AgentDebugX: An Open-Source Toolkit for Failure Observability, Attribution, and Recovery in LLM Agents

Kunlun Zhu[1*] **Xuyan Ye**[1*] **Zhiguang Han**[1*] **Yuchen Zhao**[1] **Bingxuan Li**[1] **Weijia Zhang**[1]
Muxin Tian[2] **Xiangru Tang**[3] **Pan Lu**[4] **James Zou**[4] **Jiaxuan You**[1] **Heng Ji**[1]
[1]University of Illinois Urbana-Champaign [2]University of Toronto [3]Google [4]Stanford University
kunlunz2@illinois.edu
[*]Equal contribution.

Abstract

LLM agent failures are difficult to debug because the step where an error surfaces is often not the one that caused it. Existing observability tools replay execution traces but provide little support for identifying the root cause or translating diagnosis into recovery. We present **AgentDebugX**, an open-source debugging framework that organizes debugging as a closed loop of **Detect**, **Attribute**, **Recover**, and **Rerun**. At its core, *DeepDebug* performs multi-turn root-cause diagnosis through global trajectory understanding, structure-guided investigation, and cross-examination. On the Who&When benchmark, DeepDebug achieves the best strict attribution accuracy among the evaluated methods on both tested open-weight backbones, reaching 28.8% exact agent-and-step accuracy on `qwen3.5-9b` versus 21.7% for the strongest single-pass baseline. On GAIA, DeepDebug repairs 13 of 73 failed tasks in a single rerun, compared with 4–6 for three decoupled self-correction baselines, improving overall accuracy from 55.8% to 63.6%. AgentDebugX exposes this workflow through a Python library, CLI, web console, and installable agentic skill, and provides an opt-in Error Hub for sharing scrubbed failure–diagnosis–repair bundles and reusing them as debugging memory.

1 Introduction

Large language model (LLM) agents are increasingly deployed in settings that require long-horizon reasoning, external tool use, memory, and coordination across components. They support requests through live APIs, modify software repositories, operate graphical interfaces, and collaborate through planner–executor and multi-agent workflows (Aghzal et al., 2025; Liu et al., 2025; Li et al., 2025; Yang et al., 2025). This flexibility also makes their failures difficult to diagnose. As these systems become more capable, however, their failures become substantially harder to debug.[1]

System	Port. schema	Taxonomy	Attribution	Recovery	Error hub
AgentDebug (Zhu et al., 2025)	○	●	●	–	–
MAST (Cemri et al., 2026)	–	●	○	–	–
Who&When (Zhang et al., 2025)	–	○	●	–	–
AgentDiagnose (Ou et al., 2025)	○	○	○	–	–
AgentRx (Barke et al., 2026)	–	●	●	○	–
Langfuse (Langfuse, 2023)	○	–	–	–	–
AgentDebugX (Ours)	●	●	●	●	●

Table 1: Capability coverage versus prior work. ● = first-class, ○ = partial, – = absent. **Port. schema**: a portable, framework-agnostic trace format enabling re-analysis, comparison, and sharing (not a vendor span format). **Taxonomy**: a labelled failure-mode vocabulary. **Attribution**: localizing the responsible step/agent, not just the crash. **Recovery**: turning a diagnosis into a rerun-able fix. **Error hub**: a shareable cross-team corpus of diagnosed failures.

The central difficulty is that the step where a failure becomes visible is often not the step that caused it. An agent may return an incorrect final answer because of a planning constraint omitted much earlier, a stale memory retrieval, an invalid intermediate assumption, or an incorrect handoff between agents. The resulting symptom may surface only after many apparently reasonable actions, such as a failed tool call, an inconsistent downstream decision, or an unsupported final response. Consequently, replaying an execution trace is rarely sufficient: developers must determine which earlier decision rendered the run unsuccessful, explain why it was decisive, and translate that diagnosis into a correction that can be tested.

Existing tools address only parts of this challenge. General-purpose observability platforms provide detailed execution traces, but largely leave

[1]Project page: https://www.agentdebugx.com. Code: https://github.com/AgentDebugX/AgentDebugX. Package: https://pypi.org/project/agentdebugx/ (`pip install agentdebugx`). Screencast: https://youtu.be/ztni6w0o_l8. The project is released under the MIT license.

root-cause analysis and repair to the developer. Failure taxonomies and attribution benchmarks formalize common error modes and evaluate whether a method can identify the responsible agent or step, but are typically presented as standalone analyses rather than deployable debugging infrastructure. Self-correction methods can revise unsuccessful behavior, yet correction is considerably more reliable when the location of the underlying error is already known (Tyen et al., 2024). As summarized in Table 1, current systems therefore leave a gap between observing a failed execution, attributing its root cause, proposing an actionable repair, and verifying that repair through rerunning.

