

Causal Attribution Meets Impact-aware Verification: Debugging LLM-based Multi-Agent Systems via Static–Dynamic Synergy

Yawen Wang^{1,2,3*}, Wenjie Wu^{1,2,3*}, Junjie Wang^{1,2,3†}, Qing Wang^{1,2,3†}

¹Institute of Software, Chinese Academy of Sciences, Beijing, China

²State Key Laboratory of Complex System Modeling and Simulation Technology
Beijing, China

³University of Chinese Academy of Sciences, Beijing, China

{yawen2018, junjie, wq}@iscas.ac.cn, wuwenjie26@mails.ucas.ac.cn

Large Language Model (LLM)-based Multi-Agent Systems (MASs) are vulnerable to stochastic reasoning and coordination failures, yielding verbose, unstructured trajectories that are difficult to debug. Existing debuggers typically rely on either post-hoc static analysis, which suffers from ambiguous causal attribution, or dynamic verification, which incurs prohibitive computational costs and execution instability. To bridge this gap, we propose SynDebug, a static–dynamic synergistic debugging framework. Statically, SynDebug reconstructs execution topologies from flat logs via a *Hierarchical Causal Graph* and prioritizes root-cause candidates using synthesized *Virtual Oracles*. Dynamically, it performs targeted interventions on prioritized candidates to progressively recalibrate suspiciousness rankings based on execution impact. Evaluated across three MAS frameworks and four benchmarks, SynDebug outperforms six baselines, improving the repair success rate by 5.59%–81.81% while maintaining competitive efficiency.

1 Introduction

Large Language Models (LLMs) enable autonomous agents with advanced perception (Zheng et al., 2024), planning (Erdogan et al., 2025), and reasoning (Putta et al., 2024) capabilities. By orchestrating these agents, LLM-based Multi-Agent Systems (MASs) tackle complex tasks ranging from software engineering (Yang et al., 2024; Hong et al., 2023) to general problem-solving (Mialon et al., 2023; Yoran et al., 2024). However, complex agent-tool-environment interactions introduce substantial fragility, with failure rates reaching up to 86.7% on challenging benchmarks (Cemri et al., 2025). Diagnosing these failures is critical, yet the stochasticity and complexity of MASs render conventional debugging approaches inadequate.

Unlike traditional software with deterministic business logic, MAS behavior emerges at runtime via stochastic inference and inter-agent coordination (Wang et al., 2026). Consequently, failures are not traceable to static code locations; instead, they manifest within unstructured execution trajectories—verbose logs capturing natural language reasoning, multi-step communication, and tool invocations (Zhang et al., 2025d; Bouzenia and Pradel, 2025). This paradigm shift, coupled with LLM non-determinism, makes classical debugging techniques (e.g., program slicing (Weiser, 1984), delta debugging (Zeller, 2002), and breakpoints (Zhang et al., 2013)) insufficient for tracing opaque reasoning deviations or inter-agent misalignments.

Recent work in MAS debugging relies on two paradigms: (1) **Static Analysis**, which diagnoses failures post-hoc via LLM-based judges (Zhang et al., 2025d; Banerjee et al., 2025), spectrum analysis (Ge et al., 2025), or fine-tuned tracers (Zhang et al., 2025a,b); and (2) **Dynamic Debugging**, which validates failure hypotheses through active intervention and re-execution (Ma et al., 2025a; Zhu et al., 2025). Nevertheless, both paradigms face significant bottlenecks when grappling with the inherent complexity of agentic interactions.

Existing **static analysis** treats execution logs as flat textual streams, overlooking embedded causal structures. This makes capturing long-range and cross-step dependencies difficult. As shown in Fig. 1, without explicit causal tracking, the executor’s error at Step 16 remains latent until manifesting as a failure at Step 19. Disentangling root causes from downstream symptoms is especially challenging when planning and execution roles are intertwined. For instance, the constraint violation at Step 19 could stem from Orchestrator planning (Step 14) or WebSurfer execution (Step 16), leading to hallucinated attributions in long-context reasoning. Compounded by MAS stochasticity, these static diagnoses remain *unverified hypotheses* re-

*These authors contributed equally to this work.

†Corresponding authors.

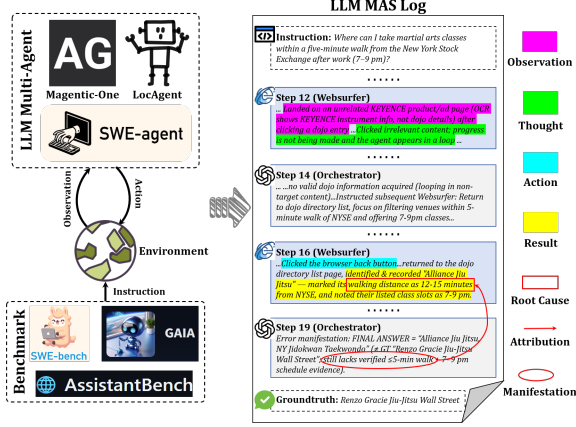


Figure 1: Unsatisfied constraint failure log: *Orchestrator* marks constraints (Step 14, red) overlooked by *Websurfer* (Step 16).

quiring dynamic validation.

Existing **dynamic debugging** techniques try to resolve these ambiguities via re-execution and intervention. However, without accurate static guidance, dynamic verification easily falls into a blind search in a non-deterministic state space. The stochastic nature of LLM outputs further exacerbates this issue by hindering stable failure reproduction across replays, and blind replays or random interventions impose substantial computational overhead.

To address these challenges, we propose SynDebug, a Static–dynamic synergistic **Debugging** framework. Instead of a naive combination, SynDebug advances both paradigms. Statically, it reconstructs execution topologies from flat logs via a *Hierarchical Causal Graph* (HCG) and leverages synthesized *Virtual Oracles* for causal backtracking to prioritize root-cause candidates. Dynamically, SynDebug employs an adaptive, impact-aware intervention strategy instead of relying on unguided retries. It iteratively perturbs the top candidate, quantifies the execution impact as a numerical delta, and updates suspiciousness scores to steer the search toward the true root cause. Crucially, the two phases operate in an iterative loop: static causal attribution narrows the search space, while dynamic verification quantitatively refines candidate suspiciousness to enable efficient and reliable debugging.

We evaluate SynDebug through extensive experiments across three representative MASs on four diverse benchmarks, spanning general-purpose tasks, program repair, and codebase localization. Compared with six widely used and state-of-the-art (SOTA) baselines over three independent trials, SynDebug consistently achieves superior performance, improving the repair success rate by

5.59%–81.81% over the SOTA debugging baseline, DoVer (Ma et al., 2025a), across all settings. The synergy between static causal attribution and dynamic impact-aware interventions enables SynDebug to achieve higher debugging precision while maintaining comparable token consumption and latency. Our main contributions are as follows:

- **Framework:** SynDebug¹, a debugging framework bridging static causal attribution with dynamic, impact-aware verification.
- **Static Attribution:** A graph-based representation that structures verbose trajectories to prioritize potential root causes.
- **Dynamic Verification:** An intervention-driven verification mechanism that iteratively refines rankings based on execution impact.
- **Evaluation:** Experiments across three MASs and four benchmarks, showing improvements in repair success, stability, and efficiency.

2 Preliminaries & Problem Formulation

We model an LLM-based MAS as $\mathcal{M} = \langle \mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$, where $\mathcal{N}$ denotes the set of agents, $\mathcal{S}$ the state space, $\mathcal{A}$ the action space, and $\mathcal{P}$ the transition function. At each timestep t , an active agent i_t takes action $a_t \in \mathcal{A}$ to transition the system state. The execution yields a trajectory $\tau = \{s_t, (i_t, a_t)\}_{t=0}^T$ with an outcome $Z(\tau) \in \{0, 1\}$, where 1 denotes task failure.

Given a failed trajectory τ ($Z(\tau) = 1$), the goal of *failure debugging* is to identify the root cause of the failure. Following prior counterfactual formulation (Zhang et al., 2025d), we define the decisive error indicator at step t as $\Delta_t(\tau) = \mathbb{I}(Z(\tilde{\tau}^{(t)}) = 0)$, where $\tilde{\tau}^{(t)}$ represents the counterfactual trajectory where the execution at step t is corrected while all other steps remain unchanged. Due to cascading errors, a trajectory may contain multiple decisive errors. The root cause is defined as the earliest decisive error $t^* = \arg \min_{t: \Delta_t(\tau)=1} t$. Our objective is to localize the ground-truth root-cause step t^* by analyzing τ or through targeted re-execution. A detailed categorization of MAS paradigms (e.g., serial vs. parallel, hand-crafted vs. automatically optimized) is provided in Appx. A.1.

3 Related Work

Static Failure Attribution. Static approaches analyze post-hoc execution logs to localize faults

¹Publicly available at <https://doi.org/10.5281/zenodo.19346452> to support reproducibility.

without runtime re-execution. They primarily rely on pattern- or spectrum-based heuristics (Zhang et al., 2025d; Ge et al., 2025), structured causal abstractions (Banerjee et al., 2025; West et al., 2025), or specialized fine-tuned tracer models (Zhang et al., 2025a,b). Although computationally efficient, static methods are fundamentally limited because their diagnostic outputs remain untested hypotheses that cannot account for the runtime stochasticity of agent environments.

