), the Fundamental Research Funds for the Central Universities (XXX-63253249), and the Tianjin Key Research and Development Program (Grant No. 25YFYFFG00690).

References

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large

Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Press, Los Alamitos, CA, 1737–1749. doi:10.1109/ICSE48619.2023.00149

[2] Adi Banerjee, Anirudh Nair, and Tarik Borogovac. 2025. Where Did It All Go Wrong? A Hierarchical Look into Multi-Agent Error Attribution. In *NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling*. Curran Associates, Inc., New York, NY, USA. https://openreview.net/forum?id=0MyUdq7wLe

[3] Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. 2026. AgentRx: Diagnosing AI Agent Failures from Execution Trajectories. arXiv:2602.02475 [cs.AI] https://arxiv.org/abs/2602.02475

[4] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA, 2846–2857. doi:10.1109/ASE63991.2025.00234

[5] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei A Zaharia, Joseph Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail?. In *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (Eds.), Vol. 38. Curran Associates, Inc., New York, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2025/file/b1041e52d3be19f0a9bc491657488e4a-Paper-Datasets_and_Benchmarks_Track.pdf

[6] Hongyang Chen, Benran Wang, Guangba Yu, Zilong He, Pengfei Chen, Chen Sun, and Zibin Zheng. 2025. NetScope: Fault Localization in Programmable Networking Systems With Low-Cost In-Band Network Telemetry and In-Network Detection. *IEEE Transactions on Networking* 33, 6 (2025), 3487–3503. doi:10.1109/TON.2025.3590034

[7] 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. arXiv:2604.22708 [cs.MA] https://arxiv.org/abs/2604.22708

[8] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 674–688. doi:10.1145/3627703.3629553

[9] Zhendong Chu, Shen Wang, Jian Xie, Tinghui Zhu, Yibo Yan, Jingheng Ye, Aoxiao Zhong, Xuming Hu, Jing Liang, Philip S. Yu, and Qingsong Wen. 2025. LLM Agents for Education: Advances and Applications. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 13782–13810. doi:10.18653/v1/2025.findings-emnlp.743

[10] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. TRAIL: Trace Reasoning and Agentic Issue Localization. arXiv:2505.08638 [cs.AI] https://arxiv.org/abs/2505.08638

[11] Gen Dong, Yu Hua, Yongle Zhang, Zhangyu Chen, and Menglei Chen. 2025. Understanding and Detecting Fail-Slow Hardware Failure Bugs in Cloud Systems. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 1127–1142. https://www.usenix.org/conference/atc25/presentation/dong

[12] 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. arXiv:2509.13782 [cs.SE] https://arxiv.org/abs/2509.13782

[13] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. 2008. Exploring network structure, dynamics, and function using NetworkX. Los Alamos National Laboratory (LANL). https://www.osti.gov/biblio/960616

[14] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*, Vol. 2024. http://OpenReview.net, Ithaca, NY, 23247–23275. https://openreview.net/forum?id=VtmBAGCN7o

[15] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, Changhong Liu, Tao Xie, Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan. 2025. DEEPSERVE: Serverless Large Language Model Serving at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 57–72. https://www.usenix.org/conference/atc25/presentation/hu-junhao

[16] Xiaosong Huang, Hongyi Liu, Yifan Wu, Lingzhe Zhang, Tong Jia, Ying Li, and Zhonghai Wu. 2026. UDA-RCL: Unsupervised Domain Adaptation for Microservice Root Cause Localization Utilizing Multimodal Data. *IEEE Transactions on Services Computing* 19, 2 (2026), 1661–1672. doi:10.1109/TSC.2025.3646329

[17] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2020. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1410–1420. doi:10.1145/3368089.3417054

[18] Fanqi Kong, Ruijie Zhang, Huaxiao Yin, Guibin Zhang, Xiaofei Zhang, Ziang Chen, Zhaowei Zhang, Xiaoyuan Zhang, Song-Chun Zhu, and Xue Feng. 2026. Aegis: Automated Error Generation and Attribution for Multi-Agent Systems. In *The Fourteenth International Conference on Learning Representations*, Vol. 2026. http://OpenReview.net, Ithaca, NY, 52994–53028. https://openreview.net/forum?id=zqcYoxXiN3

[19] Yang Liu, Hongjiang Feng, Junsong Pu, and Zhuangbin Chen. 2026. MASPrism: Lightweight Failure Attribution for Multi-Agent Systems Using Prefill-Stage Signals. arXiv:2605.07509 [cs.SE] https://arxiv.org/abs/2605.07509

[20] Yu Luo, Rongchen Gao, Lu Teng, Xidao Wen, Jiamin Jiang, Qingliang Zhang, Yongqian Sun, Shenglin Zhang, Jiasong Feng, Tong Liu, Wenjie Zhang, and Dan Pei. 2026. Graph of States: Solving Abductive Tasks with Large Language Models. In *Forty-third International Conference on Machine Learning*.

