2026 IEEE International Conference on Software Testing, Verification and Validation (ICST) | 979-8-3195-3308-1/26/$31.00 ©2026 IEEE | DOI: 10.1109/ICST69053.2026.00037

Observability and Fault Injection for LLM-Based Multi-Agent Systems in Software Engineering

Zahra Seyedghorban*, Egor Klimov†, Arie van Deursen*, Annibale Panichella*, Burcu Kulahcioglu Ozkan*
*Delft University of Technology, The Netherlands
{z.seyedghorban, arie.vandeursen, a.panichella, b.ozkan}@tudelft.nl
†JetBrains Research, The Netherlands
egor.klimov@jetbrains.com

***Abstract*—Large Language Model-based multi-agent systems are increasingly explored for software engineering tasks, but they remain difficult to inspect, debug, and evaluate under controlled failures. We present `llmmas-otel`, a lightweight and framework-agnostic tool that combines OpenTelemetry-based distributed tracing with fault injection for LLM-based multi-agent systems in software engineering workflows. The tool instruments agent executions with trace-aligned telemetry across workflow phases, agent steps, inter-agent communication, tool calls, and LLM invocations, and supports targeted fault injection at selected interaction points. This makes it possible to compare baseline and faulty executions in a reproducible way and inspect the effects through aligned traces and run artifacts. We describe the motivation, architecture, implementation, current capabilities, and initial validation of the tool on a minimal demo workflow and a real LLM-based multi-agent system for software development. A screencast is available at: ⟨https://youtu.be/tyQh2xO-flo⟩**

***Index Terms*—LLM-based multi-agent systems, software engineering, observability, fault injection, tracing, debugging, OpenTelemetry**

I. Introduction

Large Language Model-based multi-agent systems (LLM-MAS) are increasingly explored for software engineering (SE) tasks such as planning, coding, reviewing, and testing. Instead of relying on a single model call, the work is distributed across multiple specialized agents that communicate, coordinate, and use external tools to complete a task [1], [2]. This makes LLM-based multi-agent systems attractive for complex software engineering workflows, but in practice they still fail frequently and are hard to debug in a principled way [3], [4]. Recent empirical studies report high failure rates across popular MAS frameworks (including SE-oriented ones) [3].

When these systems fail, the cause is not always obvious. A final incorrect output may come from a much earlier problem, such as a missed constraint, a weak handoff between agents, a tool error, or a model response that quietly pushes the workflow off track. Recent work has shown that failures in multi-agent LLM systems are frequent and varied, including inter-agent misalignment and missing or incomplete verification [5], [6]. The difficulty is not only that the workflows are multi-step and distributed across agents, but also that they are stochastic: the same task can unfold differently across runs. As a result, simple logs or final success rates are often not enough to understand what happened or to compare normal and faulty executions in a principled way.

Multi-agent system behavior can be systematically tracked using distributed tracing [7]. A distributed trace captures an entire execution workflow as a single continuous chain, relying on context propagation to link operations together. A trace typically starts from the initial system invocation, such as a user prompt to a planning agent. As the execution context propagates through the system, every distinct operation is recorded as a span. Each span represents a specific part of the execution and contains attributes that provide semantic details about what occurred. OpenTelemetry [8] is a vendor-agnostic open standard for observability that defines the conventions for generating and collecting telemetry data, while also providing the SDKs to implement them. To ensure these semantic details are consistent across different instrumentation libraries, OpenTelemetry relies on standard semantic conventions.

Existing work helps us observe that LLM-based multi-agent systems fail, but practitioners still lack a reusable engineering layer for two things: (1) capturing executions in a structured, comparable form across workflow phases, agents, tool calls, and LLM calls, and (2) injecting controlled faults at those same interaction boundaries to study how local perturbations propagate. This paper presents `llmmas-otel`, a lightweight and framework-agnostic tool that addresses this gap through two tightly connected capabilities: observability and fault injection.