To address the gap, we introduce AgentDebugX, an open-source toolkit that closes this gap by organizing agent debugging as the iterative loop shown in Figure 1: **Detect**, **Attribute**, **Recover**, and **Rerun**. Given either a live execution or an exported log, AgentDebugX first converts framework-specific events into a portable trajectory representation. Detection identifies observable failures and associates them with structured failure modes. Attribution then traces those symptoms backward to the agent and step whose correction would most likely have prevented the failure. Recovery converts the resulting diagnosis into a concrete retry directive, and rerunning applies the approved correction from an appropriate checkpoint while retaining both the original and repaired branches for comparison. When a rerun remains unsuccessful, its new trajectory re-enters the same loop.

At the center of AgentDebugX is **DeepDebug**, a multi-turn root-cause diagnostic agent designed for failures that cannot be reliably localized through a single trace reading. DeepDebug combines a global trajectory read with a structure-guided probe—tracing handoffs in multi-agent runs or bisecting single-agent traces. It cross-examines conflicting candidates and outputs an auditable report with the responsible agent and step, supporting evidence, an explanation, and one concrete fix. A diagnosis becomes operationally useful only when it can improve the next execution. AgentDebugX therefore connects attribution directly to a policy-gated rerun loop. Its native recovery path uses DeepDebug's localized, evidence-backed correction as the retry directive, while alternative recoverers can reformulate the same diagnosis.

Beyond individual debugging sessions, AgentDebugX provides an opt-in **Error Hub** for storing scrubbed trajectory–diagnosis–repair bundles. These bundles can serve as incident records, continuous-integration regression fixtures, and reusable debugging memory. Through a shared, framework-independent format, teams can compare diagnoses across methods and versions, retrieve similar historical failures, and accumulate reviewed examples of long-tail failure modes without modifying the original execution evidence.

We evaluate the two capabilities of AgentDebugX: accurately localizing the cause of a failure and converting that diagnosis into a successful repair. On the Who&When benchmark, DeepDebug achieves the strongest attribution performance among the evaluated methods on both tested open-weight backbones. With `qwen3.5-9b`, it reaches 28.8% strict agent-and-exact-step accuracy, compared with 21.7% for the strongest single-pass baseline. On GAIA, applying DeepDebug's diagnosis in a single rerun repairs 13 of 73 trajectories initially failed by the underlying agent, compared with 4–6 repairs for three decoupled self-correction baselines, increasing overall accuracy from 55.8% to 63.6%.

2 Related Works

Existing research provides strong components for observing, diagnosing, or correcting agent failures, but rarely connects them into a unified workflow from failure detection to verified recovery. Observability platforms such as LangSmith, Langfuse, and Phoenix capture and replay detailed agent traces (Dong et al., 2024), but leave developers to determine which step was responsible, why, and how to repair the run. A parallel line formalizes agent failures through taxonomies, attribution benchmarks, and trajectory diagnostics — AgentDebug (Zhu et al., 2025), MAST (Cemri et al., 2026), Who&When (Zhang et al., 2025), TRAIL (Deshpande et al., 2025), and AgenTracer (Zhang et al., 2026). These show that even strong models struggle with root-cause localization, but are presented as standalone taxonomies, benchmarks, or attribution methods rather than deployable infrastructure for heterogeneous runtimes. The closest system, AgentDiagnose (Ou et al., 2025), provides an open toolkit for scoring trajectories along interpretable dimensions and curating training data, but does not connect step-level attribution to verified recovery or a shared incident corpus. Self-correction methods — Reflexion (Shinn et al., 2023), Self-Refine (Madaan et al., 2023), CRITIC (Gou et al.,