Dynamic Debugging and Verification. Dynamic paradigms validate failure hypotheses through active interventions—such as state-trace editing or counterfactual re-runs—and subsequent system re-execution (Ma et al., 2025a; Zhu et al., 2025; Epperson et al., 2025; Ma et al., 2025b). While providing high-fidelity verification, these methods suffer from substantial computational overhead (in execution latency and token consumption) due to the lack of an efficient, guided search strategy over long trajectories.

Our Synergy. Bridging these paradigms, SynDebug establishes a **static–dynamic synergy**. Unlike unverified static inference or exhaustive dynamic trials, SynDebug utilizes high-precision rankings from static causal attribution to guide impact-aware dynamic verification, achieving accurate localization with minimal replay overhead. A detailed literature review is provided in Appx. A.2.

4 Method

As in Fig. 2, SynDebug combines static causal attribution with dynamic verification. Statically, it reconstructs a *Hierarchical Causal Graph* (HCG) to rank candidates via backtracking and counterfactual attribution. Dynamically, it iteratively intervenes on the top candidate, updating suspiciousness scores based on execution impact to localize the root cause.

4.1 Static Causal Attribution

The static phase of SynDebug transforms flat trajectories into a HCG, enabling structured, divide-and-conquer causal attribution.

4.1.1 HCG Construction

To capture the latent execution topology underlying error propagation, we construct a HCG denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which abstracts the raw trajectory τ into semantic nodes and causal dependencies across multiple granularities.

Hierarchical Nodes. The node set $\mathcal{V} = \mathcal{V}_{sub} \cup \mathcal{V}_{agt}$ represents high-level semantic execution units with structured attributes as detailed in Fig. 3a:

(1) **Subtask Nodes** ($S_k \in \mathcal{V}_{sub}$) represent distinct task phases. To partition the raw trajectory τ into a sequence $\{S_k\}_{k=0}^K$ that is both logically coherent and faithful to the execution log, we derive subtasks via *RAG-based Task Decomposition*, using human-annotated solution exemplars from existing benchmarks (e.g., GAIA (Mialon et al., 2023), AssistantBench (Yoran et al., 2024)) as few-shot prompts to guide the LLM in generating a modular task plan. The generated subtasks are further refined through *Trajectory-aligned Reflection*, which verifies the alignment between S_k and raw trajectory τ to prevent hallucinated decomposition.

(2) **Agent Nodes** ($a \in \mathcal{V}_{agt}$) represent atomic execution units corresponding to specific agents within a subtask. Each node is characterized by an *OTAR* tuple (Observation, Thought, Action, Result), extended from the *TAR* schema (Bouzenia and Pradel, 2025), and extracted from raw logs using an LLM-based *OTAR* parser.

Causal Edges. The edge set $\mathcal{E} = \mathcal{E}_{sub} \cup \mathcal{E}_{agt} \cup \mathcal{E}_{step}$ defines causal pathways for error tracing as illustrated in Fig. 3b. *Subtask Edges* and *Agent Edges* capture high-level task progression and inter-agent collaboration, respectively, using *Counterfactual Patterns* Φ (e.g., $\text{Bias}(u) \xrightarrow{\Phi} \text{Anomaly}(v)$). At the finest granularity, *Step Edges* represent explicit data-flow dependencies (e.g., variable references) between execution steps, serving as evidence links for tracing data-related error propagation.

4.1.2 Oracle-Guided Backtracking and Counterfactual Attribution

Based on the HCG, we identify root causes by synthesizing intermediate supervision (i.e., *Virtual Oracles*) and performing top-down backtracking with causal attribution.

Oracle-Guided Backtracking. For each S_k , we synthesize a subtask *Virtual Oracle* $\mathcal{O}_k = \langle \mathcal{G}_{sub}, \mathcal{P}_{pre}, \mathcal{E}_{key}, \mathcal{C}_{acc} \rangle$, which specifies the expected subtask goal, preconditions, key evidence, and acceptance criteria. These intermediate verifiers guide causal backtracking over the HCG. We formulate oracle synthesis as a generation function $\mathcal{F}_{gen}$ parameterized by an LLM: $\mathcal{O}_k = \mathcal{F}_{gen}(\mathcal{I}, \mathcal{O}_{<k}, \tau_{>k}, \tau_s)$. The inputs include the task instruction $\mathcal{I}$, previously generated oracles $\mathcal{O}_{<k}$, the remaining trajectory $\tau_{>k}$, and similar task decomposition examples τ_s retrieved via RAG.

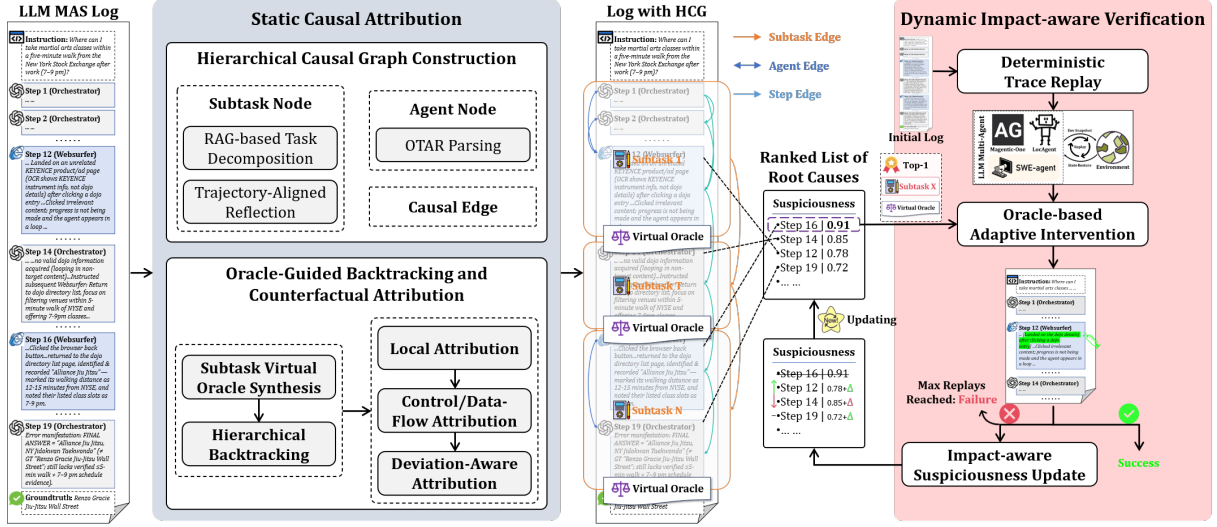


Figure 2: The overview of SynDebug.

Using the HCG and synthesized oracles, we perform a top-down backtracking procedure to progressively narrow the failure scope from subtasks to individual execution steps. This process relies on a LLM evaluator $\mathcal{F}_{eval}$ that compares actual execution outcomes against oracle constraints at multi-levels: (1) *Subtask level*. For each subtask S_k , $\mathcal{F}_{eval}$ compares its execution output with the oracle’s *Goal* ($\mathcal{G}_{sub}$) and *Acceptance Criteria* ($\mathcal{C}_{acc}$). A binary output of 1 indicates inconsistency, marking the subtask as a failure candidate: $\mathcal{C}_{sub} = \{S_k \in \mathcal{V}_{sub} \mid \mathcal{F}_{eval}(S_k, \langle \mathcal{G}_{sub}, \mathcal{C}_{acc} \rangle) = 1\}$. (2) *Agent level*. For each candidate subtask $S_k \in \mathcal{C}_{sub}$, we analyze its constituent agents. The evaluator $\mathcal{F}_{eval}$ checks whether an agent’s *OTAR* tuple ($a \in S_k$) is consistent with the oracle’s *Preconditions* ($\mathcal{P}_{pre}$) and *Key Evidence* ($\mathcal{E}_{key}$). Agents with detected inconsistencies ($= 1$) are identified as candidates: $\mathcal{C}_{agt} = \{a \in S_k \mid \mathcal{F}_{eval}(a_{otar}, \langle \mathcal{P}_{pre}, \mathcal{E}_{key} \rangle) = 1\}$. (3) *Step level*. For each candidate agent $a \in \mathcal{C}_{agt}$ within $S_k \in \mathcal{C}_{sub}$, we further inspect its executed steps $\mathcal{X}$. $\mathcal{F}_{eval}$ verifies each step $x \in \mathcal{X}$ by comparing its execution details (e.g., inputs and outputs) with the agent’s *OTAR* summary (a_{otar}) and the subtask oracle constraints ($\mathcal{O}_k$). Based on this comparison, it assigns a *suspiciousness score* to each inspected step, while steps not examined in this stage receive a score of 0:

$$S_x(\tau) = \begin{cases} \mathcal{F}_{eval}(x, \langle a_{otar}, \mathcal{O}_k \rangle), & x \in \mathcal{X} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Counterfactual Attribution. To distinguish the true root cause from propagated symptoms within $\mathcal{X}$, we design an LLM-based attributor with a pro-

gressive screening strategy along two dimensions: causal scope (local vs. non-local) and dependency type (control vs. data). Finally, a validity filter prunes transient errors that are subsequently self-corrected. The attributor would adjust the suspiciousness scores of steps in $\mathcal{X}$ based on the following heuristics, where higher scores indicate a greater likelihood of being the true root cause.

(1) Local Attribution. We first determine whether the localized error originates at step x itself. If no upstream triggers satisfy the counterfactual pattern Φ bound to the causal edges, the error is attributed locally to the current step.