A key design choice in `llmmas-otel` is to work *around* an existing LLM-based multi-agent system rather than replacing them. Instead of proposing yet another agent framework, `llmmas-otel` provides instrumentation and injection points that can be integrated into an existing workflow with limited code changes. We report initial validation on both a demo workflow and an existing LLM-MAS. The intended users of `llmmas-otel` are researchers and developers who already have an LLM-based multi-agent workflow and want to inspect its execution or stress-test it under controlled perturbations. In summary, this paper makes the following contributions:

- We present `llmmas-otel`, a reusable tool for observability and fault injection in LLM-based multi-agent systems for software engineering.
- We describe a trace model that captures runs across workflow segments, agent steps, communication events,

979-8-3195-3308-1/26/$31.00 ©2026 IEEE
DOI 10.1109/ICST69053.2026.00037

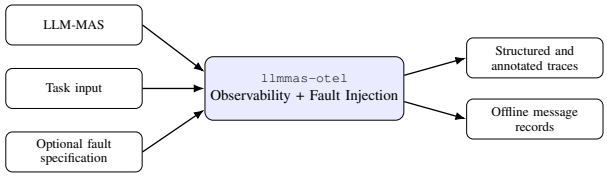

Fig. 1. High-level input/output view of `llmmas-otel`. The framework wraps an existing LLM-based multi-agent system, optionally applies configured faults, and produces aligned artifacts for analysis.

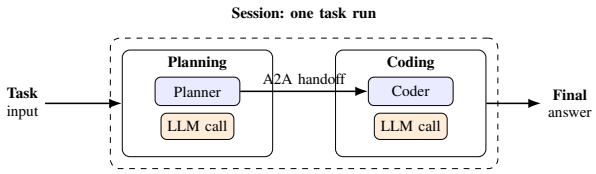

Fig. 2. Running example used throughout the paper: a minimal Planner→Coder LLM-based multi-agent system workflow for a software engineering task.

tool invocations, and LLM calls in a consistent and framework-agnostic way.
- We provide an initial validation of the tool on a minimal demo workflow and a real LLM-based multi-agent system setting.

The rest of the paper is organized as follows. Section II presents `llmmas-otel`, including its overall design and its observability and fault injection layers. Section III reports the current validation results. Section IV discusses related work and the novelty of the tool. Section V outlines current limitations and directions for future work. Finally, Sections VI and VII discuss tool availability and conclude the paper.

II. Framework

A. Overview

`llmmas-otel` is a lightweight and framework-agnostic tool for observing and perturbing executions of LLM-based multi-agent systems in software engineering workflows. Rather than proposing yet another agent framework, it wraps an existing workflow at a small number of meaningful boundaries and produces aligned execution artifacts. At a high level, the tool takes as input an existing LLM-based multi-agent system, a task to execute, and optionally a fault specification. It then produces structured traces together with optional offline message records and trace annotations that make both baseline and faulty runs inspectable in a consistent way.

Figure 1 shows this high-level view. The underlying LLM-based multi-agent system remains responsible for performing the task itself. `llmmas-otel` is responsible for instrumentation, optional perturbation, and export of aligned artifacts for later analysis.

From a user perspective, adopting `llmmas-otel` consists of three practical steps. First, the user marks a small number of existing workflow boundaries, such as task/session start, workflow phases, agent steps, and selected A2A, tool, or LLM operations. Second, the user runs the target workflow once without perturbation to obtain a baseline trace and optional offline message records. Third, the user optionally enables one or more configured fault rules and re-executes the same task to obtain a structurally aligned faulty run.

Throughout the rest of this section, we use the minimal demo workflow in Figure 2 as a running example. The demo contains the core ingredients of an LLM-based multi-agent system execution. A task is first handled by a *Planner* agent in a *planning* phase. The Planner performs one LLM call to generate a short implementation plan and then sends that plan as an agent-to-agent message to a *Coder*. The workflow then enters a *coding* phase, where the Coder receives the message and performs a second LLM call to produce the final answer. This gives us a compact pipeline with two phases, two agent steps, one inter-agent handoff, and two LLM calls.

