INTERACTIVE TRAJECTORY ENVIRONMENTS FOR LONG-HORIZON AGENT DIAGNOSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diagnosing agentic systems requires identifying where an execution went wrong, which becomes increasingly challenging for long-horizon executions as the diagnostic space grows and relevant evidence becomes dispersed across distant but dependent steps. Existing approaches primarily focus on designing the diagnosis agent or procedure, while treating the execution trajectory as passive context. We instead formulate diagnosis as **agent–environment interaction**, jointly designing the diagnosis agent and the environment through which it investigates an execution. We introduce TIDE (**T**rajectory-to-**I**nteractive **D**iagnosis **E**nvironment), which transforms a failed execution trajectory into an interactive environment for diagnosis. Specifically, we construct a dependency graph over execution steps and hierarchically organize it into execution processes; together, these structures define an environment in which the agent can navigate across levels of abstraction and inspect dependency-related evidence. An **Agent-as-a-Judge** adaptively interacts with this environment to explore the diagnostic space, acquire relevant evidence, and localize the erroneous step. Experiments on two benchmarks show that TIDE consistently improves error localization across agentic systems and judge models, reducing mean localization distance by up to 55% on long-horizon trajectories. Further analyses show that dependency-aware exploration is particularly beneficial when diagnostically related steps are distant in execution order.

1 INTRODUCTION

Large language models (LLMs) increasingly power agentic systems that reason, use tools, interact with external environments, and coordinate with other agents to complete complex tasks (Yao et al., 2023; Wang et al., 2024; Yu et al., 2025). Evaluating their final outcomes establishes whether a task succeeded, but provides limited insight into where an execution went wrong or which agent was responsible. Agentic system diagnosis addresses this gap by examining execution trajectories to localize the step responsible for an execution failure (Cemri et al., 2025; Zhang et al., 2025b; Barke et al., 2026). Such diagnoses can guide targeted system improvements and provide feedback for self-evolving agents (Fang et al., 2025; ang Gao et al., 2026; Du et al., 2026). As agentic systems undertake increasingly long-running tasks, diagnosis must account for errors and their consequences across extended sequences of actions (Erdogan et al., 2025; Dong et al., 2026; Hu et al., 2025).

Diagnosing long-horizon agentic systems presents two closely related challenges: a growing search space and dispersed diagnostic evidence. Longer trajectories contain more candidate error locations, while dependencies between steps make it difficult to assess an action in isolation (Zhuge et al., 2024). An apparently reasonable action may rely on an incorrect result produced much earlier, while a visible failure may emerge only after that result propagates through subsequent steps. Consequently, execution order alone provides an incomplete guide to diagnosis. A judge must determine both where to investigate and which related steps to inspect to distinguish an error from its downstream manifestations.

Existing approaches use LLM judges to assess complete trajectories or individual steps (Zhang et al., 2025b; 2026), while more recent methods employ agents to structure and conduct the investigation (Banerjee et al., 2025; Wang et al., 2026). These advances primarily focus on the *diagnosis agent or procedure*: how the judge decomposes the task, examines evidence, and produces an attribution. The execution trajectory, however, largely remains the context over which this procedure operates.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

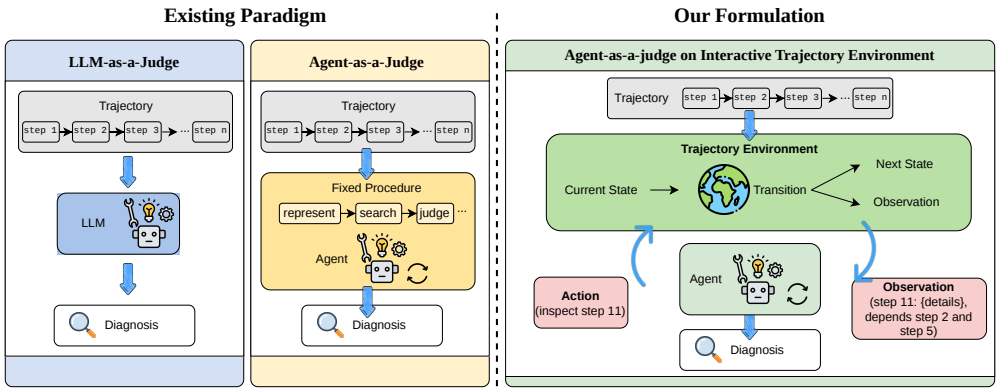


Figure 1: Comparison of diagnosis paradigms. Existing approaches primarily design how a judge analyzes an execution trajectory, whereas we jointly design the diagnosis agent and the trajectory environment through which it investigates the execution.

We argue that effective diagnosis requires designing not only the agent, but also the *environment through which the agent investigates the execution*. This motivates our central question: *How can we jointly design the diagnosis agent and its trajectory environment for long-horizon agentic system diagnosis?*

We formulate diagnosis as **agent–environment interaction**. Given a recorded execution, we construct a *trajectory environment* that organizes execution evidence and determines how it can be accessed through actions and observations (see Figure 1). Complementarily, a diagnosis agent adaptively determines where to navigate and what evidence to inspect based on its observations. The environment therefore determines how the diagnostic space can be explored, while the agent determines how the investigation proceeds. Importantly, this interaction concerns post-hoc investigation of a recorded execution rather than modification or re-execution of the underlying task.

Building on this formulation, we introduce TIDE (**T**rajectory-to-**I**nteractive **D**iagnosis **E**nvironment), a framework that transforms a failed execution trajectory into an interactive environment through which a diagnosis agent investigates the source of failure. To construct this environment, we first build a dependency graph that captures information dependencies across execution steps, exposing diagnostically relevant relations beyond temporal order. We further organize these dependencies into a hierarchy of progressively coarser execution processes, supporting investigation at different levels of granularity. Together, these structures provide the basis for dependency-guided, coarse-to-fine interaction with the trajectory. Complementarily, we develop an **Agent-as-a-Judge** that adaptively interacts with the environment, deciding where to navigate and what evidence to inspect until it can localize the erroneous step and responsible agent. Together, the trajectory environment and diagnosis agent transform diagnosis from judgment over passive trajectory context into adaptive investigation through agent–environment interaction.

Our contributions are: (1) We formulate long-horizon agentic system diagnosis as **agent–environment interaction**, shifting the design focus from the diagnosis agent alone to the joint design of the agent and its trajectory environment. (2) We introduce TIDE, which transforms execution trajectories into **interactive trajectory environments** that support dependency-guided and hierarchical, coarse-to-fine investigation. (3) We develop an **Agent-as-a-Judge** that adaptively interacts with this environment to localize erroneous steps and responsible agents. (4) Experiments on two benchmarks demonstrate that TIDE consistently improves error localization across agentic systems and judge models, reducing mean localization distance by up to 55% on long-horizon trajectories.

2 Related Work

Agent System Diagnosis. Outcome-level evaluation reveals little about why an agentic system fails, motivating growing work on diagnosis from execution trajectories. MAST categorizes failure modes in multi-agent systems (Cemri et al., 2025), AgentDiagnose evaluates trajectory-level competencies (Ou et al., 2025), and AgentRx localizes critical failure steps through constraint-based

validation (Barke et al., 2026). More specifically, Who&When formalizes failure attribution as identifying the responsible agent and decisive error step (Zhang et al., 2025b), followed by approaches based on specialized training (Zhang et al., 2026), counterfactual reasoning (West et al., 2025), and iterative evaluator teams (Zhu et al., 2026; Li et al., 2026). Recent work further exploits structured trajectories for diagnosis, including hierarchical or causal representations (Banerjee et al., 2025; Wang et al., 2026), behavioral abstractions, and dependency-guided search. TraceElephant additionally demonstrates the importance of full execution observability for reliable failure attribution (Chen et al., 2026). Our work instead formulates diagnosis as **agent–environment interaction**. We construct an interactive trajectory environment that supports dependency-guided, coarse-to-fine exploration, while a separate diagnosis agent adaptively determines what to inspect and how to navigate the execution.

