

AgentTelemetry: Unified Observability for Autonomous AI Agent Systems

Krishna Chaitanya Balusu
Independent Researcher
United States
krishnabkc15@gmail.com

Abstract

Autonomous AI agents built on large language models are transitioning from research prototypes to production deployments, yet observability infrastructure for these systems remains fundamentally inadequate. Standards such as OpenTelemetry lack semantic primitives for agent-specific phenomena including reasoning chains, tool invocations, inter-agent delegation, and cost accumulation. We present AGENTTELEMETRY, an open-source observability library that introduces (1) a telemetry data model with nine agent-specific span kinds and semantic attribute conventions; (2) W3C-compatible context propagation for multi-agent tracing; (3) a privacy-first design with opt-in content capture; and (4) a pluggable instrumentor-exporter architecture supporting seven major agent frameworks with export to any OTLP backend. Micro-benchmarks show instrumentation overhead under 13 μ s per span ($<0.003\%$ of typical LLM latency), and real-agent experiments with Claude confirm suitability for always-on deployment.

CCS Concepts: • Software and its engineering → Software verification and validation; Software post-development issues.

Keywords: AI agents, observability, telemetry, distributed tracing, LLM monitoring, OpenTelemetry, multi-agent systems

ACM Reference Format:

Krishna Chaitanya Balusu. 2026. AgentTelemetry: Unified Observability for Autonomous AI Agent Systems. In . ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. Conference'17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Large language model (LLM) based autonomous agents represent a paradigm shift in software architecture. Unlike conventional services that execute deterministic logic, an LLM agent pursues a goal through iterative reasoning, planning, tool use, and self-reflection—often delegating sub-tasks to other agents [4, 6]. Frameworks such as LangChain, CrewAI, AutoGen, and DSPy have accelerated adoption, and organizations are deploying agents for customer support, code generation, and autonomous operations [10].

This proliferation has exposed a critical gap: *there is no standardized observability infrastructure for AI agent systems*. Production teams cannot reliably answer fundamental questions: Why did the agent choose this tool? Where did the hallucination originate? How much did this task cost across all LLM calls?

OpenTelemetry (OTel) [11] has become the standard for distributed tracing in microservice architectures, but its SpanKind enum and semantic conventions have no representation for reasoning steps, LLM calls, tool invocations, or inter-agent messages.

We present AGENTTELEMETRY, an open-source library that fills this gap. Our contributions are:

1. A **telemetry data model** with nine agent-specific span kinds, semantic attribute conventions, and built-in cost estimation.
2. A **context propagation protocol** compatible with W3C Trace Context, extended for multi-agent boundaries.
3. A **privacy-first design** where prompt and completion content is never captured unless explicitly opted in.
4. A **pluggable instrumentor-exporter architecture** with implementations for seven agent frameworks (LangChain, CrewAI, AutoGen, DSPy, Anthropic Claude, SmoLAgents, LlamaIndex) and three export targets including OTLP.

AGENTTELEMETRY is a *production-oriented, framework-agnostic telemetry library* that bridges existing distributed tracing standards and the unique requirements of autonomous AI systems.

2 Motivation and Gap Analysis

An LLM agent's execution produces a deeply nested, non-deterministic trace of cognitive operations. A multi-agent research assistant—with planner, researcher, and writer agents—may produce dozens of spans across three or more agents,

Table 1. Agent concepts mapped to observability primitives. Bold entries are novel AGENTTELEMETRY contributions.

Agent Concept	OTel	AgentTelemetry
Agent	Service	agent.name, .role
Task execution	Trace	AgentTrace
Reasoning step	(none)	REASONING span
LLM call	(none)	LLM_CALL span
Tool invocation	(none)	TOOL_CALL span
Planning	(none)	PLANNING span
Self-reflection	(none)	REFLECTION span
Retrieval (RAG)	(none)	RETRIEVAL span
Agent message	(none)	AGENT_COMM span
Human-in-loop	(none)	HUMAN_INPUT span
Token count	(none)	llm.input/output_tokens
Cost	(none)	llm.cost_usd

accumulating thousands of tokens and non-trivial cost. Debugging a failure requires tracing backward across agent boundaries; without unified tracing, operators resort to ad-hoc logging that is inconsistent across frameworks and scales poorly.

OpenTelemetry provides trace and span identifiers, context propagation, and a rich exporter ecosystem. AGENTTELEMETRY is designed to be OTel-compatible, with spans mapping one-to-one to OTel spans via the OTLP exporter. However, OTel lacks the semantic layer needed for agent observability. Table 1 summarizes the gaps: OTel’s SpanKind has no equivalent for reasoning, planning, reflection, or retrieval; its semantic conventions lack attributes for token counts, model identity, cost, and agent role.