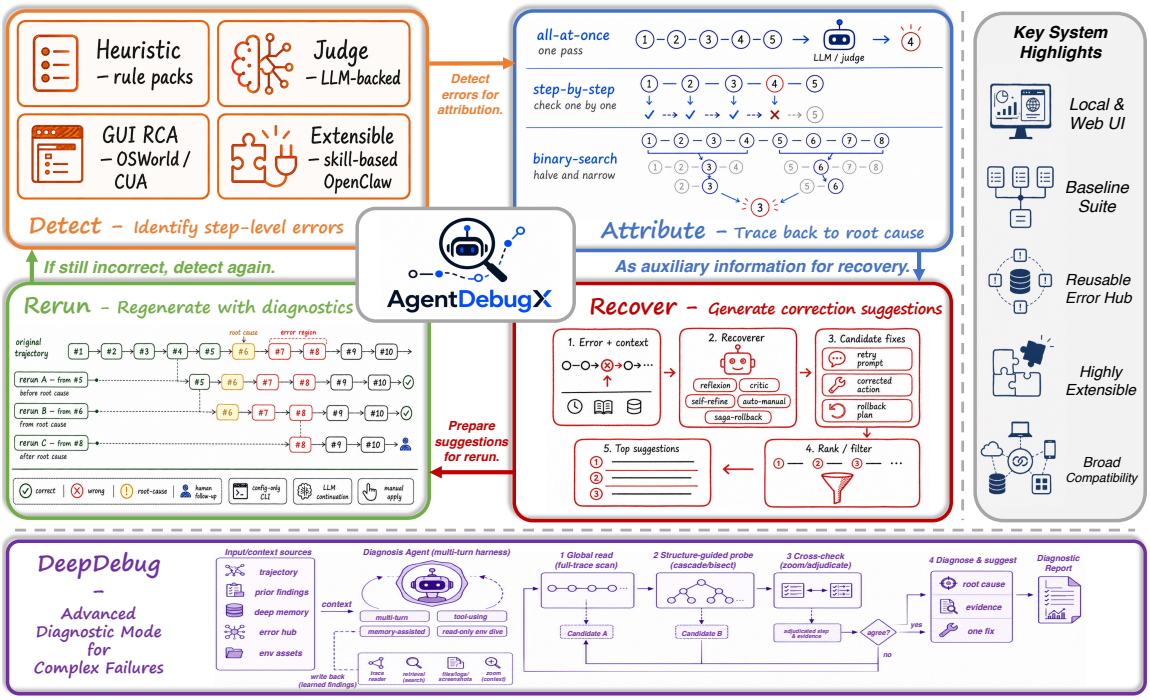

Figure 1: **Overview of AgentDebugX.** AgentDebugX forms a closed debugging loop: **Detect** identifies step-level errors, **Attribute** locates their root causes, **Recover** proposes ranked fixes, and **Rerun** regenerates the trajectory around the failure point. Hard cases are escalated to **DeepDebug**, a multi-round diagnosis agent that produces an auditable root-cause report with evidence and a recommended fix. AgentDebugX supports local and web UIs, reusable error sharing, baseline evaluation, and broad agent-framework compatibility.

2024), AutoManual (Chen et al., 2024) — study how models revise unsuccessful behavior, and are complementary to our setting: models correct errors far more reliably when the error's *location* is supplied (Tyen et al., 2024), which is exactly what AgentDebugX's attribution provides.

3 System Overview

Figure 1 illustrates AgentDebugX, a closed-loop debugging framework consisting of four stages: **Detect**, **Attribute**, **Recover**, and **Rerun**. These stages are coordinated through a portable debugging artifact, while difficult cases are escalated to the DeepDebug agent for diagnosis and recovery.

3.1 Trace Capture and Representation

The input to the loop is an `AgentTrajectory`, a portable record of an agent execution. The runtime layer converts the events an agent framework emits — LLM calls, tool calls and results, memory operations, handoffs, UI actions — into an ordered sequence of `AgentEvents`. Each event records the acting agent, module, step index, parent event, inputs, outputs, metadata, and any error or artifact produced (including screenshots for GUI agents). A trajectory can be captured from a live runtime through adapters (LangGraph, CrewAI, the OpenAI Agents SDK, OpenTelemetry, raw ReAct) or reconstructed from an exported log through offline importers; all mechanisms produce the same representation, so diagnosis is independent of the original framework. Crucially, a *diagnosis is layered on top of this record rather than written back into it*: the same execution can be re-analyzed by any method, compared across versions, and shared as a regression case without altering the evidence.

3.2 Closed-Loop Debugging Pipeline

Given a captured trajectory, AgentDebugX runs the four-stage loop of Figure 1. Each stage consumes structured outputs from the preceding one and produces inspectable artifacts.

Detect. Deterministic rule packs first target mechanically verifiable failures — malformed tool calls, no-progress loops, invalid outputs, premature success — with no model call. When rules are insufficient, an LLM judge reads the goal and a bounded trace window and returns typed findings (affected event, failure mode, evidence, confidence), expressed in a shared taxonomy whose seed contains 19 modes spanning planning, memory, tool use, verification, and coordination (Ap-

pendix A). Detection locates where a failure *manifests*; the detected event is not necessarily the one responsible, so its findings seed attribution rather than settle it.

Attribute. The attribution stage traces a symptom to the step that rendered the task unsuccessful, using a family of strategies that trade cost for resolution: inexpensive heuristics and single-pass whole-trace reading, then binary search, per-step inspection, and budgeted ensembles. Each attributor returns ranked hypotheses with confidence and provenance, not one unqualified blame assignment, so a deployment dials accuracy against latency and token cost. Cases that remain ambiguous escalate to DeepDebug (Section 3.3).

Recover. Once a responsible step is localized, recovery turns the finding into a concrete retry proposal grounded in the root-cause step, failure mode, evidence, and surrounding context. The native path uses DeepDebug's own correction as the retry directive, requiring no extra model call after diagnosis; **Reflexion** (Shinn et al., 2023), **CRITIC** (Gou et al., 2024), and **AutoManual** (Chen et al., 2024) ship as alternative strategies and baselines. All are *suggest-only*: because a repaired action can change the world, application stays behind an explicit human or policy gate.