(2) Control/Data-Flow Attribution. For non-local errors, we first decouple *Control-Flow* responsibility between planning and execution. Planner responsibility arises when the planner repeatedly produces identical thoughts or commands despite error feedback. Executor responsibility is attributed when the planner proposes valid strategy shifts but the executor continues to yield abnormal results. We then reconstruct the error propagation path via *Data-Flow* edges using step-level dependencies $\mathcal{E}_{step}$ and *OTAR* variable references. Auditing data consistency along this path identifies the step where valid inputs are first corrupted, distinguishing the true data generator (root cause) from downstream propagators (symptoms).

(3) Deviation-Aware Attribution. Finally, we assess *Causal Reversibility*. For a suspicious upstream step x_t , we examine the subsequent trajectory $\tau_{>t}$ for evidence of self-correction. If a later step x_{t+k} re-satisfies the oracle criteria (i.e., the system returns to a valid state), the deviation at x_t is considered reversible and assigned no responsi-

bility. Irreversible deviations are prioritized as the final root cause.

This phase outputs a *Suspiciousness-ranked List of Root Causes*, where each step x is assigned a suspiciousness score $S_x(\tau)$ according to the strength of the causal evidence gathered during static phase.

4.2 Dynamic Impact-aware Verification

Using the ranked candidates, SynDebug conducts impact-aware verification via adaptive interventions. At each iteration, the top candidate is perturbed, and the trajectory is deterministically replayed to observe behavioral changes. The resulting impact, measured against the original outcome, updates suspiciousness scores to guide subsequent interventions.

4.2.1 Deterministic Trace Replay

Execution nondeterminism, arising from stochastic LLM sampling, asynchronous interactions, and shared mutable contexts, often prevents reliable failure reproduction. To address this, we introduce *Deterministic Trace Replay* to stabilize execution contexts for consistent replay.

Because reproducibility depends heavily on the MAS control-flow architecture, SynDebug adopts two target replay strategies (Fig. 4): **(1) Serial MASs** (e.g., SWE-agent (Yang et al., 2024), LocAgent (Chen et al., 2025)): We enforce deterministic replay by fixing random seeds and utilizing zero-temperature LLM sampling, re-executing the trajectory from the beginning to the candidate step. **(2) Parallel MASs** (e.g., Magentic-One (Fourney et al., 2024), CaptainAgent (Song et al., 2024)): We snapshot and restore runtime contexts—including agent memory, tool states, and message events—at the candidate step to recover original execution dependencies and interaction orders.

We further evaluate the replay fidelity of these architecture-aware strategies across serial and parallel MASs; detailed experimental settings and results are reported in Appx. A.8 and Table 13.

4.2.2 Oracle-based Adaptive Intervention

For a candidate step x , SynDebug performs an *Oracle-based Adaptive Intervention* by replacing its output with an oracle-aligned correction to verify if it is the root cause. The trajectory is then replayed from this point under the original MAS policy to observe behavioral changes. To approximate the expected behavior, we prompt an LLM to synthesize a corrected output satisfying the subtask-

level *Virtual Oracle* $\mathcal{O}_k$ obtained during the static phase. We categorize these adaptive interventions into three semantic types:

(1) Data-Flow Intervention: Targets factual or reasoning errors by replacing step outputs with oracle-consistent values, thereby preventing corrupted information from propagating downstream. **(2) Control-Flow Intervention:** Corrects planning or decision-making errors by modifying control signals. This enables (i) *branch redirection* to steer execution toward a valid path, and (ii) *loop termination* to break ineffective reasoning cycles. **(3) Reflection Intervention:** Addresses omitted actions by injecting reflective corrections, explicitly executing missing operations like re-querying information or validating intermediate results.

4.2.3 Impact-aware Suspiciousness Update

When an intervention on the top candidate A fails to resolve the failure, the resulting counterfactual trajectory τ' provides additional evidence to refine the suspiciousness scores of the remaining candidates. To quantify the impact of intervening on step A upon another candidate step $B \in \mathcal{C}_{step}$, we define an impact-aware suspiciousness delta $\Delta_A(B)$ inspired by quantitative causal impact analysis (Jung et al., 2022). For each candidate $B \in \mathcal{C}_{step}$ reachable in the perturbed trajectory, the update is formulated as:

$$\Delta_A(B) = \alpha_A \cdot \rho_{A,B} \cdot \left[\delta_A(B) + \omega_A(B) + \pi_A(B) \right], \quad (2)$$

$$\forall B \in (\mathcal{C}_{step} \setminus \{A\}) \cap \tau'$$

where $\alpha_A \in [0, 1]$ denotes the *global task completion*, measuring the overall task progress after correcting A (via an LLM judge) and thus reflecting the empirical fidelity of the replay. $\rho_{A,B}$ denotes the *causal strength* between steps A and B , which scales the update based on their topological distance within the HCG. $\Delta_A(B)$ integrates three complementary dimensions of causal impact.

(1) Suspiciousness Shift ($\delta_A(B)$). It quantifies the individual suspiciousness change of step B induced by the intervention at A . Specifically, it is computed as the difference between the suspiciousness scores of B in the counterfactual trajectory τ' and the original trajectory τ : $\delta_A(B) = S_B(\tau') - S_B(\tau)$, where $S_B(\cdot)$ denotes the suspiciousness score produced by the static causal attribution module.

Intuitively, if correcting step A significantly reduces the static suspiciousness score of step B in the counterfactual trace τ' , it suggests that the error at B is likely a propagated symptom of A , justifying a penalty to lower its ranking. Otherwise, persistent suspiciousness implies B is an independent root cause, justifying a reward to promote its priority in the next ranking cycle.

(2) Intra-subtask Consistency ($\omega_A(B)$). Steps within the same subtask typically share a common logical objective and execution context. To mitigate isolated suspiciousness bias, we introduce an *intra-subtask consistency* term, defined as the expected suspiciousness shift of peer candidate steps within the same subtask as B :

$$\omega_A(B) = \kappa_A \cdot \mathbb{E}_X [\delta_A(X)], \quad (3)$$

$$X \in (\mathcal{C}_{step} \setminus \{A\}) \cap \mathcal{N}(B)$$

where $\kappa_A \in [0, 1]$ denotes the *local subtask completion*, measuring the degree of subtask completion after correcting A (assessed by an LLM judge) and thus reflecting the confidence in the subtask execution. $\mathbb{E}_X [\delta_A(X)]$ denotes the expected suspiciousness shift over candidate steps X that belong to the same subtask neighborhood as B .

Intuitively, this term enforces local consistency. Given an intervention at A , B 's suspiciousness is adjusted in alignment with its peer steps, preventing isolated outliers from skewing the ranking.

(3) Causal Propagation ($\pi_A(B)$). Errors in MASs often propagate along directed causal edges. To capture the transitive impact of an intervention at step A on its downstream causal descendants in the HCG, we introduce a *causal propagation* term, defined as the expected suspiciousness shift over all downstream candidate steps along the error propagation chain:

$$\pi_A(B) = \mathbb{E}_X [\delta_A(X)], \quad (4)$$

$$X \in (\mathcal{C}_{step} \setminus \{A\}) \cap \mathcal{D}(A)$$

where $\mathbb{E}_X [\delta_A(X)]$ is the expected suspiciousness shift over downstream causal descendants.

Intuitively, this term captures whether correcting A consistently changes the suspiciousness of its downstream steps. If multiple descendants exhibit reduced suspiciousness after correcting A , it suggests that their errors were propagated from A , justifying a penalty that lowers their rankings in the next iteration.

Iterative Suspiciousness Update. After each dynamic verification cycle, SynDebug updates

suspiciousness scores using the newly obtained causal evidence. For each remaining candidate $B \in \mathcal{C}_{step} \setminus \{A\}$, the score at iteration $k + 1$ is updated as $S_B^{(k+1)} \leftarrow S_B^{(k)} + \Delta_A(B)$.

This update enables adaptive re-ranking of the candidate steps. Steps whose suspiciousness decreases significantly after the intervention at A (indicated by negative values of δ , ω , or π) are demoted as propagated symptoms, while steps that remain persistently suspicious are promoted. Through this iterative feedback loop, SynDebug progressively prunes non-decisive deviations and steers the search toward the true root-cause step.

5 Experimental Design

We evaluate our framework by investigating four core Research Questions: **RQ1 (Effectiveness):** How does SynDebug compare to state-of-the-art (SOTA) debugging baselines? **RQ2 (Efficiency):** What is the computational and token overhead of SynDebug? **RQ3 (Synergy):** Do static attribution and dynamic verification exhibit synergistic effects? **RQ4 (Ablations):** How do components and backbone LLMs affect SynDebug's performance?

Target MASs and Benchmarks. We evaluate SynDebug across three diverse MASs and four representative benchmarks: (1) **Magentic-One (M1)** (Fourney et al., 2024) evaluated on *GAIA* (Levels 2–3) (Mialon et al., 2023) and *Assistant-Bench* (Yoran et al., 2024); (2) **SWE-agent** (Yang et al., 2024) on the *SWE-bench Verified* (Jimenez et al., 2024) subset; (3) **LocAgent** (Chen et al., 2025) on *Loc-Bench* (entity-level) (Chen et al., 2025). We collect execution failure trajectories from these setups to form our evaluation dataset. Detailed benchmark statistics, selection criteria, and preprocessing are provided in Appx. A.3.1.

Baselines. We compare SynDebug against six baselines spanning three paradigms: (1) *Static Attribution*: **Random** intervention, **Step-by-Step** (Zhang et al., 2025d), and **FAMAS** (Ge et al., 2025); (2) *Dynamic Debugging*: **DoVer** (Ma et al., 2025a) and **AgentDebug** (Zhu et al., 2025); (3) *Self-Correction*: **Reflexion** (Shinn et al., 2023). For static baselines, we append an oracle-guided intervention at their predicted failure steps to enable dynamic repair evaluation. Full baseline descriptions are detailed in Appx. A.3.2.