Agent-as-a-Judge. LLM-as-a-Judge enables scalable evaluation by asking language models to assess outputs from fixed context (Liu et al., 2023; Zheng et al., 2023). Recent work extends this paradigm toward **Agent-as-a-Judge**, where evaluators actively acquire and verify evidence during evaluation. Zhuge et al. evaluate intermediate artifacts and requirement satisfaction throughout agent execution (Zhuge et al., 2025). AJ-Bench studies judges that interact with environments to verify execution states and processes (Shi et al., 2026), while Online Agent-as-a-Judge allows evaluators to interact with target agents to elicit otherwise unobserved behaviors (Ryu et al., 2026). Our work extends Agent-as-a-Judge to post-hoc root-cause diagnosis through the **joint design of the diagnosis agent and its environment**. We transform a failed execution into a trajectory environment supporting dependency-guided, coarse-to-fine exploration, through which the diagnosis agent adaptively investigates sparse errors without rerunning or modifying the target system.

3 PRELIMINARIES

3.1 PROBLEM FORMULATION

We consider an agentic system consisting of a set of agents $\mathcal{A} = a_1, \ldots, a_M$. Given a task x, the system produces an execution trajectory $\tau = (s_1, s_2, \ldots, s_T)$, where s_t denotes the t-th execution step and $a(s_t) \in \mathcal{A}$ denotes the agent responsible for that step. Each step records intermediate information produced during execution, such as reasoning, actions, tool interactions, observations, and responses. The trajectory τ therefore captures the execution process through which the system progresses from the task x toward its final outcome y.

Given a failed execution trajectory τ, **agentic system diagnosis** aims to localize the execution step responsible for the observed failure (Zhang et al., 2025b; Barke et al., 2026). We denote this root-cause step as $s_e \in \tau$ and formulate diagnosis as

$$\mathcal{D}(x, \tau, y) \to s_e, \tag{1}$$

where $\mathcal{D}$ denotes the diagnosis procedure. Since each execution step is associated with its responsible agent, localizing s_e also determines the responsible agent $a(s_e)$. We therefore focus on step-level root-cause localization, with agent-level attribution following directly from the predicted step.

3.2 DIAGNOSIS AS AGENT–ENVIRONMENT INTERACTION

Rather than treating diagnosis as a fixed procedure over τ, we formulate it as interaction between a **diagnosis agent** π_D and a **trajectory environment** $\mathcal{E}_\tau$ constructed from the failed execution. The environment defines what execution information can be accessed and how the investigation can navigate through it, while the agent determines what to investigate based on the evidence observed.

Let $\mathcal{U}$ denote the set of actions supported by the environment and z_k the environment state at interaction round k, representing the current focus and accessible scope of the investigation. Based on its interaction history $h_k = (o_0, u_0, \ldots, o_k)$, the diagnosis agent selects $u_k \sim \pi_D(\cdot \mid h_k)$. The environment then executes u_k from the current state according to its transition function $\mathcal{T}$ and returns the corresponding observation through $\mathcal{O}$:

$$z_{k+1} = \mathcal{T}(z_k, u_k), \qquad o_{k+1} = \mathcal{O}(z_{k+1}). \tag{2}$$

Through successive interactions, the agent selectively investigates the execution until terminating with a predicted root-cause step $\hat{s}_e$. Under this formulation, diagnosis requires jointly constructing

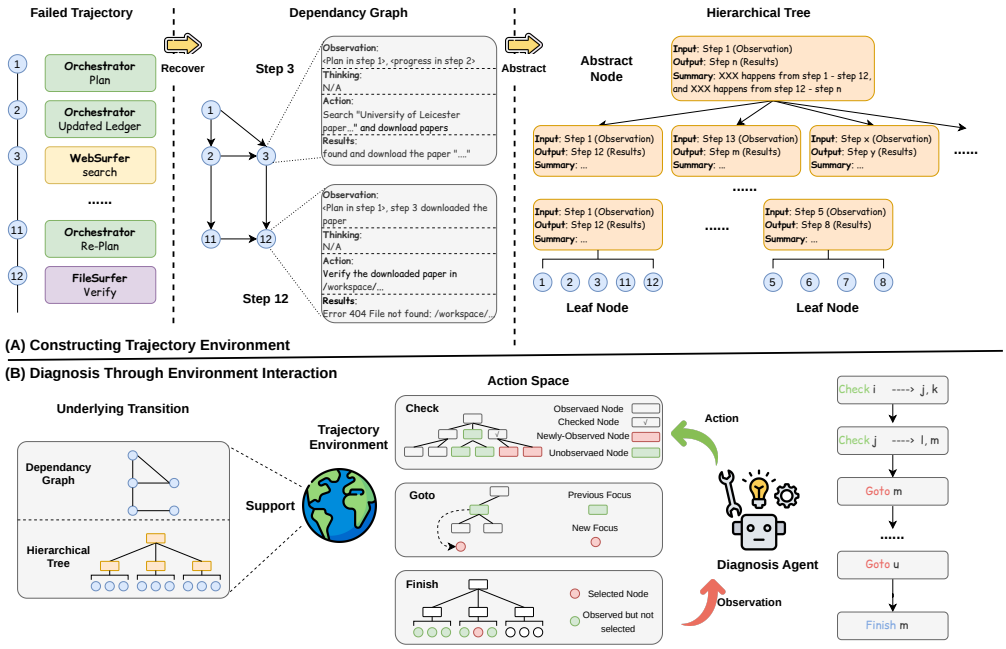


Figure 2: Overview of TIDE. (A) Trajectory environment constructed from inter-step dependencies and their hierarchical organization. (B) Agent-as-a-Judge adaptively investigates the environment through CHECK, GOTO, and FINISH to localize the erroneous step.

an environment that supports structured exploration of τ and an agent that adaptively determines how to explore it.

4 TIDE: AGENT–ENVIRONMENT DIAGNOSIS

Following Section 3.2, we instantiate our agent–environment formulation as TIDE, which jointly constructs the trajectory environment $\mathcal{E}_\tau = (\mathcal{Z}, \mathcal{U}, \mathcal{T}, \mathcal{O})$ and the diagnosis agent π_D (see Figure 2). We construct $\mathcal{E}_\tau$ by recovering dependencies among execution steps and hierarchically organizing them, providing a structured space through which the failed execution can be progressively investigated. We then define an interaction interface $(\mathcal{U}, \mathcal{T}, \mathcal{O})$ that controls how π_D navigates this space and acquires execution evidence until localizing the root-cause step.

4.1 CONSTRUCTING THE TRAJECTORY ENVIRONMENT

A raw trajectory exposes execution steps primarily in temporal order, without explicitly representing how information propagates across steps or how related behaviours form higher-level execution processes. We therefore construct the structural basis of $\mathcal{E}_\tau$ in two stages. First, we recover information dependencies among execution steps to construct a dependency graph G. Second, we recursively abstract G into a hierarchy H of execution processes at different granularities. The hierarchy provides the primary structure for navigating the execution, while the dependency graph is retained to support fine-grained verification when the investigation reaches individual behaviours.

4.1.1 DEPENDENCY GRAPH CONSTRUCTION

Given the trajectory $\tau = (s_1, \ldots, s_T)$, we construct a directed dependency graph $G = (V, E)$ whose nodes represent agent behaviours and whose edges capture information dependencies among them.