Rerun. AgentDebugX packages the diagnosis, selected checkpoint, and retry directive into a rerun request; runtime-specific executors consume that request to produce a new trajectory, and the console additionally supports model-generated continuation branches for interactive comparison. The branch is scored against the objective and kept beside the original, preserving both the failure and the attempted repair; a successful branch is saved as a resolved case, a failed one re-enters detection. AgentDebugX does not automatically replay arbitrary external tools from an imported log.

3.3 DeepDebug: Multi-Turn Root-Cause Agent

Single-pass attribution has complementary blind spots: a global read preserves task context but anchors on the loudest downstream symptom, while a narrow step-wise scan loses sight of the objective. DeepDebug (lower panel of Figure 1) resolves this through a multi-turn, read-only process over the captured trace, in four stages.

Stage 1 — global read. The agent reads the whole trajectory, reconstructs the objective and history, and names an initial candidate for the decisive step, keeping the context needed to tell a causal error from a locally unusual but valid action.

Stage 2 — structure-guided investigation. A second pass uses the strategy matching the trace shape: for a multi-agent run it walks the handoff cascade upstream from the visible failure to the earliest step that doomed the run; for a single-agent run it bisects the step range and re-reads the surviving region, yielding an independent second candidate.

Stage 3 — cross-examination. If the two passes agree, the step is accepted; if they disagree, DeepDebug inspects both candidates side by side with their context, inputs, outputs, and downstream effects, and selects the stronger causal explanation — reducing root-cause selection from a search over the whole trace to a focused adjudication between two hypotheses.

Stage 4 — diagnosis and suggestion. With the step fixed, DeepDebug emits a structured report: the responsible agent and step, a plain-language explanation, quoted evidence, and one concrete fix that recovery can use directly or reformulate. Every inspection is recorded, so the verdict ships with its audit trail. The agent inspects the trace but never re-executes the run's tools.

3.4 Extensible Failure Taxonomy

Detection and diagnosis share a structured vocabulary of failure modes. A fixed taxonomy cannot anticipate every long-tail error, so AgentDebugX proposes extensions for human review: when the judge meets a recurring failure outside the seed set it records a novel-mode candidate, and an inducer collects such residuals, clusters them (label, then lexical or embedding similarity, gated by a support threshold), proposes one candidate mode per cluster, and deduplicates against the seed. Proposals *never* overwrite the curated taxonomy: when several runs show agents waiting on one another indefinitely, for example, the inducer proposes a new *multi-agent deadlock* mode, notes its kinship to the existing lost-handoff category, and leaves the decision to a maintainer.

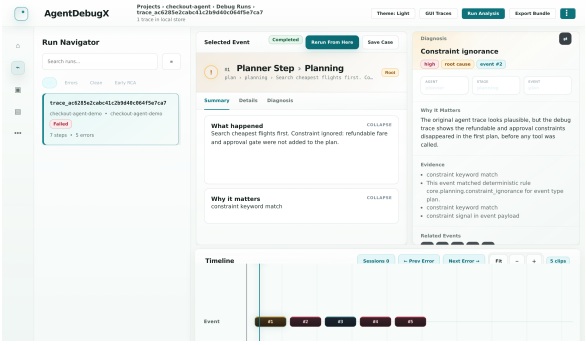

Figure 2: The AgentDebugX console on a failed run, as a four-step workflow: (1) select a stored trace in the run navigator; (2) jump from the visible failure to the attributed responsible event; (3) read the failure mode, evidence, and proposed fix in the diagnosis panel; (4) create a policy-gated rerun branch and compare it with the original timeline. All artifacts stay in the local store unless a bundle is explicitly exported.

3.5 System Surfaces and Integrations

The pipeline is exposed through surfaces that share the same trajectory, finding, report, and recovery types, so a case created in one can be continued in another. The **interactive console** (Figure 2) launches with `agentdebug serve` over the local store the library writes; as a no-build single-page app shipped in the wheel, it takes a developer from `pip install` to a live debugger. A typical session follows the four numbered steps of Figure 2: select a failed run, jump from the visible failure to the attributed responsible event, review evidence and the proposed fix, then fork a *rerun-from-step* branch that is scored against the original. The same views serve computer-use agents: an OSWorld importer normalizes screenshot-and-action steps and a GUI root-cause mode reasons over the visual channel, while a Python-style traceback or JSON report serves CI when no browser is wanted.

The **Error Hub** turns incidents into shared knowledge. A trajectory, its report, and artifacts are packed into a bundle whose scrubber, by default, strips event inputs wholesale — the fields carrying prompts and tool arguments — and applies known-pattern credential and PII redaction to every remaining string; because pattern redaction cannot guarantee removal of arbitrary sensitive content, sharing stays opt-in and bundles should be reviewed before publication. Bundles are written to a local directory, a private Git remote, or a public dataset, doing double duty as a CI fixture and an entry in a cross-team corpus. That corpus is also AgentDebugX's long-term **memory**: DeepDebug can retrieve similar past cases to seed its hypotheses and writes each new case back; through the shared bundle format, accepted entries and reviewed taxonomy additions become available to future diagnosis (this effect is not yet evaluated). Finally, DeepDebug is packaged as an installable **agentic skill** and CLI, so tool-using agents (Claude Code, OpenClaw, and Hermes, each installing the skill) can normalize their own failed run, diagnose it, and feed the fix back into their next attempt — one CLI contract through which supported host agents can diagnose their own runs and each other's.