[21] Guoqing Ma, Jia Zhu, Hanghui Guo, Weijie Shi, Jiawei Shen, Jingjiang Liu, and Yidan Liang. 2025. Automatic Failure Attribution and Critical Step Prediction Method for Multi-Agent Systems Based on Causal Inference. arXiv:2509.08682 [cs.AI] https://arxiv.org/abs/2509.08682

[22] Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2026. DoVer: Intervention-Driven Auto Debugging for LLM Multi-Agent Systems. In *The Fourteenth International Conference on Learning Representations*, Vol. 2026. http://OpenReview.net, Ithaca, NY, 28059–28087. https://openreview.net/forum?id=mrEK16Jy6h

[23] Minghua Ma, Shenglin Zhang, Junjie Chen, Jim Xu, Haozhe Li, Yongliang Lin, Xiaohui Nie, Bo Zhou, Yong Wang, and Dan Pei. 2021. Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Carlsbad, CA, 413–426. https://www.usenix.org/conference/atc21/presentation/ma

[24] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: a benchmark for General AI Assistants. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. http://OpenReview.net, Ithaca, NY, 9025–9049. https://proceedings.iclr.cc/paper_files/paper/2024/file/25ae35b5b1738d80f1f03a8713e405ec-Paper-Conference.pdf

[25] Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. 2025. Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis. In *Companion Proceedings of the ACM on Web Conference 2025* (Sydney NSW, Australia) *(WWW '25)*. Association for Computing Machinery, New York, NY, USA, 422–431. doi:10.1145/3701716.3715225

[26] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15174–15186. doi:10.18653/v1/2024.acl-long.810

[27] Hezhe Qiao, Hanghang Tong, Ee-Peng Lim, Bing Liu, and Guansong Pang. 2026. VerifyMAS: Hypothesis Verification for Failure Attribution in LLM Multi-Agent Systems. arXiv:2605.17467 [cs.CL] https://arxiv.org/abs/2605.17467

[28] Davide Rovelli, Pavel Chuprikov, Philipp Berdesinski, Ali Pahlevan, Patrick Jahnke, and Patrick Eugster. 2025. FiDe: Reliable and Fast Crash Failure Detection to Boost Datacenter Coordination. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 765–788. https://www.usenix.org/conference/atc25/presentation/rovelli

[29] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. 2025. Agent Laboratory: Using LLM Agents as Research Assistants. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 5977–6043. doi:10.18653/v1/2025.findings-emnlp.320

[30] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. 2026. OpenAI GPT-5 System Card. arXiv:2601.03267 [cs.CL] https://arxiv.org/abs/2601.03267

[31] Kevin Song, Anand Jayarajan, Yaoyao Ding, Qidong Su, Zhanda Zhu, Sihang Liu, and Gennady Pekhimenko. 2025. Aegis: Taxonomy and Optimizations for Overcoming Agent-Environment Failures in LLM Agents. arXiv:2508.19504 [cs.MA] https://arxiv.org/abs/2508.19504

[32] Kai Sun, Wenqiang Li, Bo Dong, Yuxin Lin, Jingyao Zhang, and Bin Shi. 2026. Scope Delineation Before Localization: A Two-Stage Framework for Enhancing Failure Attribution in Multi-Agent Systems. *Proceedings of the AAAI Conference on Artificial Intelligence* 40, 39 (Mar. 2026), 33108–33116. https://ojs.aaai.org/index.php/AAAI/article/view/40594

[33] Gou Tan, Zilong He, Min Li, Haiyu Huang, Yilun Wang, Pengfei Chen, Giuliano Casale, and Chuanfu Zhang. 2026. LLMRCA: Multilevel Root Cause Analysis for LLM Applications Using Multimodal Observability Data. *ACM Trans. Softw. Eng. Methodol.* (April 2026). Just Accepted. doi:10.1145/3806200

[34] 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. arXiv:2602.23701 [cs.AI] https://arxiv.org/abs/2602.23701