Behaviour Nodes. Each trajectory step s_i is converted into a structured OTAR node $v_i \in V$ following (Banerjee et al., 2025; Wang et al., 2026): $v_i = \langle O_i, T_i, A_i, R_i \rangle$, where O_i, T_i, A_i, and R_i denote the *Observation*, *Thought*, *Action*, and *Result* associated with the behaviour, respectively.

This representation separates the information received by an agent from its reasoning, action, and resulting output, providing a common unit for tracing information flow through the execution.

Dependency Edges. A directed edge $v_i \to v_j$ indicates that information produced by behaviour v_i contributes to the observation of behaviour v_j. Specifically,

$$(v_i, v_j) \in E \quad \text{if} \quad R_i \rightsquigarrow O_j, \tag{3}$$

where $\rightsquigarrow$ denotes that information contained in R_i contributes to O_j. These edges make explicit which earlier behaviours provide information used by later behaviours, including dependencies between temporally distant steps.

Incremental Construction. We construct G incrementally following the temporal order of the trajectory. For each new node v_j, we identify preceding behaviours whose results contribute to its observation O_j and connect them to v_j. We enforce **temporal consistency**, such that every dependency $v_i \to v_j$ satisfies $i < j$, and assume **result-mediated dependency**, whereby information produced by a behaviour becomes available to subsequent behaviours through its result R_i or, where applicable, its action–result pair. Repeating this procedure over the trajectory yields a directed acyclic graph that makes both local and long-range information dependencies explicit.

The resulting graph forms the fine-grained dependency structure underlying $\mathcal{E}_\tau$. It is retained throughout diagnosis and exposed locally when the investigation reaches an individual behaviour, allowing the agent to inspect that behaviour in the context of its information dependencies.

4.1.2 Hierarchical Environment Construction

Although G exposes information dependencies, directly investigating the complete graph requires reasoning over all fine-grained behaviours simultaneously. We therefore organize G into a hierarchy H of progressively coarser **causal processes**. The hierarchy provides the primary navigation structure of $\mathcal{E}_\tau$, allowing the investigation to move from high-level execution processes toward individual behaviours.

Hierarchical Nodes. The hierarchy contains **leaf nodes** and **abstract nodes**. Each leaf node corresponds to a behaviour $v_i \in V$ and preserves its OTAR representation. An abstract node h represents a set of dependency-related behaviours $\phi(h) \subseteq V$ as a higher-level execution process. We represent it as

$$h = \langle I(h), O(h), C(h) \rangle, \tag{4}$$

where $I(h)$ and $O(h)$ denote the information entering and leaving the process, respectively, and $C(h)$ is a concise summary of its child processes. The input–output pair characterizes the external behaviour of the process, while $C(h)$ provides a compact description of its internal decomposition. Together, these fields allow the diagnosis agent to reason about an abstract process before deciding whether to investigate its children in greater detail.

Hierarchical Relations. An edge (u, v) in H represents an abstraction relation rather than an execution dependency. Specifically, $\phi(v) \subset \phi(u)$, meaning that the process represented by v is contained within the coarser process represented by u. Descending through such an edge therefore exposes a finer-grained view of the corresponding execution process.

Self-Contained Causal Processes. To determine which graph nodes can be abstracted together, we define a **self-contained causal process**. For a set $S \subseteq V$, the induced subgraph $G[S]$ forms such a process if: (i) $G[S]$ is connected; (ii) it contains a unique beginning node $b(S)$; and (iii) for every $v \in S \setminus \{b(S)\}$, the observation dependencies required by v are satisfied within S. Except for information entering through $b(S)$, the process is therefore internally dependency-complete and can be represented by its external input and output.

Recursive Abstraction. We construct H bottom-up over G. Starting from the terminal behaviour associated with the observed outcome, we traverse dependencies backward and collect connected behaviours. Once the collected nodes form a self-contained causal process, they are grouped into an abstract node. Its input and output are inherited from the beginning and terminal behaviours of

the process, respectively, while a concise summary is generated from its child nodes to describe the process they collectively perform. We recursively apply the same criterion to the resulting processes, merging them into progressively coarser abstract nodes until no further abstraction is possible.

The resulting hierarchy organizes the execution at multiple levels of granularity, while the underlying dependency graph preserves relationships among individual behaviours. They provide the structural basis over which the environment controls navigation and evidence exposure during diagnosis.

4.2 Diagnosis through Environment Interaction

Given the constructed trajectory environment, we instantiate its interaction components $(\mathcal{U}, \mathcal{T}, \mathcal{O})$ to control how the diagnosis agent π_D explores the execution. The complete trajectory and hierarchy are not exposed to the agent upfront. Instead, the environment maintains a diagnostic state z_k that records the current focus and the nodes exposed so far, and progressively reveals execution information through interaction.

Environment Transition and Observation. The transition function $\mathcal{T}$ determines how an agent action updates the diagnostic state, while the observation function $\mathcal{O}$ determines what information is returned after the action. In particular, $\mathcal{T}$ constrains navigation according to the hierarchy and maintains which nodes have become accessible, whereas $\mathcal{O}$ exposes only the structural or diagnostic information permitted by the resulting state. We instantiate these functions through three interaction actions:

$$\mathcal{U} = \{\texttt{goto}(v), \texttt{check}(v), \texttt{finish}(s)\}, \tag{5}$$

corresponding to *where to investigate*, *what evidence to inspect*, and *when to conclude*.

`goto`(v): Navigation. The `goto` action moves the diagnostic focus to a selected node v. To constrain the investigation to structurally related regions, v must be either an ancestor of the current focus or a node within its descendant subtree:

$$v \in \mathcal{V}_{\text{goto}}(z_k) = \text{Anc}(z_k) \cup \text{Desc}(z_k). \tag{6}$$

For a valid target, $\mathcal{T}$ updates the current focus to v. If v is a leaf node, its local dependency neighborhood in G becomes observed:

$$\mathcal{N}(v) = \{u \mid (u, v) \in E\} \cup \{u \mid (v, u) \in E\}, \qquad G_v = G[\{v\} \cup \mathcal{N}(v)]. \tag{7}$$

The observation function $\mathcal{O}$ returns this newly exposed structural information to the agent. Thus, `goto` supports hierarchical navigation while additionally triggering local dependency verification when the investigation reaches a leaf node.

`check`(v): Evidence Acquisition. While `goto` determines the region being explored, `check` acquires additional diagnostic information from an already observed node. It can only be applied to a node that has already been observed: $v \in \mathcal{V}_k^{\text{obs}}$, where $\mathcal{V}_k^{\text{obs}}$ denotes the nodes exposed up to round k. Since `check` does not change the diagnostic focus, $\mathcal{T}$ leaves it unchanged. For an abstract node, `check` expands the node and exposes its child processes, providing a finer-grained view of the corresponding execution process.

`finish`(s): Root-Cause Localization. When the accumulated evidence is sufficient, the agent invokes `finish`(s). The transition function moves the environment to a terminal state,

$$\mathcal{T}(z_k, \texttt{finish}(s)) = z_{\text{term}}, \tag{8}$$

and the environment returns $\hat{s}_e = s$ as the predicted root-cause step. Its responsible agent is subsequently recovered as $a(\hat{s}_e)$.

Agentic Diagnosis. Diagnosis begins at the root of H. At each round, π_D selects an action based on its interaction history, while $\mathcal{T}$ and $\mathcal{O}$ determine how the environment changes and what information becomes available. Through `goto`, the agent progressively exposes relevant execution regions; through `check`, it selectively acquires their diagnostic content; and through `finish`, it terminates once sufficient evidence has been collected. The hierarchy therefore governs coarse-to-fine navigation, the dependency graph supports local verification, and the environment interface controls their progressive exposure to the diagnosis agent.