3 System Design

AGENTTELEMETRY is organized into three layers: a core telemetry model, a framework instrumentor layer, and an exporter layer. Figure 1 shows the architecture.

3.1 Data Model and Span Kinds

An AgentSpan encapsulates a unit of agent work, carrying a trace identifier, span identifier, optional parent span, name, span kind, start and end timestamps, terminal status, a set of semantic attributes, and an ordered sequence of events. The formal tuple definition and full span lifecycle are provided in Appendix A.

Spans progress through creation (ID allocation, timestamp), attribute setting during execution, event recording for discrete occurrences, completion with automatic cost estimation for LLM_CALL spans, and export to all registered backends.

The AgentSpanKind enumeration defines nine span types: TASK (root), REASONING (chain-of-thought), LLM_CALL (model invocation), TOOL_CALL (function execution), PLANNING (task decomposition), REFLECTION (self-critique), RETRIEVAL (RAG

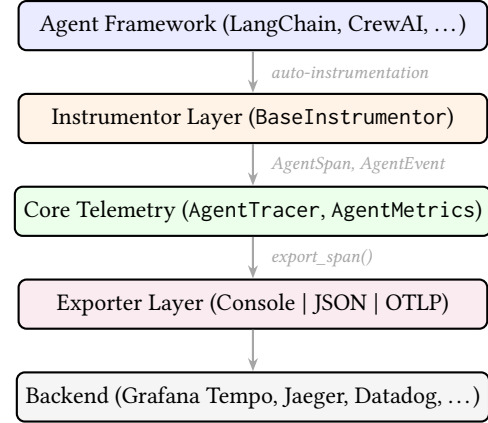


Figure 1. Layered architecture of AGENTTELEMETRY. Instrumentors are framework-specific; core and exporter layers are framework-agnostic.

lookup), AGENT_COMM (inter-agent message), and HUMAN_INPUT (human-in-the-loop). Each kind determines which semantic attributes are expected (e.g., LLM_CALL spans carry llm.* attributes) and enables kind-specific dashboards and alerting.

We define three attribute namespaces: agent.* for identity (name, framework, role), llm.* for model metadata (model, tokens, cost, latency), and tool.* for tool execution (name, input, output, success). Cost is auto-estimated from model name and token counts via a built-in pricing table. Spans may contain timestamped AgentEvent instances (e.g., ERROR, WARNING, GUARDRAIL_TRIGGERED, COST_THRESHOLD) for fine-grained intra-span signals. Full attribute tables are in Appendix A.

3.2 Context Propagation and Privacy

Multi-agent architectures require trace context to flow across agent boundaries. The AgentContext class serializes context to a carrier dictionary containing a W3C traceparent header, an agentstate header with the source agent’s name, and optional baggage metadata. The receiving agent deserializes the carrier so that subsequent spans inherit the trace ID and parent span reference, producing a unified trace tree across multiple agents.

AGENTTELEMETRY adopts a privacy-first posture: the llm.prompt and llm.completion attributes are *never populated* unless the operator explicitly sets capture_content=True. This opt-in gate applies at both tracer and instrumentor levels. When content capture is disabled, telemetry still records all structural information—span kind, duration, token counts, cost, model identity—providing comprehensive observability without exposing sensitive content.

Complementing trace data, the AgentMetrics class provides thread-safe counters, histograms, and gauges (see Appendix A).

Listing 1. Instrumentor usage: automatic telemetry requires no user code changes.

```

1 from agenttelemetry.instrumentors.langchain \
2     import LangChainInstrumentor
3 from agenttelemetry.exporters.otlp \
4     import OTLPExporter
5 inst = LangChainInstrumentor()
6 inst.tracer.add_exporter(
7     OTLPExporter(endpoint="collector:4317"))
8 inst.instrument()
9 agent = create_langchain_agent(...)
10 result = agent.invoke("Summarize this paper")
11 inst.uninstrument()

```

4 Implementation

4.1 Instrumentor Architecture

The instrumentor layer uses a template method pattern. The abstract base instrumentor class provides a default tracer and metrics instance, convenience methods for recording LLM and tool metrics, abstract `instrument()` and `uninstrument()` hooks for framework-specific patching, and a `capture_content` flag that propagates the privacy setting. Listing 1 shows the usage pattern: the instrumentor applies patches transparently and telemetry is collected without changes to user code.