4 Evaluation

We evaluate the two things a deployed debugger must do, in pipeline order: localize the cause accurately, then turn that diagnosis into a *fix*.

Setup. Unless noted, the debugged policy is `qwen3.5-9b` and all diagnosis (judge and DeepDebug) runs on `gemini-2.5-flash`, at temperature 0 with thinking disabled. Diagnostic memory and the Error Hub stay *empty* throughout. Per-task outputs, configs, and a script regenerating Table 3 ship with the code; full protocol in Appendix C.

Failure attribution. We evaluate on all 184 Who&When traces (Zhang et al., 2025). Each trace is labelled with *who* erred — the *responsible agent*, whose decision doomed the run — and *when* — the exact *mistake step*. Following the benchmark's reference-answer protocol, every method receives the task's reference answer but neither gold label. With `qwen3.5-9b` (Table 2), DeepDebug's two-reading adjudication outperforms the evaluated single-strategy localizers on all five reported metrics — responsible agent (56.0 vs 47.8), exact/near (± 1) step (28.8/44.0 vs 22.3/38.6), and strict agent-and-step (28.8/32.1 vs 21.7/23.9). The joint gain carries to `qwen3.6-27b` (strict 38.0 vs 36.4). The benefit is model-dependent: on the hosted backbones of Appendix C a single global reading is already stronger and adjudication does not improve it, motivating per-model routing rather than invoking the multi-call method unconditionally. Swapping the structure-aware reading for a second global searcher costs 4.8 strict points on `gpt-5.4-mini` (ablation, Appendix C).

Where the multi-turn agent pays off. Figure 3 breaks the same run down by trace length: the margin concentrates on traces longer than 40 events, the regime the structure-guided turn and cross-examination target. Only 26 traces fall in this range,

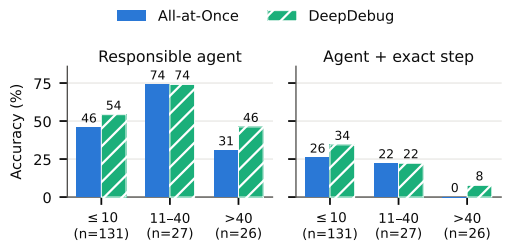

Figure 3: Who&When accuracy by trace length (n=184; `qwen3.5-9b`). Left: responsible-agent accuracy (*who*). Right: strict joint accuracy requiring both the responsible agent and the exact mistake step (*who+when*) to be correct.

Method	Agent	Step (exact)	Step ($\pm$1)	A+S (exact)	A+S ($\pm$1)
Rule heuristic	14.1	1.6	6.5	1.6	2.7
All-at-Once	47.8	22.3	35.3	21.7	23.9
Step-by-Step	41.8	18.5	38.0	17.9	19.6
Binary-Search	41.8	17.4	38.6	17.4	20.1
DeepDebug	**56.0**	**28.8**	**44.0**	**28.8**	**32.1**

Table 2: Failure attribution on the full Who&When benchmark (n=184; `qwen3.5-9b`). "Agent" measures responsible-agent accuracy (*who*); "Step" measures mistake-step localization (*when*) under exact and $\pm$1 criteria; "A+S" requires both agent and step to be correct.

so we treat it as descriptive evidence.

End-to-end recovery on GAIA. Localization is only useful if it *changes an outcome*, so we close the loop. A vanilla `qwen3.5-9b` Open-DeepResearch agent solves 55.8% of GAIA validation and fails 73 tasks; we diagnose each failure and rerun it once. AgentDebugX's **native path** phrases DeepDebug's own localized fix directly into the retry; Reflexion, CRITIC, and AutoManual run as *decoupled* baselines: same failed-task context plus a generic judge-produced failure summary, but not DeepDebug's localization, evidence, or authored fix. Under this shared subset, DeepDebug's recovery produces more than double the one-rerun repairs of the decoupled baselines (Table 3), with the largest gain on Level-2 multi-hop tasks (48.8 $\rightarrow$ 61.6) — evidence its localized, evidence-backed fix is useful, though the experiment evaluates the full recipe rather than isolating attribution alone.

Cost-aware selection. Every backend declares its inference cost (free rules; one call for All-at-Once and the judge; $O(\log N)$/$O(N)$ searches; $\sim$5 calls for DeepDebug), letting deployments trade model calls for localization accuracy. The escalation is cheaper than the call count suggests: on a 25-trace stratified Who&When sample, one whole-trace pass averages 8.1K tokens against DeepDebug's 12.8K (1.6$\times$; its later turns read focused windows), so escalating only ambiguous cases keeps expected cost near a single pass; every attributor returns ranked hypotheses with provenance rather than one authoritative blame claim.