Method	Model	Step Accuracy ↑			Tolerance ↑		
		With GT	W/o GT	Avg.	$\Delta\leq1$	$\Delta\leq2$	$\Delta\leq3$
Random	–	4.16	4.16	4.16	–	–	–
All-at-Once	`GPT-4o`	5.26	3.51	4.39	12.1	19.8	30.2
All-at-Once	`Claude-3.5-Sonnet-v2`	6.00	2.10	4.05	14.9	22.3	35.0
All-at-Once	`DeepSeek-V4-Flash`	3.45	3.45	3.45	13.8	19.0	29.3
All-at-Once	`GPT-5.6-Luna`	8.62	6.90	7.76	15.5	24.1	27.6
Step-by-Step	`GPT-4o`	7.02	8.77	7.90	14.7	16.4	18.1
Step-by-Step	`Claude-3.5-Sonnet-v2`	6.60	6.90	6.75	16.6	18.9	20.9
Step-by-Step	`DeepSeek-V4-Flash`	6.90	13.79	10.35	10.3	19.0	31.0
Step-by-Step	`GPT-5.6-Luna`	18.97	20.69	19.83	36.2	36.2	39.7
Binary Search	`GPT-4o`	6.90	6.90	6.90	13.8	19.0	22.4
Binary Search	`Claude-3.5-Sonnet-v2`	6.90	6.90	6.90	13.8	19.0	22.4
Binary Search	`DeepSeek-V4-Flash`	8.62	3.45	6.04	15.5	22.4	27.6
Binary Search	`GPT-5.6-Luna`	15.52	10.34	12.93	24.1	27.6	32.8
AgenTracer	`AgenTracer-8B`	20.68	20.68	20.68	–	–	–
GraphTracer	`GraphTracer-8B`	28.63	27.97	28.30	–	–	–
CHIEF	`DeepSeek-V3.2-think`	29.31	<u>29.31</u>	29.31	–	–	–
ECHO	`Claude-3.5-Sonnet-v2`	28.10	26.80	27.45	35.1	38.6	42.1
TIDE	`DeepSeek-V4-Flash`	**34.48**	25.86	<u>30.17</u>	**51.7**	**58.6**	**67.2**
	`GPT-5.6-Luna`	**34.48**	**41.38**	**37.93**	<u>46.6</u>	<u>50.0</u>	<u>51.7</u>

Table 1: Step-level diagnosis performance on Who&When Hand-Crafted. $\Delta\leq k$ denotes localization within k steps of the ground-truth error. Tolerance is reported with ground-truth outcomes. "–" denotes unavailable results.

5 Experimental Setup

Benchmarks. We evaluate TIDE on *Who&When* (Zhang et al., 2025b) and *AgentRX* (Barke et al., 2026). For *Who&When*, we use its 58 hand-crafted trajectories and report Step-Level Accuracy and Tolerance under both settings with and without reference answers. For *AgentRX*, we evaluate on τ-Bench and Magentic-One and report Step-Level Accuracy and Average Step Distance.

Baselines. We compare against LLM-as-a-Judge methods from *Who&When* (Zhang et al., 2025b), including All-at-Once, Step-by-Step, and Binary Search, as well as AgenTracer (Zhang et al., 2026), GraphTracer (Zhang et al., 2025a), CHIEF (Wang et al., 2026), and ECHO (Banerjee et al., 2025). On *AgentRX*, we compare with its All-at-Once, Step-then-Category, and Taxonomy Checklist configurations (Barke et al., 2026).

Implementation. We instantiate TIDE and LLM-as-a-Judge baselines with `DeepSeek-V4-Flash` and `GPT-5.6-Luna`. For TIDE, the same underlying LLM is used for dependency identification, hierarchical abstraction, and diagnosis. For other methods, we report published results under their original model configurations. More details are provided in Appendix A.2.

6 Results Analysis

6.1 Overall Diagnosis Performance

Results on Who&When. As shown in Table 1, TIDE consistently improves step-level diagnosis over existing methods. With `GPT-5.6-Luna`, TIDE achieves the highest average Step Accuracy of 37.93, substantially outperforming the strongest prior baseline, CHIEF (29.31), with particularly strong performance when the ground-truth task outcome is unavailable. The gains also persist under controlled model comparisons: using the same base model, TIDE substantially outperforms all LLM-as-a-Judge variants for both `GPT-5.6-Luna` and `DeepSeek-V4-Flash`, indicating that the improvement is not merely due to stronger underlying models. Moreover, TIDE shows strong tolerant localization, reaching 67.2 at $\Delta\leq3$ with `DeepSeek-V4-Flash`, compared with 42.1 for ECHO. Together, these results demonstrate improvements in both exact and approximate error localization.

Method	Model	τ-Bench		Magentic-One	
		Step Acc. ↑	Avg. Dist. ↓	Step Acc. ↑	Avg. Dist. ↓
All-at-Once	`GPT-5`	29.9	5.4	32.6	22.5
Step-then-Category	`GPT-5`	31.6	5.5	**34.1**	19.7
Taxonomy Checklist	`GPT-5`	30.8	5.9	31.1	22.9
TIDE	`DeepSeek-V4-Flash`	27.6	6.2	31.8	6.0
	`GPT-5.6-Luna`	**37.9**	**5.0**	29.5	**4.5**

Table 2: Step-level diagnosis results on the AgentRX benchmark. Step Acc. denotes exact critical-step localization accuracy, and Avg. Dist. denotes the average distance between the predicted and ground-truth critical steps.

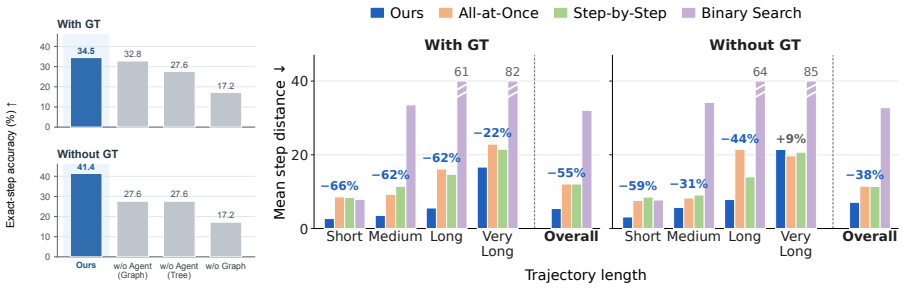


Figure 3: Analysis of TIDE on Who&When. **Left:** Ablation of agentic interaction and dependency-graph construction using GPT-5.6-Luna. **Right:** Mean step distance across trajectory lengths using DeepSeek-V4-Flash, with and without ground-truth task outcomes.

Results on AgentRX. Table 2 further evaluates TIDE on the τ-Bench and Magentic-One subsets of AgentRX. On τ-Bench, TIDE with `GPT-5.6-Luna` achieves the highest Step Accuracy of 37.9 and the lowest average step distance of 5.0, outperforming the strongest baseline on both metrics. On Magentic-One, while Step-then-Category achieves higher exact Step Accuracy, TIDE substantially reduces the average step distance from 19.7 to 4.5, indicating more precise localization when the exact critical step is not identified. This reduction in localization distance is also observed with `DeepSeek-V4-Flash`, which achieves an average distance of 6.0. Together with the Who&When results, these findings demonstrate that TIDE generalizes across different agentic systems and consistently improves the localization of execution failures.