B. Observability Layer

The usefulness of this trace model comes not only from its hierarchy, but also from the semantic information associated with each span. This information is captured through a consistent vocabulary of span attributes, where each attribute has a stable name and meaning across executions. Table I summarizes the vocabulary used by the current implementation. This consistency allows traces to be compared and queried systematically across different runs and workflows.

The running example in Figure 2 makes these attributes concrete. At the highest level, the full execution is represented by a *session span*. Inside it, the planning phase becomes a *segment span* with its own name and ordering information, and the Planner's work becomes an *agent_step span* that records the acting agent and step index. The Planner's model invocation is then captured as an `llm_call` span, while the handoff from Planner to Coder is captured through `a2a_send` and `a2a_receive` spans carrying the communication metadata listed in Table I. This means that the trace not only shows that an LLM call occurred or that a message was sent, but also which phase it occurred in, which agent was responsible, and how later steps relate to it.

From the user side, the instrumentation is small and lightweight in the sense that users annotate only a small number of existing workflow boundaries and reuse the host MAS implementation. Users mark a few existing workflow boundaries, while the lower layers handle span creation, context propagation, and export. In the current implementation, thin decorators and context managers form the public API, while the span factory is responsible for creating spans, attaching the attributes in Table I, and linking related events across communication boundaries. This keeps the tool close to the host workflow instead of forcing users into a new orchestration model.

A further benefit of this design is that communication edges remain explicit. On send, the tool can propagate tracing context in the message carrier; on receive, it can restore that context and record the relation between the two events. This is

TABLE I
MAIN SPAN ATTRIBUTES USED BY THE OBSERVABILITY LAYER.

Span	Attributes	Meaning
Session	`session.id`	Identifies one full task run and groups all child spans under the same execution.

Segment	`segment.name`
	`segment.order`
	`segment.origin (opt.)`
		Records the workflow phase, its position in the run, and optionally where the segment originated from.

Agent-step	`agent.id` `step.index`	Identifies which agent performed the step and the turn index of that agent within the run.

A2A send / receive	`source_agent.id`
	`target_agent.id`
	`edge.id`
	`message.id`
	`channel`
	`message.preview`
	`message.sha256`
		Describes the communication edge, the message identity, the channel, and lightweight message-content hints for inspection or matching.

Tool call	`operation.name`
	`tool.name`
	`tool.type (opt.)`
	`tool.call.id`
	`tool.args.preview (opt.)`
	`tool.args.sha256 (opt.)`
		Records which tool was invoked, the call identity, and optional lightweight summaries of tool arguments.

LLM call	`operation.name`
	`provider.name`
	`request.model`
	`request.id`
	`llm.input.preview (opt.)`
	`llm.input.sha256 (opt.)`
		Records the provider, model name, and request identity, plus optional lightweight summaries of prompt input.

Injected span	`fault.injected`
	`fault.type`
	`fault.spec_id`
	`fault.decision`
		Marks that a configured fault was applied on an A2A, tool, or LLM boundary and records the decision taken.

particularly useful in agent-based workflows, where the final outcome often depends on the quality and timing of inter-agent handoffs rather than on isolated LLM calls alone.

C. Fault Injection Layer

On top of the observability layer, `llmmas-otel` provides a configurable fault injection layer. The guiding idea is simple: the same boundaries that are important for tracing are also the most natural points for controlled perturbation. Instead of modifying the core workflow logic, the tool evaluates configured rules at selected hook points such as LLM calls, tool invocations, A2A send, and A2A receive. If a rule matches the current execution context, the tool applies the requested action at that boundary.

This design gives the tool flexibility along two dimensions. First, faults can be targeted precisely. A specification can select a fault based on boundary type and contextual information such as phase name, acting agent, source and target agents, message channel, or similar execution metadata. Second, the same instrumentation points can support different fault behaviors, depending on the boundary. Table II summarizes the currently supported faults in the implementation. The injection layer is also extensible: new hook types and new actions can be added without changing the host workflow code. Because the mechanism is configuration-driven, the same workflow can be re-executed under different fault scenarios without re-instrumentation.