[35] Zhe Wang, Huanwu Hu, Linghe Kong, Xinlei Kang, Qiao Xiang, Jingxuan Li, Yang Lu, Zhuo Song, Peihao Yang, Jiejian Wu, Yong Yang, Tao Ma, Zheng Liu, Xianlong Zeng, Dennis Cai, and Guihai Chen. 2024. Diagnosing Application-network Anomalies for Millions of IPs in Production Clouds. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 885–899. https://www.usenix.org/conference/atc24/presentation/wang-zhe

[36] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. 2025. Intent-Driven Network Management with Multi-Agent LLMs: The Confucius Framework. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 347–362. doi:10.1145/3718958.3750537

[37] Yutao Wu, Xiao Liu, Yunhao Feng, Jiale Ding, and Xingjun Ma. 2026. PaperAsk: A Benchmark for Reliability Evaluation of LLMs in Paper Search and Reading. In *Proceedings of the ACM Web Conference 2026* (United Arab Emirates) *(WWW '26)*. Association for Computing Machinery, New York, NY, USA, 2330–2338. doi:10.1145/3774904.3792597

[38] Ziluo Xue, Yanjie Zhao, Shenao Wang, Kai Chen, and Haoyu Wang. 2025. A Characterization Study of Bugs in LLM Agent Workflow Orchestration Frameworks. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA, 3369–3380. doi:10.1109/ASE63991.2025.00278

[39] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[40] Bo Yang, Huanwu Hu, Yifan Li, Yunguang Li, Xiangyu Tang, Bingchuan Tian, Gongwei Wu, Jianfeng Xu, Xumiao Zhang, Feng Chen, Cheng Wang, Ennan Zhai, Yuhong Liao, Dennis Cai, and Tao Lin. 2025. SkyNet: Analyzing Alert Flooding from Severe Network Failures in Large Cloud Infrastructures. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 512–526. doi:10.1145/3718958.3750536

[41] Yifan Yu, Moyan Li, Shaoyuan Xu, Jinmiao Fu, Xinhai Hou, Fan Lai, and Bryan Wang. 2025. CORRECT: COndensed eRror RECognition via knowledge Transfer in multi-agent systems. arXiv:2509.24088 [cs.MA] https://arxiv.org/abs/2509.24088

[42] Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng YAN. 2026. AgenTracer: Who Is Inducing Failure in the LLM Agentic Systems?. In *The Fourteenth International Conference on Learning Representations*, Vol. 2026. http://OpenReview.net, Ithaca, NY, 11377–11399. https://openreview.net/forum?id=l05DseqvuD

[43] Heng Zhang, Yuling Shi, Xiaodong Gu, Haochen You, Zijian Zhang, Lubin Gan, Yilei Yuan, and Jin Huang. 2025. GraphTracer: Graph-Guided Failure Tracing in LLM Agents for Robust Multi-Turn Deep Search. arXiv:2510.10581 [cs.GR] https://arxiv.org/abs/2510.10581

[44] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) *(ICML'25)*. JMLR.org, Article 3076, 17 pages.

[45] Shenglin Zhang, Yongxin Zhao, Xiao Xiong, Yongqian Sun, Xiaohui Nie, Jiacheng Zhang, Fenglai Wang, Xian Zheng, Yuzhi Zhang, and Dan Pei. 2024. Illuminating the Gray Zone: Non-intrusive Gray Failure Localization in Server Operating Systems. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 126–137. doi:10.1145/3663529.3663834

[46] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[47] Zhenyu Zhong, Ruowei Fu, Minghua Ma, Shenglin Zhang, Yongqian Sun, Chetan Bansal, and Dan Pei. 2026. LLM-Enhanced Failure Localization in Microservices: Integrating Multi-Modal Data and Expert Interpretation. *IEEE Transactions on Services Computing* 19, 3 (2026), 2275–2288. doi:10.1109/TSC.2026.3676262

[48] Chenyang Zhu, Spencer Hong, Jingyu Wu, Kushal Chawla, Yuhui Tang, Youbing Yin, Nathan Wolfe, Erin Babinsky, and Daben Liu. 2026. RAFFLES: Reasoning-based Attribution of Faults for LLM Systems. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vera Demberg, Kentaro Inui, and Lluís Marquez (Eds.). Association for Computational Linguistics, Rabat, Morocco, 7659–7688. doi:10.18653/v1/2026.eacl-long.359

[49] 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. arXiv:2509.25370 [cs.AI] https://arxiv.org/abs/2509.25370

[50] Taiyu Zhu, Yifan Wu, Weilin Jin, Ying Li, and Gang Huang. 2026. StepFinder: A Temporal Semantic Framework for Failure Attribution in Multi-Agent Systems. arXiv:2606.03467 [cs.AI] https://arxiv.org/abs/2606.03467