6.2 Effect of Trajectory Environment and Agentic Interaction

Effect of Dependency Modeling. Figure 3 (left) examines the contribution of dependency modeling to diagnosis. Removing the dependency graph and instead constructing the hierarchical environment directly from the temporal trajectory substantially reduces exact-step accuracy from 34.5% to 17.2% with ground truth, and from 41.4% to 17.2% without ground truth. This degradation indicates that hierarchical abstraction alone is insufficient: explicitly modeling information dependencies between execution steps provides important structure for localizing the source of failure.

Effect of Agentic Interaction. We further remove agentic interaction and directly provide the constructed trajectory representation to an LLM-as-a-Judge. Using the dependency graph directly achieves 32.8% and 27.6% exact-step accuracy with and without ground truth, respectively, while using its hierarchical summary achieves 27.6% in both settings. In comparison, TIDE achieves 34.5% and 41.4%, with the largest improvement observed when the ground-truth outcome is unavailable. These results suggest that the benefit of TIDE arises not only from structuring the trajectory, but also from allowing the diagnosis agent to actively investigate this structure.

6.3 Diagnosis over Long-Horizon Trajectories

Performance across Trajectory Lengths. Figure 3 (right) evaluates localization as the diagnostic space grows, grouping trajectories by execution length. TIDE reduces mean step distance by 55%

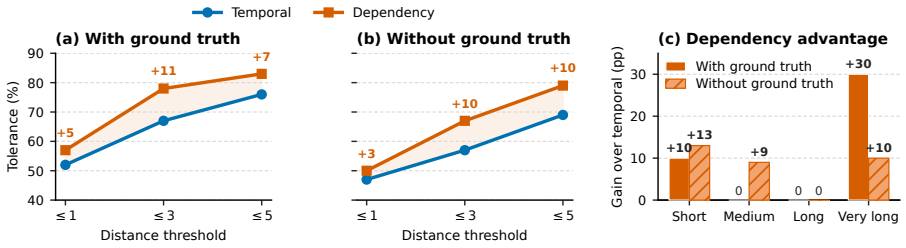


Figure 4: Temporal versus dependency-based localization tolerance across trajectory lengths on Who&When using `DeepSeek-V4-Flash`. Results report the percentage of predictions within $\Delta \leq 3$ of the ground-truth error under temporal step distance and dependency-graph distance.

with ground truth and 38% without ground truth compared with the strongest baseline. The advantage is pronounced from short to long trajectories but narrows on very long ones, indicating that TIDE better maintains localization performance as trajectories grow, although very long executions remain challenging.

Temporal vs. Dependency Distance. Figure 4 compares localization under temporal and dependency-graph distance. At $\Delta \leq 3$, dependency-based tolerance increases from 67.2% to 77.6% with ground truth and from 56.9% to 67.2% without ground truth. The gap reaches 30 percentage points on very long trajectories with ground truth, showing that diagnostically related steps can be temporally distant yet close through information dependencies.

Diagnosis Behavior. Figure 5 illustrates how the diagnosis agent navigates the trajectory environment. With `GPT-5.6-Luna` without ground-truth outcomes, the agent explores multiple top-level subtrees in 51.7% of trajectories but all subtrees in only 20.7%, suggesting selective rather than exhaustive exploration. When initially entering a subtree without the ground-truth error, it redirects to the correct subtree in 54.8% of cases and ultimately recovers the root-cause step in 29.0%, demonstrating its ability to revise the investigation as new evidence is acquired.

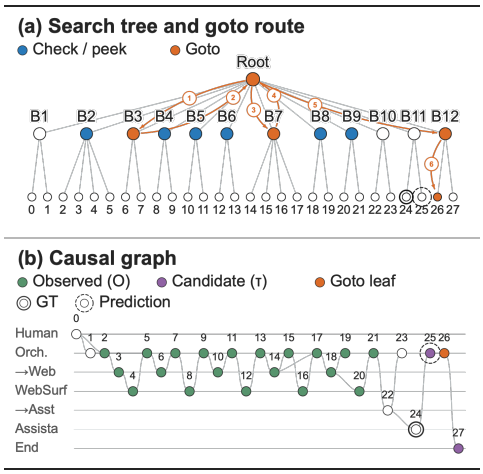


Figure 5: Case study of TIDE diagnosis.

7 CONCLUSION

We introduced TIDE, a framework that formulates agentic system diagnosis as **agent–environment interaction**, shifting the design focus from the diagnosis agent alone to the joint design of the agent and its trajectory environment. TIDE transforms a failed execution into an interactive environment that combines dependency modeling with hierarchical abstraction, enabling dependency-guided, coarse-to-fine investigation. An **Agent-as-a-Judge** adaptively interacts with this environment to navigate the diagnostic space, acquire relevant evidence, and localize the erroneous step. Experiments on Who&When and AgentRX demonstrate improved error localization across agentic systems and judge models, while further analyses highlight the benefits of dependency modeling and agentic interaction, particularly when diagnostically related steps are distant in execution order. Overall, our results suggest that jointly designing diagnosis agents and the environments through which they investigate trajectories provides a promising direction for long-horizon agentic system diagnosis.

AI USE STATEMENT

In this work, we have not used generative AI tools for tasks with required disclosure, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI

tools for polishing the manuscript to improve readability and for assisting with debugging software code. We have reviewed all AI-assisted work. All AI-assisted edits were reviewed and revised by the authors, and debugging suggestions were manually inspected and verified by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

Reproducibility Statement

To facilitate reproducibility, we provide an anonymous code repository containing the implementation of TIDE, experimental configurations, prompts, and instructions for reproducing our experiments: `https://anonymous.4open.science/r/Agent-diagnosis-artifact-8CAC/`.

References

Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru WANG, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. ISSN 2835-8856. URL https://openreview.net/forum?id=CTr3bovS5F.

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

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. Agentrx: Diagnosing ai agent failures from execution trajectories. *arXiv preprint arXiv:2602.02475*, 2026.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *Advances in Neural Information Processing Systems*, 38, 2025.

Mengzhuo Chen, Junjie Wang, Fangwen Mu, Yawen Wang, Zhe Liu, Huanxiang Feng, and Qing Wang. Seeing the whole elephant: A benchmark for failure attribution in LLM-based multi-agent systems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19888–19905, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.912. URL https://aclanthology.org/2026.acl-long.912/.

Guanting Dong, Xiaoshuai Song, Yuyang Hu, Jiajie Jin, Chenghao Zhang, Yifei Chen, Xiaoxi Li, Huaying Yuan, Xinyu Yang, Tongyu Wen, et al. Towards long-horizon agents: A survey. 2026.

Shangheng Du, Jiabao Zhao, Jinxin Shi, Zhentao Xie, Xin Jiang, Yanhong Bai, and Liang He. A survey on the optimization of large language model-based agents. *ACM Computing Surveys*, 58 (9):1–37, 2026.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 15419–15462. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/erdogan25a.html.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025.

Jiazheng Li, Emine Yilmaz, Bei Chen, and Thu Le. Towards self-improving error diagnosis in multi-agent systems. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2063–2077, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.98. URL https://aclanthology.org/2026.findings-acl.98/.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2511–2522, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.153. URL https://aclanthology.org/2023.emnlp-main.153/.

Tianyue Ou, Wanyao Guo, Apurva Gandhi, Graham Neubig, and Xiang Yue. AgentDiagnose: An open toolkit for diagnosing LLM agent trajectories. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 207–215, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-demos.15. URL https://aclanthology.org/2025.emnlp-demos.15/.