TABLE II
CURRENTLY SUPPORTED FAULTS BY INJECTION BOUNDARY.

Boundary	Fault	Meaning

A2A send	`delay`
	`drop`
	`truncate`
		Pause before sending the message.
		Suppress the outgoing message.
		Shorten the message payload before send.

A2A receive	`delay` `drop`	Pause before processing the received message. Discard the message at the receive side.

Tool call	`delay`
	`not_installed`
	`timeout`
	`malformed_response`
		Pause before tool completion.
		Raise a tool-not-found error.
		Raise a timeout error.
		Return an intentionally malformed tool output.

LLM call	`delay`
	`rate_limit`
	`timeout`
	`network_error`
	`malformed_response`
		Pause before the model response.
		Raise an injected rate-limit error.
		Raise an injected timeout error.
		Raise an injected network error.
		Return an intentionally malformed model output.

The fault injection layer is closely tied to the trace model. When a fault injection is applied, the affected execution is still represented by the same operational span type. In other words, an A2A delay is recorded on an A2A communication span, and an LLM delay is recorded on an LLM-call span. The difference is that the span now carries fault-specific attributes and events indicating that a perturbation was injected, what type it was, which rule triggered it, and what decision was taken (Table I). This keeps baseline and faulty runs structurally comparable, which is essential for side-by-side analysis.

Suppose we configure the injection of a delay on the Planner→Coder handoff in Figure 2. The workflow itself does not change. What changes is that the communication boundary now carries fault metadata, and the configured delay is applied before the message continues through the workflow. In such a run, the corresponding `a2a_send` span can still appear in exactly the same structural position as in the baseline trace, but now with attributes such as `llmmas.fault.injected=true`, `llmmas.fault.type=a2a.delay`, and `llmmas.fault.spec_id=A2_A2A_DELAY`, together with scenario-specific data such as the applied delay. Similarly, if the injected fault targets the Planner's LLM call, the planning-phase `llm_call` span remains in place, but it is

annotated as faulted and delayed. This is an important design decision of `llmmas-otel`: fault injection is not treated as a separate logging mode or a disconnected experiment pipeline, but as an overlay on the same execution structure used for normal runs.

Taken together, the observability and fault injection layers provide a single experimental setup for debugging and evaluation. Observability gives a structured account of what happened during a run. Fault injection introduces controlled perturbations at explicit boundaries. It helps users answer concrete engineering questions about their workflows: which boundaries are most sensitive to perturbation, whether a local issue remains localized or cascades across later phases, and whether the workflow degrades through delay, divergence, retries, or outright failure. Combined, they support a more systematic analysis of LLM-based multi-agent systems behavior than final outputs or flat logs alone.

III. Validation

In this section, we demonstrate how `llmmas-otel` can be used to run controlled stress test scenarios and quantify their runtime effect. We use the same setup across two target systems: the minimal Planner→Coder workflow used throughout this paper, and an existing LLM-MAS. In both cases, we use the 30-task ProgramDev benchmark [5] and execute each task five times per condition. We consider three scenarios: a fault-free baseline, a run with a single `llm.delay = 1000 ms` injected at the first LLM call in the planning phase, and a run with a single `a2a.delay = 1000 ms` injected at the Planner→Coder send in the planning phase. To summarize the runtime effect of an injected fault, we use *amplification*, defined as the extra end-to-end runtime caused by the fault divided by the injected delay. An amplification close to 1 means that a 1000 ms injected delay produces roughly a 1000 ms slowdown overall, while values above 1 indicate that the injected delay propagates and causes a larger slowdown than the perturbation itself.

We first applied this setup to the demo workflow. Table III shows that both injected scenarios lead to measurable end-to-end slowdown. In the demo, injecting the planning-phase LLM delay yields a mean amplification of 1.053 and a median of 1.036, which is close to linear overhead. In contrast, delaying the Planner→Coder handoff yields a higher mean amplification of 1.295 and a median of 1.390, suggesting that perturbing the inter-agent communication boundary causes stronger downstream slowdown than delaying the single LLM call alone.