Evaluation Metrics. Following dynamic debugging literature (Ma et al., 2025a), we adopt two

Table 1: Comparison of RSR% $\uparrow$ and RS% $\uparrow$ across different Agent-Benchmark pairs.

Paradigm	Method	M1 / GAIA		M1 / AsstBench		SWE-agent / -bench		LocAgent / Loc-Bench	
		RSR	RS	RSR	RS	RSR	RS	RSR	RS
Random		2.47 (± 1.23)	1.23	0.00 (± 0.00)	0.00	4.26 (± 2.46)	2.86	19.18 (± 0.79)	18.72
Static	Step-by-Step	3.71 (± 0.71)	3.29	0.00 (± 0.00)	0.00	8.51 (± 3.69)	6.38	30.14 (± 2.37)	28.77
	FAMAS	18.52 (± 0.71)	18.11	12.91 (± 3.72)	10.75	14.89 (± 3.25)	12.06	31.51 (± 2.09)	29.22
Dynamic	DoVer	27.16 (± 1.43)	26.34	16.13 (± 1.86)	15.05	38.29 (± 2.13)	36.17	56.16 (± 2.09)	53.88
	AgentDebug	29.63 (± 1.43)	28.81	9.68 (± 1.86)	8.61	38.29 (± 3.25)	34.75	57.53 (± 2.09)	55.25
Self-Reflect	Reflexion	9.88 (± 0.71)	9.47	3.23 (± 0.00)	3.23	10.64 (± 1.23)	9.93	35.62 (± 0.79)	34.71
SynDebug ($\pm$std.)		49.38 (± 0.71)	48.97	29.03 (± 1.86)	27.96	40.43 (± 1.23)	39.01	61.64 (± 1.37)	60.27

metrics: (1) **Repair Success Rate (RSR)**: The ratio of resolved failure instances after intervention at the identified failure step. (2) **Repair Stability (RS)**: The average success probability across K independent trials ($K = 3$) to account for LLM stochasticity, formulated as $RS = \frac{1}{KN} \sum_{i=1}^N S_i$, where N is the total instances and $S_i \in \{0, \dots, K\}$ is the success count for instance i .

Experimental Setup. SynDebug is implemented in Python 3.11 with *GPT-5.1* as the default backbone LLM, also used by all LLM-based baselines for fair comparison. Dynamic verification is capped at 5 iterations, and results are averaged over three independent trials. Setup and hardware details and the RQ3/RQ4 configurations are provided in Appx. A.3.3.

6 Results

6.1 RQ1: Effectiveness

Table 1 summarizes the debugging performance across the four configurations. SynDebug outperforms all baselines in both Repair Success Rate (RSR) and Repair Stability (RS). Random and Step-by-Step yield <4% RSR on GAIA because they cannot capture long-range dependencies, while the spectrum-based FAMAS improves to 18.52% but still lags behind SynDebug. This performance gap indicates that purely statistical correlations between execution steps and failures are insufficient for diagnosing complex agent behaviors. By contrast, SynDebug leverages HCG-based causal attribution to model cross-step dependencies, identifying more precise failure candidates.

Dynamic approaches like *DoVer* and *AgentDebug* achieve stronger results by verifying candidate failures through execution-based interventions; however, SynDebug yields substantial improvements on all benchmarks. Specifically, on

GAIA, SynDebug improves the RSR to 49.38% (vs. 29.63% for *AgentDebug*), and on AsstBench to 29.03% (vs. 16.13% for *DoVer*). Its 40.43% and 61.64% RSR on SWE-bench and Loc-Bench further show that structured static attribution guides dynamic verification more effectively than unguided runtime exploration. *Reflexion* performs poorly (e.g., 9.88% RSR on GAIA), indicating that agents struggle to reliably self-diagnose failures without dedicated external debugging mechanisms. Finally, SynDebug also maintains high RS with low variance (48.97% on GAIA and 60.27% on Loc-Bench), demonstrating stable debugging outcomes despite LLM stochasticity.

Two-sided Welch’s t -tests find significant differences in 22 of 24 comparisons ($p < 0.05$); only *DoVer* and *AgentDebug* on SWE-bench are not significant (complete p -values: Appx. A.4, Table 8).

6.2 RQ2: Efficiency

In terms of efficiency, as shown in Table 2, lightweight baselines such as *Random*, the one-pass static method *Step-by-Step*, and the runtime reflection approach *Reflexion* naturally exhibit lower latency and token consumption due to their minimal reasoning and verification overhead. Compared with replay-heavy approaches (e.g., *FAMAS*), SynDebug achieves substantially lower computational overhead. For example, on *M1 / GAIA*, SynDebug consumes 554.5k tokens, which is less than half of the spectrum-based *FAMAS* (1290.3k), while also reducing execution time from 4411.1s to 2822.0s.

A similar trend is observed across other benchmarks. Compared with dynamic debugging baselines (*DoVer* and *AgentDebug*), SynDebug exhibits comparable efficiency. Despite its iterative static-dynamic workflow, the overall computational cost remains similar to these dynamic methods across most settings. This efficiency stems from the HCG-

Table 2: Average time (s ↓) and token (k ↓) consumption per instance across Agent-Benchmark pairs.

Paradigm	Method	M1 / GAIA		M1 / AsstBench		SWE-agent / -bench		LocAgent / Loc-Bench	
		Time	Tokens	Time	Tokens	Time	Tokens	Time	Tokens
Random		985.1	426.0	749.9	401.1	534.1	427.3	43.7	42.6
Static	Step-by-Step	1038.5	466.1	800.8	430.9	682.8	821.7	63.8	43.7
	FAMAS	4411.1	1290.3	3560.8	1245.4	2105.7	1249.5	338.3	154.4
Dynamic	DoVer	2232.8	477.5	1061.8	376.1	1053.4	562.3	61.3	51.9
	AgentDebug	1527.6	534.7	1637.9	558.6	722.4	973.9	75.8	46.0
Self-Reflect	Reflexion	994.9	440.0	760.3	416.8	704.8	821.5	55.3	33.0
SynDebug		2822.0	554.5	3489.4	614.2	1224.3	583.8	92.6	123.4
		(±61.8)	(±9.0)	(±89.2)	(±13.0)	(±34.8)	(±11.6)	(±2.7)	(±4.4)

Table 3: RSR% ↑ and RS% ↑ of isolated static or dynamic debugging vs. our combined framework.

Agent / Bench.	Metric	Static-Only	Dynamic-Only	SynDebug
M1 / GAIA	RSR	24.69	29.63	49.38
	RS	23.05	28.81	48.97
M1 / AsstBench	RSR	6.45	12.91	29.03
	RS	5.38	11.83	27.96
SWE-agent / -bench	RSR	23.41	31.91	40.43
	RS	21.28	29.79	39.01
LocAgent / Loc-Bench	RSR	34.25	46.58	61.64
	RS	33.79	45.21	60.27

based causal attribution, which avoids blind exploration of less suspicious steps. Given the substantial gains in debugging effectiveness (RQ1), the additional overhead of SynDebug represents an acceptable trade-off for precise root-cause identification in complex MAS executions.

We further examine how computational cost is distributed across five execution-ordered pipeline stages, whose definitions and complete results are provided in Appx. A.5. Root Cause and Patch, the two final reasoning-intensive stages, together account for 79.3%–82.5% of token cost and 88.7%–97.2% of runtime. This concentration reflects the cost of reasoning over complete trajectories and causal structures, generating corrective patches, and replaying or retrying execution.

6.3 RQ3: Synergy Analysis

Table 3 presents the performance of SynDebug’s isolated phases compared to the integrated framework. The *Static-Only* configuration achieves decent performance (e.g., 24.69% RSR on GAIA and 34.25% on Loc-Bench), outperforming static baselines in RQ1. This shows that HCG-based static attribution effectively identifies suspicious steps

from complex execution logs. However, static attribution alone cannot verify whether the identified steps cause the failure, particularly in long trajectories where multiple candidates appear plausible.

The *Dynamic-Only* configuration (which utilizes a randomized initial ranking) consistently outperforms the *Static-Only* setting (e.g., reaching 29.63% RSR on GAIA and 46.58% on Loc-Bench). This indicates that execution-based verification is effective for confirming failure causes by observing behavioral changes after interventions. Nevertheless, without structured static guidance, the dynamic process must explore candidate steps with weaker priors, reducing localization efficiency.

Integrating both phases in the full SynDebug framework yields substantial improvements across all benchmarks. For example, on GAIA, the RSR increases to 49.38% (vs. 24.69% *Static-Only* and 29.63% *Dynamic-Only*), and on AsstBench to 29.03% (vs. 6.45% and 12.91%, respectively). These results demonstrate a clear synergy: the static stage narrows the search space by prioritizing suspicious steps, while the dynamic stage verifies and refines these hypotheses through impact-aware interventions, enabling SynDebug to efficiently converge to the true root cause.

6.4 RQ4: Ablations

Table 4 shows that removing any core component degrades performance. The larger drops without *S-M1* (*HCG Construction*) or *D-M2* (*Suspiciousness Update*) highlight the importance of structured causal representation and impact-aware recalibration, while the remaining variants confirm the contributions of specialized backtracking and causal attribution, as well as adaptive intervention guided by failure semantics.