Method	Rec.	L1	L2	L3	All
Vanilla `qwen3.5-9b`	—	77.4	48.8	34.6	55.8
+ CRITIC	4/73	**81.1**	51.2	34.6	58.2
+ AutoManual	5/73	79.2	53.5	34.6	58.8
+ Reflexion	6/73	77.4	55.8	34.6	59.4
+ DeepDebug (ours)	**13/73**	**81.1**	**61.6**	34.6	**63.6**

Table 3: End-to-end recovery on GAIA validation (165 tasks; official scorer). "Rec." denotes the number of repaired cases among the 73 vanilla-failed trajectories. Accuracy (%) is reported by difficulty level and overall after a single rerun.

Takeaway. Across both experiments, DeepDebug's multi-turn diagnosis delivers state-of-the-art attribution on Who&When (28.8% strict vs 21.7% for the best single strategy) and turns those diagnoses into 2–3$\times$ more repaired GAIA failures than decoupled self-correction (+7.9 points), at only 1.6$\times$ the tokens of a single pass.

5 Use Cases and Applications

AgentDebugX supports a range of debugging workflows. A typical session begins by capturing or importing a failed execution. DeepDebug then localizes the responsible step, explains the root cause, and proposes a repair, which can be reviewed before a policy-controlled rerun. The resulting trajectory is retained alongside the original and can optionally be stored in the Error Hub as reusable debugging memory. This workflow supports interactive debugging, CI regression testing, incident response, and self-debugging agents.

6 Conclusion

In this work, we presented AgentDebugX, a closed-loop debugging framework that connects failure detection, root-cause attribution, recovery, and rerun for LLM agents. Its core method, DeepDebug, demonstrates that improving fault attribution can translate into measurable gains in downstream recovery. We hope AgentDebugX provides a practical foundation for debugging increasingly capable LLM agents and facilitates future research on reliable agent development.

Ethical Considerations

AgentDebugX collects potentially sensitive agent traces, including prompts, tool arguments, user data, files, screenshots, and outputs. The default deployment is local-first, and shared corpus upload is opt-in. Any production deployment should configure redaction, retention, access control, and audit logging before collecting user data. Diagnostic labels and recovery suggestions can be wrong, so the UI and API should present confidence and evidence rather than framing attribution as ground truth. For high-impact domains, deployments should require human or policy approval rather than automatic recovery.

Broader Impact

AgentDebugX can help make agent reliability an inspectable and measurable engineering practice. Today, failures are often patched privately and then forgotten; systematic attribution and recovery evidence can instead inform audit trails, deployment gates, regression tests, and decisions about when an agent should not be trusted. This matters as agents enter software development, science, education, accessibility, and public-facing services, where silent or repeated errors can propagate beyond a single run. An open-source toolkit and common trace representation also lower the cost for researchers and smaller organizations to study robustness and compare systems on shared evidence.

The **Error Hub** extends this impact from individual deployments to a community: one team's failure can become another team's regression case and, at scale, contribute to realistic benchmarks, evolving taxonomies, and better diagnostic or recovery methods. Together, AgentDebugX and the Hub create a path from isolated incidents to organizational learning and eventually shared robustness standards. Realizing this vision requires consent, provenance, effective redaction, moderation, takedown processes, and audits of whose systems and failures remain underrepresented. With these safeguards, AgentDebugX could make agent robustness more cumulative, collaborative, and broadly accessible rather than a proprietary advantage.

References

Mohamed Aghzal, Erion Plaku, Gregory J Stein, and Ziyu Yao. 2025. A survey on large language models for automated planning. *arXiv preprint arXiv:2502.12435*.

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

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. 2026. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. 2024. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. *Preprint*, arXiv:2405.16247.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. Trail: Trace reasoning and agentic issue localization. *Preprint*, arXiv:2505.08638.

Liming Dong, Qinghua Lu, and Liming Zhu. 2024. Agentops: Enabling observability of llm agents. *Preprint*, arXiv:2411.05285.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*.

Langfuse. 2023. Langfuse: Open-source LLM engineering platform. `https://github.com/langfuse/langfuse`. Open-source tracing and observability for LLM applications.

Bingxuan Li, Yiwei Wang, Jiuxiang Gu, Kai-Wei Chang, and Nanyun Peng. 2025. METAL: A multi-agent framework for chart generation with test-time scaling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30054–30069, Vienna, Austria. Association for Computational Linguistics.

Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang, Chenchen Zhang, Ge Zhang, Jiebin Zhang, and 1 others. 2025. A comprehensive survey on long context language modeling. *arXiv preprint arXiv:2503.17407*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*.

OpenTelemetry. 2026. Opentelemetry semantic conventions for generative AI. https://github.com/open-telemetry/semantic-conventions-genai. Accessed 2026-07-09.