For the second setting, we use ChatDev [1], a chat-powered software development framework in which multiple specialized software agents collaborate through language-based communication across three main software lifecycle phases: design, coding, and testing. It is therefore a useful second validation target because it represents a richer and more realistic system than the minimal demo, while still remaining firmly in the software engineering domain. Table III shows that delay faults in ChatDev lead to substantially higher

TABLE III
Amplification results for the validation scenarios.

System	Scenario	Mean
Demo	Baseline	–
Demo	LLM delay	1.053
Demo	A2A delay	1.295
ChatDev	Baseline	–
ChatDev	LLM delay	48.1
ChatDev	A2A delay	59.2

amplification factors, reflecting how a localized perturbation can cascade across many downstream interactions and LLM calls. In particular, injecting a single LLM delay yields a mean amplification of 48.1 and a median of 13.9, while delaying an inter-agent message boundary yields an even higher mean amplification of 59.2 (median 6.6). Overall, these results highlight that in a multi-phase, message-heavy workflow, the runtime impact of small injected delays can be magnified dramatically, and the proposed tracing+injection setup makes this amplification visible and quantifiable in a trace-aligned manner.

IV. Related Work

A line of work focuses on *understanding* and *classifying* failures in multi-agent systems. MAST provides the first empirically grounded taxonomy of multi-agent LLM system failures, organizing failure modes around specification issues, inter-agent misalignment, and task verification problems [5]. AgentFail extends this direction to platform-orchestrated agentic systems by proposing a root-cause taxonomy across agent-, workflow-, and platform-level failures, and by releasing a benchmark for diagnosing these causes [6]. Related work, such as TRAIL studies, structured agent traces for issue localization and debugging, while Who&When focuses on identifying the decisive agent and step responsible for a failed outcome [**?**], [4]. These papers are highly relevant to our motivation: they show that failures in LLM-based multi-agent systems are diverse, often distributed across long traces, and difficult to localize by manual inspection alone. However, their main contribution is post-hoc analysis, attribution, or taxonomy building. They do not provide a general-purpose tool layer for injecting controlled perturbations into live executions and comparing baseline and faulty runs through a shared trace structure.

Another line of work is more directly related to fault injection. AEGIS [9] generates faulty multi-agent trajectories by taking successful executions and applying controlled, context-aware error injections, producing labeled data for attribution and analysis. AgenTracer [10] also uses synthetic failure generation, combining counterfactual replay with programmatic perturbation of selected steps so that the decisive faulty step is known by construction. These are the closest works to `llmmas-otel`, because they treat controlled perturbation as a way to study failure. Still, their primary

goal is dataset construction and failure attribution rather than providing a reusable runtime tool for engineering evaluation. In AEGIS and AgenTracer, injection is mainly a means to generate faulty trajectories for learning or labeling. In contrast, `llmmas-otel` packages fault injection as a configurable execution-time capability around an existing workflow. The goal is not only to create faulty variants, but also to support repeatable stress-testing, baseline-versus-faulty comparison, and trace-aligned diagnosis at concrete boundaries such as A2A communication, tool calls, and LLM calls.

A third line of work addresses observability and operations for agentic systems. LumiMAS [11] argues for real-time monitoring of multi-agent systems and highlights security- and reliability-relevant phenomena such as prompt injection, memory poisoning, and cascading hallucinations. AgentOps [12] similarly frames observability as a core requirement for autonomous and nondeterministic agents, and discusses the artifacts and telemetry that should be captured across the agent lifecycle. More broadly, recent surveys of LLM-based multi-agent systems emphasize persistent challenges around robustness, memory, coordination, tool use, and evaluation [13]. These works are important for our observability layer, as they motivate why traces must capture more than final outputs. At the same time, they stop short of coupling observability with controlled perturbation. In our setting, observability is not an independent monitoring dashboard; it is the mechanism that makes injected runs interpretable and comparable.