To assess reliability and error propagation of

Table 4: Impact of key components and backbone LLMs on RSR% $\uparrow$ and RS% $\uparrow$.

Type	Variant	M1 / GAIA		M1 / AsstBench		SWE-agent / -bench		LocAgent / Loc-Bench	
		RSR	RS	RSR	RS	RSR	RS	RSR	RS
Component Impact	w/o S-M1	39.51	38.27	12.91	10.75	23.41	21.99	49.32	47.95
	w/o S-M2	43.21	42.39	16.13	15.05	29.79	27.66	53.42	52.05
	w/o D-M1	44.44	43.21	19.35	16.13	31.91	30.50	50.68	47.95
	w/o D-M2	43.21	41.98	16.13	13.98	27.66	26.24	56.16	54.34
Base LLM Generalizability	<i>Gemini-3</i>	51.85	51.44	32.26	31.18	38.29	36.17	57.53	56.16
	<i>Claude 4.5 Sonnet</i>	54.32	53.51	35.48	34.41	42.56	41.13	63.01	61.19
	<i>DeepSeek-V3</i>	41.98	40.74	22.58	19.35	31.91	29.08	50.68	48.86
	<i>Kimi-K2</i>	48.15	47.74	25.81	24.73	36.17	34.04	53.42	50.68
SynDebug (GPT-5.1)		49.38	48.97	29.03	27.96	40.43	39.01	61.64	60.27

Table 5: Cumulative stage-wise human replacement on 20 failed trajectories; gains vs. Original.

Setting	Repaired	RSR
Original	11/20	55%
Gold Subtask Nodes	11/20	55% (+0 pp)
Gold Agent Nodes	11/20	55% (+0 pp)
Gold Edges	12/20	60% (+5 pp)
Gold Root Cause	13/20	65% (+10 pp)
Gold Patch	14/20	70% (+15 pp)

Table 6: Same-backbone comparison of overall RSR% $\uparrow$ across all evaluated instances.

Backbone	SynDebug	AgentDebug	Δ
GPT-5.1	48.71	37.50	+11.21
Gemini-3	48.28	36.64	+11.64
Claude 4.5 Sonnet	52.16	40.09	+12.07
DeepSeek-V3	40.09	29.74	+10.35
Kimi-K2	44.40	33.62	+10.78

LLM-generated intermediate outputs, we conduct a stage-wise human-replacement study on 20 failed trajectories; the stages and annotation protocol are detailed in Appx. A.5 and Appx. A.7, respectively.

In the stage-wise human-replacement study, RSR increases from 55% to 70% (Table 5). The first two replacements produce no change, whereas replacing Edges, Root Cause, and Patch each adds 5 percentage points over the preceding setting. These results suggest that SynDebug’s reliability arises from combining causal attribution with dynamic verification across the debugging pipeline, rather than being determined by any single LLM-generated output; nevertheless, errors that persist across stages can still accumulate and reduce final repair performance. To further identify where the remaining failures arise, we analyze the nine trajectories not repaired by the Original pipeline. Seven first deviate from the human references at the Root Cause or Patch stage, indicating that the remaining difficulties primarily arise when analysis is converted into final repair. As the final decision-

making stages of the pipeline, errors introduced at Root Cause or Patch have a direct impact on the final repair outcome and leave less room for subsequent correction. Detailed failure-stage results are reported in Appx. A.7, Table 12.

To separate framework gains from backbone capability, Table 6 compares SynDebug with the strongest baseline, *AgentDebug*, under the same backbone. To aggregate results across the four Agent-Benchmark settings, we compute overall RSR as the proportion of successfully repaired instances among all evaluated instances. SynDebug’s overall RSR ranges from 40.09% to 52.16% across backbone LLMs and consistently exceeds *AgentDebug* by 10.35–12.07 percentage points. Thus, backbone choice affects absolute performance but not SynDebug’s consistent advantage; the default-backbone rationale is provided in Appx. A.3.3.

7 Conclusion

Debugging LLM-based Multi-Agent Systems (MASs) is challenged by stochastic reasoning, complex interactions, and unstructured traces. Existing debuggers rely on either isolated static analysis or dynamic verification, while we introduced SynDebug, a synergistic framework that bridges both. SynDebug reconstructs execution topologies using a *Hierarchical Causal Graph* to prioritize root-cause candidates, and refines these rankings through targeted interventions. Evaluation across three MAS frameworks and four benchmarks shows that SynDebug outperforms six baselines, achieving higher repair success and stability with competitive overhead.

Limitations

While SynDebug demonstrates promising performance in debugging multi-agent systems, several limitations should be noted:

First, our framework relies on LLMs for trajectory parsing, causal reasoning, and the synthesis of virtual oracles, which introduces a potential vulnerability to LLM hallucinations. To assess and mitigate this risk, we conduct a targeted evaluation of our static attribution phase against human-annotated datasets in Appx. A.6, confirming high step-level accuracy against manually verified ground truth.

Second, reliably evaluating debugging effectiveness is inherently challenging due to the stochasticity of LLM-based MASs. To mitigate this challenge, we employ widely adopted metrics, repeat experiments across three independent trials, and adopt architecture-aware replay strategies; detailed settings and replay-fidelity results are reported in Appx. A.8 and Table 13. Although dynamic verification does not require token-level identity between the replayed and original trajectories, it still depends on restoring an execution context and failure outcome consistent with the original failure. Uncontrolled external states, tool outputs, and asynchronous scheduling may therefore reduce the reliability of dynamic evidence. Moreover, iterative dynamic verification incurs non-trivial computational costs.

Finally, while we evaluate SynDebug across three MAS frameworks, four benchmarks, and multiple backbone LLMs, the generalizability of our findings to even more complex agentic protocols or broader task domains remains to be further explored in future work.

We provide a comprehensive discussion regarding these limitations and associated threats to internal, construct, and external validity in Appx. A.9.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No. 62232016, the Basic Research Program of IS-CAS under Grant Nos. ISCAS-JCZD-202405 and ISCAS-JCZD-202304, and the Innovation Team 2024 of ISCAS under Grant No. 2024-66.

References

Anonymous. 2026. Traceelephant: Seeing the whole elephant for failure attribution in llm-based multi-agent systems. <https://github.com/TraceElephant/TraceElephant>.

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*.

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)*, pages 2846–2857. IEEE.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727.

Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fournery, Jack Gerrits, Erkang Zhu, and Saleema Amershi. 2025. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–15.

Lutfi Eren Erdogan, Hiroki Furuta, Sehoon Kim, Nicholas Lee, Suhong Moon, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks. In *Forty-second International Conference on Machine Learning*.

Adam Fournery, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, and 1 others. 2024. Magentic-one: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*.

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

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiwu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*.

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

- Yonghan Jung, Shiva Kasiviswanathan, Jin Tian, Dominik Janzing, Patrick Blöbaum, and Elias Bareinboim. 2022. On measuring causal contributions via do-interventions. In *International Conference on Machine Learning*, pages 10476–10501. PMLR.
- Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008.
- Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2025a. Dover: Intervention-driven auto debugging for llm multi-agent systems. *arXiv preprint arXiv:2512.06749*.
- Xuyan Ma, Xiaofei Xie, Yawen Wang, Junjie Wang, Boyu Wu, Mingyang Li, and Qing Wang. 2025b. Diagnosing failure root causes in platform-orchestrated agentic systems: Dataset, taxonomy, and benchmark. *arXiv preprint arXiv:2509.23735*.
- Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.
- Pranav Putta, Edmund Mills, Naman Garg, Sumet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. 2024. Agent q: Advanced reasoning and learning for autonomous ai agents. *arXiv preprint arXiv:2408.07199*.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.
- Linxin Song, Jiale Liu, Jieyu Zhang, Shaokun Zhang, Ao Luo, Shijian Wang, Qingyun Wu, and Chi Wang. 2024. Adaptive in-conversation team building for language model agents. *arXiv preprint arXiv:2405.19425*.
- Junjie Wang, Yawen Wang, Mengzhuo Chen, Xiaofei Xie, Chunyang Chen, Fangwen Mu, Zhe Liu, and Qing Wang. 2026. A survey for llm agent trajectory analysis: From failure attribution to enhancement.
- Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering*, (4):352–357.
- Alva West, Yixuan Weng, Minjun Zhu, Zhen Lin, Zhiyuan Ning, and Yue Zhang. 2025. Abduct, act, predict: Scaffolding causal inference for automated failure attribution in multi-agent systems. *arXiv preprint arXiv:2509.10401*.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.
- Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. 2024. Assistantbench: Can web agents solve realistic and time-consuming tasks? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8938–8968.
- Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10.
- Wenhao Zeng, Yaoning Wang, Chao Hu, Yuling Shi, Chengcheng Wan, Hongyu Zhang, and Xiaodong Gu. 2025. Pruning the unsurprising: Efficient code reasoning via first-token surprisal. *arXiv preprint arXiv:2508.05988*.
- Cheng Zhang, Juyuan Yang, Dacong Yan, Shengqian Yang, and Yuting Chen. 2013. Automated breakpoint generation for debugging. *JOURNAL OF SOFTWARE*, 8(3):603.
- Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. 2025a. Agentracer: Who is inducing failure in the llm agentic systems? *arXiv preprint arXiv:2509.03312*.
- Heng Zhang, Yuling Shi, Xiaodong Gu, Haochen You, Zijian Zhang, Lubin Gan, Yilei Yuan, and Jin Huang. 2025b. Graphtracer: Graph-guided failure tracing in llm agents for robust multi-turn deep search. *arXiv preprint arXiv:2510.10581*.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025c. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*.
- Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025d. Which agent causes task failures and

when? on automated failure attribution of LLM multi-agent systems. In *Forty-second International Conference on Machine Learning*.