Hyogon Ryu, Jeonghwan Kim, Yewon Lim, Chaeun Lee, Jeongwook Kim, and Donghoon Ham. Online agent-as-a-judge: Situation-generating evaluation for interactive agents. *arXiv preprint arXiv:2606.08200*, 2026. doi: 10.48550/arXiv.2606.08200. URL https://arxiv.org/abs/2606.08200. ICML 2026 Workshop on Trustworthy AI for Good.

Wentao Shi, Yu Wang, Yuyang Zhao, Yuxin Chen, Fuli Feng, Xueyuan Hao, Xi Su, Qi Gu, Hui Su, Xunliang Cai, and Xiangnan He. AJ-bench: Benchmarking agent-as-a-judge for environment-aware evaluation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 25371–25413, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.1269. URL https://aclanthology.org/2026.findings-acl.1269/.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of computer science*, 18(6):186345, 2024.

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

Alva West, Yixuan Weng, Minjun Zhu, Zhen Lin, Zhiyuan Ning, and Yue Zhang. Abduct, act, predict: Scaffolding causal inference for automated failure attribution in multi-agent systems. *arXiv preprint arXiv:2509.10401*, 2025. doi: 10.48550/arXiv.2509.10401. URL https://arxiv.org/abs/2509.10401.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Miao Yu, Fanci Meng, Xinyun Zhou, Shilong Wang, Junyuan Mao, Linsey Pan, Tianlong Chen, Kun Wang, Xinfeng Li, Yongfeng Zhang, et al. A survey on trustworthy llm agents: Threats and countermeasures. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 6216–6226, 2025.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the llm agentic systems? In *International Conference on Learning Representations*, volume 2026, pp. 11377–11399, 2026.

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

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76583–76599. PMLR, 13–19 Jul 2025b.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2020. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Chenyang Zhu, Spencer Hong, Jingyu Wu, Kushal Chawla, Yuhui Tang, Youbing Yin, Nathan Wolfe, Erin Babinsky, and Daben Liu. 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)*, pp. 7659–7688, Rabat, Morocco, March 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.359. URL https://aclanthology.org/2026.eacl-long.359/.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 80569–80611. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhuge25a.html.

Algorithm 1 Trajectory Environment Construction

Require: Failed execution trajectory $\tau = (s_1, \ldots, s_T)$
Ensure: Trajectory environment structure $\mathcal{E}_\tau$ with dependency graph G and hierarchy H

$V \leftarrow \emptyset$, $E \leftarrow \emptyset$
▷ Construct dependency graph
for $i = 1, \ldots, T$ **do**
  Parse s_i into $v_i = \langle O_i, T_i, A_i, R_i \rangle$
  $V \leftarrow V \cup \{v_i\}$
  for all $v_j \in V$ such that $j < i$ **do**
    if $R_j \rightsquigarrow O_i$ **then**
      $E \leftarrow E \cup \{(v_j, v_i)\}$
    end if
  end for
end for
$G \leftarrow (V, E)$
▷ Construct hierarchical abstraction
Initialize H with each $v_i \in V$ as a leaf node
$\mathcal{Q} \leftarrow V$
for all $v_i \in V$ **do**
  $\phi(v_i) \leftarrow \{v_i\}$
end for
while further abstraction is possible **do**
  Traverse dependencies backward from terminal behaviours
  Identify a self-contained causal process $S \subseteq \mathcal{Q}$
  if S is found **then**
    Create abstract node $h = \langle I(h), O(h), C(h) \rangle$
    $\phi(h) \leftarrow \bigcup_{v \in S} \phi(v)$
    Add h as the parent of nodes in S in H
    $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus S) \cup \{h\}$
  else
    break
  end if
end while
return (G, H)

A Appendix

A.1 Implementation Details

We summarize the complete procedure of TIDE in Algorithms 1 and 2, covering trajectory environment construction and diagnosis through environment interaction, respectively.

A.2 Experimental Setup Details

A.2.1 Benchmarks and Evaluation

Who&When. We evaluate on the 58 hand-crafted trajectories from Who&When (Zhang et al., 2025b), which provide longer and more challenging execution traces. We use Step-Level Accuracy to measure exact error localization and Tolerance to measure whether the predicted error lies within a given distance from the ground-truth error step, following the benchmark. We evaluate both settings considered by the benchmark: with and without a reference answer to the task.

AgentRX. We evaluate on the two publicly available AgentRX (Barke et al., 2026) subsets, τ-Bench and Magentic-One. Following the benchmark, we report Step-Level Accuracy and Average Step Distance between the predicted and ground-truth critical steps. We consider the setting without access to a reference answer.

Algorithm 2 Diagnosis through Trajectory Environment Interaction

Require: Dependency graph G, hierarchy H, diagnosis agent π_D
Ensure: Predicted root-cause step $\hat{s}_e$

Initialize environment state z_0 at the root of H
Initialize observed nodes $\mathcal{V}_0^{\text{obs}}$ from the root
$o_0 \leftarrow \mathcal{O}(z_0)$
$h_0 \leftarrow (o_0)$
$k \leftarrow 0$
while diagnosis has not terminated **do**
  $u_k \sim \pi_D(\cdot \mid h_k)$
  if $u_k = \texttt{goto}(v)$ **then**
    if $v \in \text{Anc}(z_k) \cup \text{Desc}(z_k)$ **then**
      $z_{k+1} \leftarrow \mathcal{T}(z_k, \texttt{goto}(v))$
      if v is a leaf node **then**
        Expose the local dependency neighborhood G_v
      end if
      $o_{k+1} \leftarrow \mathcal{O}(z_{k+1})$
    else
      $o_{k+1} \leftarrow$ invalid-action observation
      $z_{k+1} \leftarrow z_k$
    end if
  else if $u_k = \texttt{check}(v)$ **then**
    if $v \in \mathcal{V}_k^{\text{obs}}$ and v is an abstract node **then**
      $z_{k+1} \leftarrow \mathcal{T}(z_k, \texttt{check}(v))$
      Expose the children of v in H
      $o_{k+1} \leftarrow \mathcal{O}(z_{k+1})$
    else
      $o_{k+1} \leftarrow$ invalid-action observation
      $z_{k+1} \leftarrow z_k$
    end if
  else if $u_k = \texttt{finish}(s)$ **then**
    $\hat{s}_e \leftarrow s$
    return $\hat{s}_e$
  end if
  Update $\mathcal{V}_{k+1}^{\text{obs}}$
  $h_{k+1} \leftarrow (h_k, u_k, o_{k+1})$
  $k \leftarrow k + 1$
end while

A.2.2 BASELINES

Who&When. We compare against: (1) **LLM-as-a-Judge** (Zhang et al., 2025b), including All-at-Once, Step-by-Step, and Binary Search; (2) **AgenTracer** (Zhang et al., 2026), which trains a specialized failure-attribution model; (3) **GraphTracer** (Zhang et al., 2025a), which reasons over inter-step dependencies represented as a graph; (4) **CHIEF** (Wang et al., 2026), which progressively localizes errors hierarchically; and (5) **ECHO** (Banerjee et al., 2025), which performs hierarchical error attribution from coarse-grained trajectory segments to fine-grained execution steps.

AgentRX. We additionally compare against **All-at-Once**, **Step-then-Category**, and **Taxonomy Checklist** (Barke et al., 2026), three configurations reported by AgentRX that do not use its generated violation evidence.

A.2.3 IMPLEMENTATION DETAILS

We instantiate TIDE with `DeepSeek-V4-Flash` and `GPT-5.6-Luna`. For the LLM-as-a-Judge baselines, we evaluate the same two models to control for the capability of the underlying judge, while additionally reporting results with the models used in prior work. For the remaining baselines, we report their published results using the original model configurations. Unless otherwise specified,

all reported baseline results are taken from the corresponding papers. For the LLM-as-a-Judge baselines, we evaluate the same two models to control for the capability of the underlying judge, while additionally reporting results with the models used in prior work. For the remaining baselines, we report their published results using the original model configurations. Unless otherwise specified, all reported baseline results are taken from the corresponding papers.