Overall, existing work provides three important ingredients: taxonomies of what can go wrong, attribution methods for identifying where it went wrong, and observability proposals for capturing what happened. What is still missing is a lightweight tool that brings these ideas together for controlled experimentation on existing LLM-based multi-agent systems, especially in software engineering workflows. `llmmas-otel` is designed to fill that gap. It provides the practical infrastructure to run repeatable baseline and fault-injected executions, capture them in a shared trace model, and inspect how faults propagate across phases, agents, and interaction boundaries.

V. Limitations and Future Work

`llmmas-otel` provides a modular and extensible foundation for stress testing LLM-based multi-agent systems. The current implementation supports fault injection at three key runtime boundaries (agent-to-agent communication, LLM calls, and tool calls), which already enables controlled perturbation of important interaction points while preserving aligned traces for baseline and injected runs. Ongoing work extends this capability toward additional fault types and broader injection surfaces, including higher-level workflow, memory-related, and coordination faults.

The current version of `llmmas-otel` focuses on instrumentation, trace-aligned artifact generation, and configurable runtime perturbation. It does not yet provide higher-level analysis features such as paired-run differencing, trace summarization, root-cause ranking, or automated debugging reports; at present, users inspect the produced traces and run artifacts directly. A next step is therefore to build analysis support on top of the current observability layer.

VI. Tool Availability

`llmmas-otel` is publicly available, along with a sample demo, ChatDev instrumentation, and detailed instructions, at GitHub.

VII. Conclusion

We presented `llmmas-otel`, a lightweight and framework-agnostic tool that combines structured observability with fault injection for LLM-based multi-agent systems in software engineering. By keeping baseline and injected executions aligned under a shared trace model, it enables controlled stress testing and systematic inspection of how faults affect multi-agent workflows. The tool can serve as a practical foundation for future work on the reliability and evaluation of LLM-based multi-agent systems.

Acknowledgment

This work was conducted as part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology. The authors gratefully acknowledge the financial support provided by JetBrains, which made this research possible.

References

[1] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong *et al.*, "Chatdev: Communicative agents for software development," in *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, 2024, pp. 15 174–15 186.

[2] H. N. Phan, T. N. Nguyen, P. X. Nguyen, and N. D. Bui, "Hyperagent: Generalist software engineering agents to solve coding tasks at scale," *arXiv preprint arXiv:2409.16299*, 2024.

[3] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran *et al.*, "Why do multi-agent llm systems fail?" *arXiv preprint arXiv:2503.13657*, 2025.

[4] D. Deshpande, V. Gangal, H. Mehta, J. Krishnan, A. Kannappan, and R. Qian, "Trail: Trace reasoning and agentic issue localization," *arXiv preprint arXiv:2505.08638*, 2025.

[5] M. Cemri *et al.*, "Why do multi-agent llm systems fail?" 2025, paper details to be finalized.

[6] Y. Ma *et al.*, "Diagnosing failure root causes in platform-orchestrated agentic systems: Dataset, taxonomy, and benchmark," 2025, paper details to be finalized.

[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[8] OpenTelemetry Authors, "Opentelemetry," 2026, official project website.

[9] F. Kong, R. Zhang, H. Yin, G. Zhang, X. Zhang, Z. Chen, Z. Zhang, X. Zhang, S.-C. Zhu, and X. Feng, "Aegis: Automated error generation and attribution for multi-agent systems," *arXiv preprint arXiv:2509.14295*, 2025.

[10] G. Zhang, J. Wang, J. Chen, W. Zhou, K. Wang, and S. Yan, "Agentracer: Who is inducing failure in the llm agentic systems?" *arXiv preprint arXiv:2509.03312*, 2025.

[11] R. Solomon, Y. Y. Levi, L. Vaknin, E. Aizikovich, A. Baras, E. Ohana, A. Giloni, S. Bose, C. Picardi, Y. Elovici *et al.*, "Lumimas: A comprehensive framework for real-time monitoring and enhanced observability in multi-agent systems," *arXiv preprint arXiv:2508.12412*, 2025.

[12] L. Dong, Q. Lu, and L. Zhu, "Agentops: Enabling observability of llm agents," *arXiv preprint arXiv:2411.05285*, 2024.

[13] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, 2024.