Sipeng Zheng, jiazheng liu, Yicheng Feng, and Zongqing Lu. 2024. Steve-eye: Equipping LLM-based embodied agents with visual perception in open worlds. In *The Twelfth International Conference on Learning Representations*.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, and 1 others. 2025. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*.

A Appendix

A.1 Detailed Preliminaries and Formalisms

A.1.1 LLM-based Multi-Agent Systems

LLM-based MASs tackle complex tasks by orchestrating multiple specialized agents (Li et al., 2023; Hong et al., 2023; Wu et al., 2024). Existing frameworks can be broadly categorized into two paradigms. The first comprises *hand-crafted systems* (e.g., AutoGen (Wu et al., 2024), MetaGPT (Hong et al., 2023), and ChatDev (Qian et al., 2024)), where agent roles and interaction workflows are manually designed through predefined standard operating procedures. The second includes *automatically optimized systems* (e.g., AgentPrune (Zeng et al., 2025) and AFlow (Zhang et al., 2025c)), which dynamically discover or refine agent roles and collaboration topologies to improve task performance.

Formally, the MAS is defined as $\mathcal{M} = \langle \mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$, where $\mathcal{N} = \{i\}_{i=1}^N$ denotes the set of agents, $\mathcal{S}$ the system state space, $\mathcal{A}$ the action space, and $\mathcal{P}$ the state transition function. At timestep t , an active agent i_t executes an action $a_t \in \mathcal{A}$, transitioning the system from state s_t to s_{t+1} according to $\mathcal{P}(s_{t+1} | s_t, a_t)$. This interaction produces a trajectory $\tau = \{s_t, (i_t, a_t)\}_{t=0}^T$, with execution outcome $Z(\tau) \in \{0, 1\}$, where 1 denotes failure.

Serial vs. Parallel MASs. In *serial MASs* (Hong et al., 2023; Wu et al., 2024), agents act sequentially, and exactly one agent performs an action at each timestep. In contrast, *parallel MASs* (Fourney et al., 2024; Song et al., 2024) allow multiple agents to interact through asynchronous messages or shared contexts, where the actual execution order depends on runtime scheduling and communication events.

A.1.2 Failure Debugging Formulation

Given a failed trajectory τ ($Z(\tau) = 1$), the goal of *failure debugging* is to identify the execution step t that constitutes the root cause of the failure. We first define the *decisive error indicator*:

$$\Delta_t(\tau) = \mathbb{I}(Z(\tilde{\tau}^{(t)}) = 0) \quad (5)$$

where $\tilde{\tau}^{(t)}$ denotes the counterfactual trajectory in which the action at step t is corrected while all other steps remain unchanged. $\Delta_t = 1$ indicates that correcting step t leads to a successful execution ($Z(\tilde{\tau}^{(t)}) = 0$), identifying step t as a decisive error, and 0 otherwise.

In practice, a trajectory may contain multiple decisive errors due to error propagation. The root cause is defined as the earliest decisive error in the temporal sequence (Zhang et al., 2025d):

$$t^* = \arg \min_{t: \Delta_t(\tau)=1} t \quad (6)$$

Consequently, the debugging objective is to localize the ground-truth root-cause step t^* by analyzing the initial failed trajectory τ or by reproducing the failure through trajectory re-execution.

A.2 Extended Related Work

A.2.1 Static Failure Attribution

Static approaches primarily rely on post-hoc log analysis to infer root causes without runtime re-execution.

Pattern and Spectrum-based Analysis. Early efforts, such as the *Who&When* benchmark (Zhang et al., 2025d), established the *LLM-as-a-Judge* paradigm using strategies like All-at-Once and Binary Search to localize faults. To move beyond simple prompting, FAMAS (Ge et al., 2025) adapted spectrum-based fault localization (SBFL), calculating “suspiciousness” scores for execution steps. While efficient for broad localization, these methods often capture statistical correlations rather than causal explanations and struggle with the reasoning depth required for long-context logs.

Structured and Causal Reasoning. To capture complex error propagation, recent works introduce structural abstractions. *ECHO* (Banerjee et al., 2025) employs a hierarchical context representation and consensus voting to analyze interaction traces, while *A2P* (West et al., 2025) explicitly models causality using Shapley values or abductive reasoning. Despite their depth, these static approaches

Table 7: Statistics of the benchmarks.

Dataset	Domain	Characteristics	Fail/Inst
GAIA	Gen.	Multi-modal tool use	81/112
AsstBench	Web	Long-horizon reasoning	31/33
SWE-bench	SE	GitHub issue fixing	47/500
Loc-Bench	Loc.	Code graph search	73/560

remain “untested hypotheses” that lack validation against the inherent stochasticity of agent environments.

Specialized Tracer Models. A parallel line of research involves training dedicated models for diagnosis. *AgentTracer* (Zhang et al., 2025a) and *GraphTracer* (Zhang et al., 2025b) fine-tune 8B-parameter models on synthetic failure datasets. Although these models achieve high accuracy in specific domains, they incur significant upfront costs for data generation and face generalization risks when applied to unseen agent frameworks.

A.2.2 Dynamic Debugging and Verification

Dynamic debugging paradigms move beyond static analysis by verifying hypotheses through runtime interventions and replays.

Intervention-driven Debugging. *DoVer* (Ma et al., 2025a) introduces a “Do-then-Verify” pipeline that validates failure hypotheses by editing messages or plans and re-executing the system. Similarly, *AgentDebug* (Zhu et al., 2025) utilizes fine-grained error taxonomies to guide targeted re-rollouts. While these methods provide high-fidelity confirmation, they suffer from high computational overhead (token/time cost) due to the lack of an efficient search strategy. Furthermore, the inherent non-determinism of LLMs often makes dynamic verification unstable and difficult to reproduce.

Interactive Debugging. Human-in-the-loop tools like *AGDebugger* (Epperson et al., 2025) provide manual rewind and edit capabilities. To automate this, *AgentFail* (Ma et al., 2025b) proposes combining candidate identification with counterfactual verification.

A.3 Detailed Experimental Settings

A.3.1 Benchmark Details and Preprocessing

To ensure a rigorous and diverse evaluation, we utilize three multi-agent systems paired with four representative benchmarks:

- **Magentic-One (M1)** (Fourney et al., 2024): A generalist MAS designed to orchestrate specialized agents for complex web-based and file-manipulation tasks. We evaluate its performance on *GAIA* (Mialon et al., 2023) and *Assistant-Bench* (Yoran et al., 2024), which require high-level planning and tool-use. For *GAIA* dataset, since Level 1 tasks are relatively simple and rarely trigger failures in MASs, we evaluate SynDebug on the more challenging subsets, Levels 2 and 3.
- **SWE-agent** (Yang et al., 2024): A specialized software engineering agent that interacts with a computer-based interface to resolve real-world GitHub issues. It is evaluated on the *SWE-bench* (Jimenez et al., 2024) *Verified* subset, a rigorous testbed for autonomous program repair.
- **LocAgent** (Chen et al., 2025): A graph-guided framework that addresses code localization by modeling codebases as directed heterogeneous graphs. We utilize the *Loc-Bench* (Chen et al., 2025) suite to assess its multi-hop reasoning capabilities over complex code dependencies. We focus on the most challenging setting, entity-level localization, and use Acc@1 to determine task completion.

A.3.2 Baseline Descriptions

We compare SynDebug against the following baselines:

- **Random:** A naive method that randomly selects an execution step for intervention, serving as a lower bound.
- **Step-by-Step** (Zhang et al., 2025d): The representative strategy in the *Who&When* benchmark. It utilizes an LLM-based judge to scrutinize execution logs turn-by-turn to identify the earliest decisive error.
- **FAMAS** (Ge et al., 2025): A SOTA spectrum-based method that finds the root-cause step by analyzing statistical activation patterns across successful and failed traces.
- **DoVer** (Ma et al., 2025a): A SOTA intervention-driven framework following a “Do-then-Verify” paradigm. It validates failure hypotheses by actively modifying instructions/plans and re-executing the system.