A.2.4 PROMPTS

We list the used prompts in our experiments.

Dependency Identification Prompt

```
You generate incremental OTAR causal-graph deltas from raw trajectories as forensic evidence. Preserve what
happened, who said or did it, when, and whether it was proposed, requested, attempted, completed, failed, uncertain,
or corrected. Fidelity outranks fluency, brevity, and uniform length.

# Causal edge definition

A causal edge represents information flow from one node to a later node.

Core rule:

source.result contributes to target.observation

Create a directed edge source -> target only when the source node's result contributes to, is reused by, constrains,
explains, or provides necessary information for the target node's observation.

Temporal constraints:

1. source must be earlier than target.
2. source.node_id must be smaller than target.node_id.
3. Never create an edge from a future node to a past node.
4. Never create an edge between nodes without evidence in the exposed observation/result text.

Layering rule:

The same edge meaning applies at every graph level. Whether nodes are bottom-level OTAR nodes, summary nodes, or
higher-level nodes, an edge always means source.result contributes to target.observation.

# Evidence boundary

Input contains exactly `history_or_nodes` and `current_raw_steps`. Process raw steps in order, creating exactly one
node per step. Copy each 0-based consecutive `step_id` and `agent` exactly.

For each step, its raw `content` is the only source for `thought`, `action`, and `result`; never move a later
output, error, answer, verification, correction, or termination backward. `observation` may contain only explicit
pre-action context in this raw content or an earlier exposed node result that the current behavior demonstrably
uses. Cite the latter as `From step <id> result: ...`. Do not infer hidden history or helpful facts. With no
supported pre-action input, use exactly `No pre-action observation is explicitly supported by the available
evidence.`

# Preserve evidence

Privately split each raw message, in order, into its distinct context, reasoning/intention,
communication/delegation, tool request or operation, returned evidence/error, answer, verification/correction, and
termination events. Preserve every unique event in the appropriate OTAR field.

Make a private `MUST_COPY` ledger. Copy verbatim somewhere in the node (including capitalization, numbers, units,
and punctuation):

1. Final or answer-bearing expressions and termination/completion markers. Treat any text that explicitly signals
termination, completion, or final-answer delivery as a termination signal, regardless of its exact wording or format.
2. Material numbers, dates, thresholds, units, identifiers, named values, and the exact speaker/recipient when
identity matters.
3. Tool/API/function/file/command names with decisive arguments, queries, or selected/opened evidence URLs.
4. Exit or HTTP status, exception class/message, error or policy code, and explicit failure lines.
5. Negation, uncertainty, qualification, correction, contradiction, and statements that something was not found,
checked, executed, or verified.

You may summarize plans, boilerplate, metadata, long search-result lists, code blocks, and repetitive traceback
frames, but retain their unique operation, outcome, failure, answer, correction, and unresolved state. Never delete
a unique causal event merely to shorten a node.

# OTAR fields
```

```
`observation` is only supported information available before this step's behavior; never use the current action as its observation.

`thought` is only explicit reasoning, intention, interpretation, or decision rationale, retaining uncertainty. Use exactly `Not explicitly stated in the raw step.` only when none is explicit.

`action` is behavior performed or communicated in this step, with evidence-supported status: `will`/`intend`/`plan` (and bare `let's`) are intentions unless completion is shown; code supplied to another executor or a tool request is provided/requested, not executed; use executed/obtained/completed/verified/confirmed only with same-step support. A human task message only supplies a request and constraints.

`result` is what this step made observable: retain actual tool/environment output, empty result, status, or error. For communication, instruction, delegation, verification, answer, or correction, preserve the delivered content as the result, not merely `no separate result`. For a request executed later, preserve the decisive request and say no execution result appears in this step. Do not add blame, root cause, benchmark labels, unstated contracts, or advice; keep claims and verification claims distinct from facts.

Use as much text as the distinct events and `MUST_COPY` items require: concise for a simple step, detailed for a compound one. Do not pad with inferred reasoning or omit required evidence for uniformity.

# Private checks and causal edges

Before JSON, ensure `IDENTITY`, verbatim `MUST_COPY_COVERAGE`, ordered `EVENT_COVERAGE`, `ENTAILMENT`, `TEMPORAL_BOUNDARY`, observation provenance, thought precision/recall, supported action status, communication result, field boundary, polarity, cross-field consistency, and `NO_DIAGNOSIS`. If paraphrase risks a check, copy the evidence. Keep these checks private.

For every new target, compare its observation with all earlier new-node and history-node results. Add `source -> target` only when a specific exposed result is reused, answered, verified, corrected, or materially constrains that observation. State the matching source-result and target-observation facts in the reason. Do not add edges for chronology or topic alone, omit direct communication dependencies, target history nodes, or point backward.

# Output contract

Return only one JSON object with exactly `new_nodes` and `new_edges`, with no Markdown. `new_nodes` is in raw-step order and each node has only non-empty `node_id`, `agent`, `observation`, `thought`, `action`, and `result` (`node_id` is numeric). Each edge has only `source`, `target`, `reason`, and numeric `confidence` from 0.0 to 1.0.

{
  "new_nodes": [
    {
      "node_id": 0,
      "agent": "...",
      "observation": "...",
      "thought": "...",
      "action": "...",
      "result": "..."
    }
  ],
  "new_edges": [
    {
      "source": 0,
      "target": 1,
      "reason": "...",
      "confidence": 0.0
    }
  ]
}

# Forensic event, status, and dependency handling

Before drafting, make a private event ledger for every raw step. Include every task constraint, received claim, named entity, identifier, code/file/tool/API name and decisive argument, placeholder, value/unit/date, error/status, answer, request, and termination marker. Copy each material item into its proper OTAR field; do not replace it with “the task”, “the code”, “the prior claim”, or a generic summary. A code request must retain its concrete operation/identifier and decisive arguments; a received claim must retain the entity/value being received. This applies to all domains and is not a requirement to copy boilerplate or repeated traceback frames.

Classify each Action before writing it: (1) observable operation with a same-step tool/environment return or retrieved artifact; (2) supplied/requested operation awaiting another step; or (3) delivered claim, conclusion, interpretation, agreement, or task context. For class (3), Action is exactly a short communication description such as `Reported an unobserved claim and conclusion.` Do not include the claim's narrative or any operation verb in Action. Put the complete narrative in Result, attributed as the speaker's claim. “After searching”, “manual verification”, a source citation, a code block, a message heading, confidence wording, first-person narration, and
```

```
similar unsupported narrative cues are claims, not evidence of class (1). Only a same-step returned record permits
`executed`, `searched`, `performed`, `verified`, `confirmed`, `obtained`, or `completed` as an action status.

Make edges canonically after the node pass. For each target Observation, cite each explicitly reused earlier Result
as `From step <id> result: ...` and emit the corresponding edge. Add no edge without a concrete source-result /
target-observation dependency. Citations, request-to-terminal execution, tool-result-to-response,
conclusion-to-explicit agreement, and analogous explicit information-flow patterns are direct dependencies; shared
topic and chronology are not. This exact rule fixes both the required edge and the allowed edge set without adding
diagnosis.

# Supplied operations and direct references

When a raw step supplies code, a command, a tool call, or a similar operation without a same-step returned record,
its Action must explicitly say `provided`, `requested`, `proposed`, or an equivalent evidence-supported status,
never merely an intended execution; Result retains the operation, decisive arguments, and absent return.

When a raw step explicitly identifies an earlier message or event as its input, including an input whose relevant
property is absence or a similar explicit dependency, cite that earlier Result in Observation and emit its direct
edge. Do not infer edges from topic or order.

# Boundary fields

For node 0, set Observation to `User input: <user input from raw step-0 content>`. If raw step 0 contains only user
input, preserve it verbatim. If it mixes user input with other information, extract only the user-input portion for
Observation. This requirement applies only to Observation; populate Thought, Action, and Result from all supported
step-0 content under the standard OTAR rules above. Do not replace the user input with a generic task summary.

For the final node, treat the final raw step as containing the final answer and summarize that final answer
concisely in Result. If the step mixes the final answer with other material, extract its answer-bearing portion
while preserving key values, entities, qualifications, completion status, and any decisive error evidence. Do not
copy the complete raw message or a full traceback, and never reduce an answer-bearing final response to a bare
termination or completion summary.
```