Tianyue Ou, Wanyao Guo, Apurva Gandhi, Graham Neubig, and Xiang Yue. 2025. AgentDiagnose: An open toolkit for diagnosing LLM agent trajectories. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 207–215, Suzhou, China. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Gladys Tyen, Hassan Mansoor, Victor Carbune, Peter Chen, and Tony Mak. 2024. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13894–13908, Bangkok, Thailand. Association for Computational Linguistics.

Jian Yang, Xianglong Liu, Weifeng Lv, Ken Deng, Shawn Guo, Lin Jing, Yizhi Li, Shark Liu, Xianzhen Luo, Yuyu Luo, and 1 others. 2025. From code foundation models to agents and applications: A comprehensive survey and practical guide to code intelligence. *arXiv preprint arXiv:2511.18538*.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng YAN. 2026. Agentracer: Who is inducing failure in the LLM agentic systems? In *The Fourteenth International Conference on Learning Representations*.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Forty-second International Conference on Machine Learning*.

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

A System and Prompt Details

Trace schema. Each run is an `AgentTrajectory` of `AgentEvents` (type, agent, module, step index, parent, timestamp, inputs/outputs, error, duration, metadata, artifacts). Artifacts can hold text, images, audio, UI state, files, or environment snapshots, and the same events project to OpenTelemetry GenAI spans (`invoke_agent`, `chat`, `execute_tool`, `handoff`). The diagnosis layers are driven by compact, JSON-constrained prompts; we reproduce the most load-bearing ones below (the full prompt library ships in the repository), followed by a complete diagnostic report produced during the GAIA experiment.

LLM Judge — System Prompt (abridged)

```
You are AgentDebugX-Judge. Given the goal, the
ALLOWED failure-mode codes, and one run's events,
label each failed step with ONE allowed code. Be
conservative; return an empty list if no failure.
Output ONLY JSON, no prose/fences; cap findings at
{max_findings}; evidence < 120 chars; never emit
newlines inside strings; close the JSON completely.
Schema: {"findings":[{"event_id","step_index",
"agent_name","failure_mode_id","confidence",
"evidence":[...]}], "summary":"..."}
```

Whole-Trace Localizer — System Prompt (abridged)

```
You are an AI assistant tasked with analyzing
agent conversation history when solving a real
world problem.
Respond ONLY with a JSON object matching this
schema (no prose, no markdown):
{ "span_id": "<event_id or null>",
  "step_index": <int or null>,
  "agent_name": "<agent_name or null>",
  "confidence": <float 0..1>,
  "rationale": "<one or two sentences>",
  "evidence": ["<short quoted evidence>", ...] }
If the trajectory does not appear to have failed,
return all fields as null.
```

Cross-Examination (disagreement arbitration)

```
System: A run FAILED. Two candidate error steps
are shown with their context. Which ONE is the
decisive critical error step (the one whose
correction would most likely have averted the
failure)?
Respond ONLY with JSON: {"step": <int, one of
the two>}

User: Goal: <task goal>
Correct answer: <reference, when available>
CANDIDATE A = step <i>: <+-1-step context>
CANDIDATE B = step <j>: <+-1-step context>
Which is the critical error step, <i> or <j>?
```

DeepDebug Refine — System Prompt

```
You are AgentDebugX-DeepDebug writing the final,
human-readable diagnosis. The decisive ROOT-CAUSE
step has ALREADY been localized for you -- do NOT
second-guess or move it. Using the step and its
surrounding context, write: a one-or-two sentence
diagnosis of WHY that step caused the failure;
the concrete evidence (short quoted snippets from
the shown steps); and ONE concrete, actionable
fix.
Respond ONLY with JSON: {"summary": "<1-2
sentences>", "evidence": ["<short quoted
snippet>", ...], "suggestion": "<one fix>"}
```

Example diagnostic report. The abridged report below is DeepDebug's actual output on a GAIA validation task the vanilla agent failed (a statistics question over 1,002 papers with average p-value 0.04); the agent had assumed p-values were uniformly distributed to estimate the number of incorrect papers. Fed verbatim as the retry directive, this diagnosis led the rerun to the correct answer (41), one of the 13/73 recoveries in Table 3.

DeepDebug report (GAIA task 04a04a9b, abridged)

```
{ "trace_id": "04a04a9b-...",
  "root_cause_agent": "print",
  "root_cause_step_index": 9,
  "findings": [{
    "failure_mode": "attribution.root_cause",
    "step_index": 9, "confidence": 0.6,
    "evidence": [
      "# Assuming uniform distribution
         [0, 2*average_p_value]",
      "proportion_incorrect = (2*average_p_value
         - threshold) / (2*average_p_value)"],
    "suggestion": "Calculate the number of
      incorrect papers using ... the direct
      definition of the p-value as the Type I
      error rate under the null hypothesis:
      1002 * 0.04 = 40.08, rounding up to 41."
  }],
  "summary": "The model incorrectly assumed a
    uniform distribution of p-values over
    [0, 0.08] to calculate the proportion of
    incorrect papers -- a mathematically
    arbitrary assumption not supported by the
    problem statement.",
  "metadata": { "analyzer": "DeepDebugAnalyzer",
    "backend": "aao_moe", "memory_used": false }}
```

B Deployment Requirements

The design follows five requirements from production agent operations: **low-friction** capture (a context manager, callbacks, or importers); a **portable** representation with an OpenTelemetry GenAI export path (OpenTelemetry, 2026); **typed** diagnoses (root cause, evidence, confidence, fix); **local-first** storage with explicit scrubbing before sharing; and **cost-aware** analysis, where deterministic triage is free and LLM depth is opt-in.