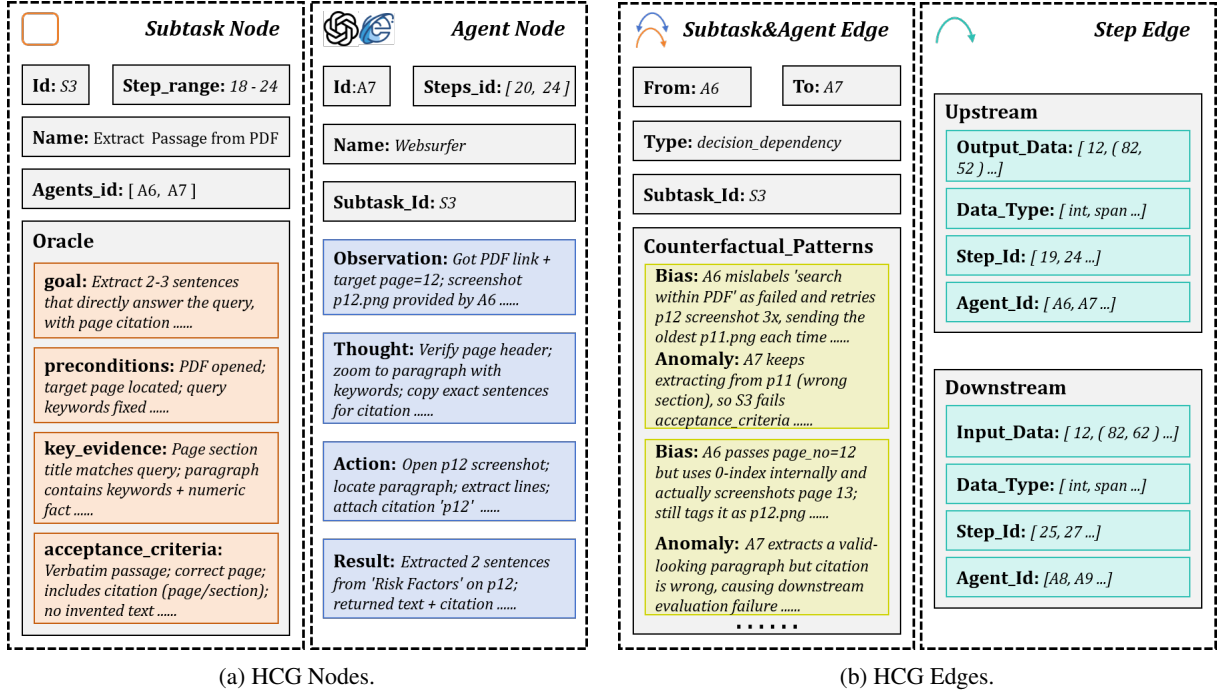


Figure 3: Detailed attributes of the hierarchical causal graph.

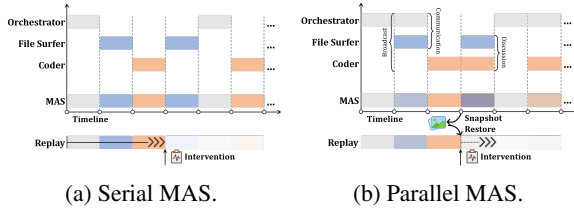


Figure 4: Two types of MASs and replay strategies.

- **AgentDebug** (Zhu et al., 2025): A three-stage pipeline that classifies module-level failures via an *AgentErrorTaxonomy*, localizes critical errors through counterfactual testing, and iteratively repairs them via feedback-guided re-rollouts.
- **Reflexion** (Shinn et al., 2023): It enables agents to generate linguistic feedback on past attempts to autonomously refine subsequent reasoning and actions.

A.3.3 Implementation and Setup Details

Infrastructure. All empirical evaluations were performed on a server equipped with an Intel i7-10700 CPU, 32GB of RAM, and an NVIDIA TITAN RTX GPU.

Baseline Executions. For all baselines, we prioritize using their official implementations and strictly follow the default hyperparameter settings reported in the original papers. For baselines without publicly available code (e.g., FAMAS (Ge et al.,

2025)), we faithfully re-implement them according to the descriptions provided in their papers.

RQ3 Synergy Configurations. To isolate the contributions of static and dynamic modules, we configure two variants:

- **Static-Only:** Retains the static causal attribution module but removes dynamic verification. It employs a single-pass *Oracle-based Intervention* on the top-ranked candidate to verify the initial static results.
- **Dynamic-Only:** Preserves dynamic verification and update logic but replaces the HCG-based static results with a randomized initial ranking of root causes.

RQ4 Ablation Configurations. To assess how individual components affect performance, we evaluate four variants: (1) *w/o S-M1* removes HCG construction and reduces the static stage to an All-at-Once attribution over raw logs; (2) *w/o S-M2* retains the HCG but replaces specialized backtracking and attribution with direct prompting over the graph; (3) *w/o D-M1* replaces adaptive intervention with a randomized strategy that ignores failure semantics; and (4) *w/o D-M2* disables impact-aware suspiciousness updating and verifies candidates according to the initial static ranking.

For backbone analysis, we evaluate SynDebug with proprietary models (*GPT-5.1*, *Gemini-3*, and *Claude 4.5 Sonnet*) and open-weight models (*DeepSeek-V3* and *Kimi-K2*) under consistent configurations. To separate framework gains from backbone capability, we additionally evaluate the strongest baseline, *AgentDebug*, using the same backbone in each comparison. We aggregate results across the four Agent-Benchmark settings using overall RSR, defined as the proportion of successfully repaired instances among all evaluated instances. Equivalently, $\text{RSR}_{\text{overall}} = \sum_c n_c \text{RSR}_c / \sum_c n_c$, where n_c is the number of evaluated instances in setting c .

Although *Claude 4.5 Sonnet* achieves the highest RSR across all four Agent-Benchmark settings, we adopt *GPT-5.1* as the default backbone after considering performance, efficiency, and cost jointly. Compared with *GPT-5.1*, *Claude 4.5 Sonnet* consumes approximately $2.4\times$ as many input tokens and $1.5\times$ as many output tokens, while its inference speed is approximately two-thirds that of *GPT-5.1*.

A.4 Statistical Significance of RQ1 Results

We conduct two-sided Welch’s t -tests between SynDebug and each baseline using the per-run repair rates from three independent trials. Table 8 reports the resulting p -values; 22 of the 24 comparisons are statistically significant at $p < 0.05$.

A.5 Component-Level Cost Analysis

We divide the debugging pipeline into five execution-ordered stages according to their intermediate outputs. *Subtask Nodes / Virtual Oracles* covers task decomposition and subtask-level oracle synthesis; *Agent Nodes / OTAR* covers atomic agent-node construction and structured trajectory summarization; *Edges / Counterfactual Pattern* covers causal-dependency reconstruction; *Root Cause* covers final causal attribution from the ranked candidates; and *Patch / Intervention Output* covers correction generation and the resulting dynamic intervention.

To examine how computational cost is distributed across the five stages, we conduct three independent runs on the complete datasets and measure the average per-instance token consumption and runtime of each stage, reporting mean $\pm$ sample standard deviation. The total rows in Table 9 report the mean summed cost. Root Cause and Patch together account for 79.3%–82.5% of token cost and 88.7%–97.2% of runtime, showing that

computational cost is concentrated in these two stages. This concentration reflects the cost of reasoning over complete trajectories and causal structures, generating corrective patches, and replaying or retrying execution.

Table 10 compares failed and successful cases over the complete datasets after normalizing each stage’s successful-case cost to 100%. Failed cases incur modestly higher Root Cause and Patch costs, consistent with more complex or ambiguous causal chains that require additional reasoning and retries. In contrast, Subtask Nodes / Virtual Oracles and Edges / Counterfactual Pattern cost slightly less in failed cases, while Agent Nodes / OTAR remains nearly unchanged. Thus, failures are associated with higher costs in the reasoning-intensive back-end stages rather than uniformly higher costs throughout the pipeline.

A.6 Evaluation on Human-Annotated Datasets

To further assess the effectiveness, we benchmark SynDebug on two public datasets with human-annotated ground truth: *Who&When* (Zhang et al., 2025d) and *TraceElephant* (Anonymous, 2026). Due to the absence of runtime contexts and the stochastic nature of MAS execution, end-to-end dynamic reproduction on these datasets is infeasible. Therefore, we focus on rigorously evaluating the static phase using their high-quality manual annotations. Following prior work (Zhang et al., 2025d; Anonymous, 2026), we measure *Step-level Accuracy*, defined as the proportion of cases where the top-1 suspicious step identified by the static phase exactly matches the human-labeled root-cause step.

We compare SynDebug with three baselines: *Random* selection, *Step-by-Step* (Zhang et al., 2025d), and *FAMAS* (Ge et al., 2025).

As shown in Table 11, SynDebug consistently outperforms the baselines on most subsets (6/7), with only one minor exception. In particular, SynDebug surpasses the naive *Random* strategy and the direct prompting baseline (*Step-by-Step*). The spectrum-based method *FAMAS* achieves the best result on the hand-crafted subset, where longer trajectories enable more reliable statistical spectrum analysis through costly replays. Nevertheless, SynDebug’s strong performance across all other settings suggests that causal reasoning offers a more efficient alternative, delivering robust performance with zero replay cost. These results concern only SynDebug’s static attribution phase and comple-

Table 8: Two-sided Welch’s t -test p -values for comparing SynDebug with each baseline. Bold values are not statistically significant at $p < 0.05$.

Comparison	GAIA	AsstBench	SWE-bench	Loc-Bench
SynDebug vs. Random	6.035×10^{-6}	0.001476	0.000210	1.311×10^{-5}
SynDebug vs. Step-by-Step	1.579×10^{-7}	0.001476	0.001979	0.000183
SynDebug vs. FAMAS	7.567×10^{-7}	0.006002	0.001925	8.503×10^{-5}
SynDebug vs. DoVer	0.000169	0.001058	0.133583	0.015902
SynDebug vs. AgentDebug	0.000237	0.000220	0.139435	0.032199
SynDebug vs. Reflexion	2.822×10^{-7}	0.001885	8.426×10^{-6}	6.180×10^{-5}

Table 9: Average per-instance component-level token and runtime costs (mean $\pm$ sample standard deviation over three independent runs).