Hierarchical Abstraction Prompt

```
You create a short evidence-preserving behavior summary for one abstract node in a hierarchical agent-trajectory
tree. The summary will be shown to a diagnosis agent together with the node's existing Observation and Result
boundaries.

Summarize what the covered children actually did. Preserve the most diagnostically discriminative visible facts:
agent-attributed decisions and actions, tool calls and observed outcomes, unsupported claims, contradictions, failed
or successful recovery, verification or lack of verification, handoffs, and important unresolved requirements.
Include step ids and agent names when they are supplied. Distinguish proposed, executed, observed, claimed, failed,
and unresolved states. A tool or environment outcome is evidence, not itself an agent action.

Do not decide the root cause, assign blame, mention a benchmark or ground truth, import facts outside the supplied
children, or infer an unstated role contract. Do not merely repeat the first Observation or last Result. Prefer
concrete behavior and evidence over generic phase descriptions. If the children contain no anomaly, describe the
normal progress and remaining state without inventing one.

Write at most 90 English words as one compact paragraph. Return exactly:

{
  "summary": "..."
}
```

Diagnosis Agent Prompt

```
You are a ReAct-style diagnostic search agent operating over a hierarchical agent-trajectory tree. Your goal is to
identify the earliest decisive error.

# Decisive-error diagnostic rubric

Use only the supplied task and visible tree context. Never infer hidden labels, unshown events, external facts, or
task-specific evaluation conventions.

Find the earliest **decisive error**. A candidate mistake is decisive only when correcting that agent behavior while
keeping the earlier prefix fixed and allowing later behavior to adapt would plausibly change the original failed
trajectory into a successful one. A mistake that the later system successfully bypasses or corrects, or that is not
a key cause of the final failure, is not decisive merely because it occurred earlier. If multiple visible mistakes
pass this intervention test, select the earliest one.

Apply these rules in order:
```

```
1. The user/human input step and an initial plan are task context, not errors. A plan is a proposal and remains
recoverable unless its visible content contradicts a requirement and the trajectory then executes that content
without correcting it. Do not select step 0 or an early plan merely because it is first in time.
2. Attribute an error to the step whose agent action, decision, omission, or result actually introduces or adopts
the harmful state. A step whose observation or result merely mentions an earlier failure is not the error. A
transcription, search, or execution defect belongs to the step that performed or responsibility-bearingly accepted
it only when correcting that step can plausibly prevent the downstream failure.
3. Require a visible mismatch with the task, expected outcome (if supplied in the task context), prior evidence, or
an explicit requirement. Do not call a plan erroneous merely because it is broad, a query erroneous because a better
query exists, or evidence erroneous merely because it is unverified.
4. A retrieval, search, computation, or execution step is decisive when its visible behavior or result lacks
information the task explicitly requires, that defect remains uncorrected, the later trajectory relies on it or
loses the required path because of it, and correcting this step would plausibly turn the trajectory from failure to
success. Later failure alone does not make every earlier incomplete attempt decisive. If a later step correctly
obtains the missing information or otherwise bypasses the defect, the earlier mistake is not decisive.
5. Proposals, tentative guesses, handoffs, pending actions, and incomplete progress remain recoverable unless they
introduce a false premise, discard a required opportunity, or are adopted as sufficient in a way that passes the
intervention test.
6. The final answer or final wrong output is a propagation step, not the initial error, unless it is the first
decisive behavior. Walk backward from the final failure, but select an earlier candidate only if correcting it
plausibly prevents that failure; visibility, temporal priority, and an uncorrected defect are not sufficient without
this causal link.
7. Perform the intervention check explicitly: replace only the candidate with a reasonable correct behavior, keep
the earlier history fixed, and allow later actions to adjust. If the trajectory would still fail, if the mistake was
actually corrected or bypassed, or if a normal next action could still succeed without recognizing and reversing the
candidate, continue later. When several candidates each pass this test, choose the earliest one.

Never select a candidate that the stated reason calls sound, merely pending, successfully recovered, causally
irrelevant to the final failure, or free of a decisive error. When evidence is genuinely ambiguous, use lower
confidence instead of inventing a defect.


# Tree navigation

The conversation is append-only. `check` expands an allowed visible abstract node without moving the current focus.
`goto` changes focus to one node copied from `target_constraints.goto`. The environment permits `goto` only to a
visible descendant in the current subtree or to an ancestor on the current `active_path`. Direct sibling and cousin
jumps are illegal.

Use `check` when one more level of evidence will distinguish live branches. Use `goto` when visible evidence
supports committing to a subtree or leaf. To switch branches, first `goto` the lowest useful ancestor, compare its
visible alternatives, and descend in a later action. If an invalid-action event contains `recommended_next_action`,
copy that action and target exactly.

# Audited candidate registry

The environment owns a persistent, deduplicated `state.candidate_registry` and one model-assessed
`state.likely_candidate`.

`check` only reveals tree evidence. It never registers a candidate. A `goto` to a visible leaf automatically
inspects that leaf's radius-1 causal neighborhood and registers the anchor plus all returned neighbors with full
OTAR evidence. This audit is part of the `goto`; never emit `inspect_neighbors`.

`auditable_leaf_goto_targets` lists the currently legal leaf goto targets that will immediately produce the first
local audit. Registry membership does not make a hidden tree node visible and does not grant `goto` permission.
Near the deadline, the environment may expose only these leaf targets so one later call remains for an explicit
`finish`.

The registry accumulates across exploration. While it is empty, return `likely_candidate=null`; the environment
also canonicalizes a premature value to null. After the registry becomes nonempty, every response must include the
complete `likely_candidate` object required by the latest response contract. It may repeat unchanged.

Assess only these two causal questions:

1. Is this the first step whose own decision, action, or omission committed the trajectory to wrong or missing
task-required information? Merely returning or receiving imperfect evidence that a later step mishandles is not
enough; the step must itself be the decisive responsible failure.
2. Was that information actually repaired later? State Yes or No and what was truly obtained and adopted.

An attempted retry, replanning statement, or intention to verify is not an actual repair unless the required
information was subsequently obtained and used.

`finish` may select any integer step listed by `target_constraints.finish`. Finishing consumes the current action
budget. In the same response, set `likely_candidate.step_id` equal to `selected_step_id` and provide both causal
```

```
answers for that selected step. When the remaining budget is tight, finish the strongest registry member that passes diagnostic rubric.

`state.visible_tree`, `state.next_hop_for_visible_target`, `auditable_leaf_goto_targets`, `legal_action_types`, and `target_constraints` are authoritative. Return the exact JSON object required by the latest `response_contract`; do not add `confidence`, `candidate_update`, status fields, or markdown.
```