C Evaluation Protocol

Benchmarks. *Who&When* (Zhang et al., 2025) is used in full (n=184: 126 algorithm-generated, 58 hand-crafted), each trace annotated with the gold

responsible agent and mistake step. *GAIA* (Mialon et al., 2024) validation (165 tasks across three difficulty levels) is used for end-to-end recovery, scored by the official question scorer.

Attribution protocol. Following the Who&When "with ground truth" setting, every method receives the task's reference answer as context; none sees the gold agent or step. Decoding is at temperature 0 with model-side thinking disabled; completions are capped at 4,096 tokens (512 for arbitration). Agent names are normalized before comparison, and step match is exact index equality. We report responsible-agent accuracy, exact and ± 1 step-localization accuracy, and the joint agent-and-step (A+S) metrics. Backbones span open-weight (`qwen3.5-9b`, `qwen3.6-27b`) and hosted (`gpt-5.4-mini`, `gemini-3.5-flash`) models, all served through an OpenAI-compatible endpoint.

GAIA recovery protocol. A vanilla Open-Deep-Research agent (`qwen3.5-9b` policy, Gemini-2.5-flash search) is run once over all 165 validation tasks, yielding a 73-task failure subset. Each failed trajectory is converted to an `AgentTrajectory`, diagnosed by the LLM judge and, for the native path, DeepDebug. Recovery then reruns *only* the 73 failed tasks once, under one of four strategies: AgentDebugX's native DeepDebug directive, or the decoupled Reflexion / CRITIC / AutoManual baselines (which see a generic judge summary, not DeepDebug's localized diagnosis). To keep the comparison clean, the diagnostic memory and Error Hub are empty during evaluation; both are opt-in for real deployments. Reruns use temperature 0, a fixed step budget, and the same search and tool stack as the baseline. "Rec." counts repaired tasks over the 73; "All" projects corrected tasks over the full 165.

Localizer design ablation. DeepDebug's turn design was chosen by measurement on a stratified 42-trace Who&When sample: replacing the structure-guided turn with a second global search drops strict accuracy from 0.310 to 0.262 on `gpt-5.4-mini`, while the shipped pairing lifts agent accuracy to 0.524 vs 0.429 for one reading; on `gemini-3.5-flash` a single reading is already strongest — adjudication pays on the evaluated open-weight backbones but not on the hosted ones — a model-dependent effect.

D Implementation

AgentDebugX is a dependency-light, MIT-licensed Python package: `pip install agentdebugx`, imported as `agentdebug`. The public API centers on `AgentDebug`, `TraceSession`, `AgentTrajectory`, `FailureFinding`, and `DiagnosticReport`; adoption is one context manager:

```
from agentdebug import AgentDebug, EventType
dbg = AgentDebug()
with dbg.trace(goal="Book flight",
               framework="my-agent") as t:
    t.record(EventType.PLAN, agent_name=
      "planner", output="Search fares")
    t.record(EventType.TOOL_RESULT,
      agent_name="browser", step_index=3,
      error="Checkout timeout")
    report = t.analyze()
```

Traces persist to append-only JSONL or SQLite. Three capture surfaces emit the same schema: runtime adapters (raw ReAct, LangChain/LangGraph, CrewAI, OpenAI Agents SDK, OpenTelemetry GenAI), offline importers (message lists, conversations, event lists, WebShop pages, OpenAI Agents spans, CrewAI events, OpenClaw sessions), and host integrations (a Claude Code skill and a CLI skill contract for tool-using agents), so detection, attribution, and recovery are source-independent. The CLI exposes the workflow as `ingest / diagnose / inspect / act`, with `serve` for the console.

E Limitations

Our evaluation covers automatic attribution and recovery, not developer debugging time or console usability. Who&When uses the benchmark's reference-answer protocol, and DeepDebug's gains vary by model, so its extra calls are not uniformly beneficial. The GAIA experiment evaluates one policy model on a fixed failed subset and compares complete retry recipes; it neither isolates attribution's effect alone nor is a blind single-shot score. Error Hub retrieval and taxonomy induction are implemented but not yet evaluated, and induction requires human acceptance. The scrubber strips event inputs and redacts known credential/PII patterns but not arbitrary content; recovery execution stays gated on human approval.