(a) Token Cost (k tokens)				
Stage	M1 / GAIA	M1 / AsstBench	SWE-agent / -bench	LocAgent / Loc-Bench
Subtask Nodes / Virtual Oracles	33.3 ± 1.8	36.9 ± 2.6	35.0 ± 2.3	5.9 ± 0.9
Agent Nodes / OTAR	25.0 ± 1.4	27.6 ± 2.1	26.3 ± 1.9	6.6 ± 0.8
Edges / Counterfactual Pattern	41.6 ± 2.1	43.0 ± 2.9	40.9 ± 2.6	13.0 ± 1.2
Root Cause	138.6 ± 4.3	144.3 ± 6.1	148.9 ± 5.6	49.6 ± 2.7
Patch / Intervention Output	316.1 ± 7.2	362.4 ± 10.6	332.8 ± 9.3	48.3 ± 3.0
Total (mean)	554.5	614.2	583.8	123.4
(b) Runtime (s)				
Stage	M1 / GAIA	M1 / AsstBench	SWE-agent / -bench	LocAgent / Loc-Bench
Subtask Nodes / Virtual Oracles	14.6 ± 3.4	17.2 ± 4.8	15.9 ± 3.1	2.0 ± 0.4
Agent Nodes / OTAR	26.6 ± 4.3	37.0 ± 6.6	31.8 ± 4.1	3.0 ± 0.4
Edges / Counterfactual Pattern	37.0 ± 5.7	44.8 ± 8.2	40.9 ± 5.2	5.5 ± 0.5
Root Cause	356.2 ± 20.2	402.8 ± 27.9	218.2 ± 13.7	16.0 ± 1.1
Patch / Intervention Output	2387.6 ± 57.9	2987.6 ± 83.9	917.5 ± 31.1	66.1 ± 2.4
Total (mean)	2822.0	3489.4	1224.3	92.6

Table 10: Component costs in failed cases relative to successful cases, with each successful-case cost normalized to 100%.

Stage	Token Cost Failure / Success	Runtime Failure / Success
Subtask Nodes / Virtual Oracles	96.2% / 100.0%	88.5% / 100.0%
Agent Nodes / OTAR	100.1% / 100.0%	99.9% / 100.0%
Edges / Counterfactual Pattern	96.0% / 100.0%	94.6% / 100.0%
Root Cause	102.9% / 100.0%	105.3% / 100.0%
Patch / Intervention Output	103.7% / 100.0%	103.3% / 100.0%

ment the end-to-end repair analysis in Sec. 6.4.

A.7 Stage-wise Human-Replacement and Primary Failure-Stage Analysis

Module-level ablations quantify the contributions of complete modules but do not directly reveal how errors in LLM-generated intermediate outputs affect final repair or where the remaining failures

first arise. We therefore conduct stage-wise human-replacement and primary failure-stage analyses using the five execution-ordered stages defined in Appx. A.5. The human-replacement study uses 20 failed trajectories, while the failure-stage analysis focuses on the nine trajectories not repaired by the Original pipeline.

A.7.1 Stage-wise Human-Replacement Study

To directly assess the reliability of LLM-generated intermediate outputs and their error propagation through the debugging pipeline, we sample 20 failed trajectories, with five from each benchmark. After a pilot calibration round, three MAS-experienced annotators follow a three-round rotating “primary annotator + double review” protocol: in each round, one annotator constructs the human references and the other two independently review them. The references cover Subtask Nodes / Virtual Oracles, Agent Nodes / OTAR, Edges / Counterfactual Pattern, Root Cause, and Patch / Intervention Output. Disagreements are resolved against

Table 11: Step-level Accuracy (%) $\uparrow$ of SynDebug’s static analysis phase on human-annotated datasets.

Method	Who&When		TraceElephant				
	Hand-Crafted	Alg.-Gen.	Captn./AsstBench	Captn./GAIA	M1/AsstBench	M1/GAIA	SWE-agent/-bench
Random	4.20	19.10	13.89	9.38	6.67	14.06	0.00
Step-by-Step	6.60	27.40	8.33	12.50	6.67	25.00	0.00
FAMAS	41.38	23.81	19.44	31.25	13.33	25.00	17.10
SynDebug	29.31	52.00	33.33	33.33	22.22	31.25	18.75

the trajectory evidence. Starting from the Original pipeline, we progressively replace these outputs in execution order, with each setting retaining all preceding human replacements.

As reported in Table 5, RSR increases from 55% to 70%. The first two replacements produce no change, whereas replacing Edges, Root Cause, and Patch each adds 5 percentage points over the preceding setting. These results suggest that SynDebug’s reliability arises from combining causal attribution with dynamic verification across the debugging pipeline, rather than being determined by any single LLM-generated output. The cumulative improvement nevertheless shows that errors persisting across stages can reduce final repair performance.

A.7.2 Primary Failure-Stage Analysis

To further identify where the remaining failures arise, we analyze the nine trajectories not repaired by the Original pipeline in the 20-trajectory human-replacement sample. For each trajectory, the annotators identify the earliest stage that deviates from its human reference as the primary failure stage. As shown in Table 12, Root Cause and Patch are the primary failure stages in four and three trajectories, respectively. Root Cause identifies the final root cause from the ranked candidates, while Patch generates the correction used for dynamic intervention; their outputs therefore determine the final repair outcome more directly than earlier intermediate outputs and leave less room for subsequent correction. Because this analysis covers only the nine unrepaired trajectories in the 20-trajectory sample, the distribution describes these sampled cases rather than component error rates over the complete datasets.

A.8 Replay Fidelity under Serial and Parallel MASs

We empirically assess whether the architecture-aware replay strategies in Sec. 4.2.1 can reproduce originally failed trajectories. Each failed trajectory

Table 12: Primary failure stages among the nine trajectories not repaired by the Original pipeline.

Primary Failure Stage	Count	Proportion
Subtask Nodes / Virtual Oracles	1	11.11%
Agent Nodes / OTAR	0	0.00%
Edges / Counterfactual Pattern	1	11.11%
Root Cause	4	44.44%
Patch / Intervention Output	3	33.33%
Total	9	100%

Table 13: Trajectory-level replay reproduction across serial and parallel MASs. Rates are averaged over three independent replays per failed trajectory.

MAS / Bench-mark	Replay Strategy	Failed Traj.	Reproduction Rate
SWE-agent / SWE-bench	Deterministic replay	47	97.87%
LocAgent / LocBench	Deterministic replay	73	100.0%
M1 / GAIA	Sequential-style replay	81	4.94%
M1 / GAIA	Snapshot-and-restore	81	72.84%
M1 / Assistant-Bench	Sequential-style replay	31	6.45%
M1 / Assistant-Bench	Snapshot-and-restore	31	93.55%

is independently replayed three times, and a replay is counted as reproduced only when both its restored execution context and failure outcome are consistent with the original trajectory. We report the mean reproduction rate across the three replays. For serial MASs, we fix the random seed, set the LLM temperature to 0, and replay from the trajectory start. For parallel MASs, we compare this sequential-style replay with snapshot-and-restore replay, which restores agent memory, tool states, and message events at the target step.

As shown in Table 13, deterministic replay reproduces 97.87% and 100.0% of the failed trajectories for SWE-agent and LocAgent, respectively. For the

parallel M1 system, sequential-style replay reproduces only 4.94% on GAIA and 6.45% on AssistantBench, whereas snapshot-and-restore increases these rates to 72.84% and 93.55%. Thus, restoring target-step runtime state is critical for replaying parallel MASs, although the remaining gap also shows that it does not guarantee complete reproduction.

Residual non-determinism can arise from LLM sampling, asynchronous message scheduling, external tool outputs, and mutable shared state. Beyond snapshot-and-restore, potential mitigations include record-and-replay mechanisms, caching external outputs, controlled scheduling, and estimating intervention effects over multiple stochastic replays. Extending dynamic verification to partially reproducible or highly non-deterministic environments remains important future work.

A.9 Threats to Validity

A.9.1 Internal Validity.

The primary threat to internal validity stems from SynDebug’s reliance on LLM-based components for trajectory parsing, causal reasoning, and the synthesis of *Virtual Oracles*. Potential LLM hallucinations may introduce bias into causal attribution and dynamic intervention. To mitigate it, we conduct a targeted evaluation of the static attribution phase using human-annotated datasets (*Who&When* and *TraceElephant*). As reported in Appx. A.6, SynDebug achieves high step-level accuracy against manually verified ground truth, indicating that the LLM-driven components provide reliable positive signals to the overall improvement. The stage-wise human-replacement study in Sec. A.7 further examines the reliability of LLM-generated outputs throughout the complete debugging pipeline.

A.9.2 Construct Validity.

Construct validity concerns whether our metrics—Repair Success Rate (RSR), Repair Stability (RS), and computational cost—adequately reflect debugging effectiveness. These metrics are widely adopted in automated failure attribution and debugging research (Ge et al., 2025; Ma et al., 2025a). Moreover, the dynamic phase of SynDebug offers an additional validation signal. A successful repair after a targeted intervention provides empirical evidence that the identified candidate corresponds to a true root cause. Furthermore, evaluation on human-annotated datasets helps ensure that the automated evaluation on benchmark datasets aligns with hu-

man judgments of failure attribution. The replay-reproduction study in Sec. A.8 additionally evaluates whether dynamic evidence can be recovered reliably under serial and parallel execution. Finally, to mitigate the inherent stochasticity of LLM-based MASs, we repeat all experiments across three independent trials and report aggregated results to ensure stable and reliable findings.

A.9.3 External Validity.

The generalizability of our findings may be limited by the diversity of MAS architectures and task domains. To mitigate it, we evaluate SynDebug across three representative MAS frameworks and four widely used benchmarks spanning general task-solving, program repair, and code localization. We further conduct a generalizability study across multiple backbone LLMs, including both proprietary and open-weight models, to ensure that the effectiveness of SynDebug is not tied to a specific model. Exploring more complex agentic protocols across broader domains is left for future work.