Three exporters are provided: a console exporter for development (single-line span summaries), a JSON file exporter for offline analysis (JSON Lines format), and an OTLP exporter for production backends via gRPC. The exporter interface is a single `export_span()` method, making custom exporters straightforward. Pre-built Grafana dashboards for agent overview and trace detail are included (Appendix C).

The direct tracing API (AgentTracer) provides context-manager methods for each span kind, enabling manual instrumentation of custom agent implementations (see Appendix B).

4.2 Framework Support

We provide instrumentors for seven major agent frameworks. Table 2 summarizes which agent concepts each framework natively exposes and how AGENTTELEMETRY maps them to spans.

No single framework exposes all nine agent concepts. LangChain covers LLM calls, tools, and retrieval but lacks inter-agent communication; CrewAI provides multi-agent delegation but omits tool calls; AutoGen captures messaging and tools but has no task root; the Anthropic Claude instrumentor captures tool use blocks and extended thinking; SmoLAgents tracks code agent steps; and LlamaIndex provides deep query engine and retriever integration. AGENTTELEMETRY’s uniform span vocabulary enables cross-framework consistency: dashboards built for one framework work identically for another.

Table 2. Framework support: agent concepts exposed by each framework and mapping to AGENTTELEMETRY span kinds. ✓ = auto-instrumented; — = not natively present.

Concept	LangChain	CrewAI	AutoGen	Dspy	Claude	SmoLAgents	LlamaIndex
Task root	✓	✓	—	✓	✓	✓	✓
LLM call	✓	✓	✓	✓	✓	✓	✓
Tool call	✓	—	✓	—	✓	✓	✓
Reasoning	✓	✓	✓	✓	✓	✓	—
Retrieval	✓	—	—	✓	—	—	✓
Agent comm	—	✓	✓	—	—	—	—
Token count	✓	✓	✓	✓	✓	✓	✓
Cost track	—	✓	—	—	✓	—	—

Table 3. Per-operation overhead (10,000 ops, 5 trials, mean ± std).

Operation	Latency (μs/op)	
Span creation (bare)	12.15 ±	0.07
Span creation (w/ attributes)	12.57 ±	0.05
Context propagation (round-trip)	1.45 ±	0.01
ConsoleExporter (to /dev/null)	1.54 ±	0.03
JSONFileExporter (to temp file)	103.48 ±	6.29
Nested trace (4-deep)	28.95 ±	0.14
Cost estimation	0.35 ±	0.00

5 Evaluation

5.1 Instrumentation Overhead

We measured per-operation overhead on Apple Silicon (macOS 25.2, Python 3.9.6) over five independent trials of 10,000 iterations each. Table 3 presents the results.

Span creation costs ~12 μs, with negligible overhead for attributes. A four-level nested trace (task → reasoning → LLM call → tool call) costs ~29 μs total. Context propagation adds 1.45 μs per round trip. Since LLM API calls take 500 ms–5 s, the most expensive in-memory operation (span creation) represents <0.003% overhead, confirming suitability for always-on production deployment.

5.2 Real-Agent Experiment

To validate instrumentation on real workloads, we ran a tool-calling agent using the

Table 4. Debugging effectiveness: default logging vs. AGENT-TELEMETRY across three fault scenarios.

Metric	Default	AgentTelemetry
<i>S1: Hallucinated Retrieval</i>		
Root cause found	Rarely	Always
Cross-agent tracing	Manual	Automatic
<i>S2: Tool Timeout</i>		
Causal chain visible	Partial	Complete
Retry context captured	No	Yes
<i>S3: Cost Overrun</i>		
Per-iteration cost	Unavailable	Tracked
Threshold event	None	Automatic

(\$0.008/run), and tool success status. Baseline latency was $7,507 \pm 1,707$ ms; instrumented latency was $7,495 \pm 2,086$ ms—a difference of -13 ms (-0.17%), well within measurement noise. This confirms that instrumentation overhead is negligible relative to LLM API latency, consistent with our micro-benchmark results.

5.3 Debugging Effectiveness

We evaluated fault diagnosis using a simulated multi-agent pipeline (planner $\rightarrow$ researcher $\rightarrow$ writer) with three injected faults: (1) hallucinated retrieval (relevance score 0.12, threshold 0.5); (2) tool timeout with retry producing degraded results; and (3) cost overrun from an 8-iteration reasoning loop exceeding a \$1.00 budget. Table 4 summarizes the results.

Agent-specific span kinds (RETRIEVAL, TOOL_CALL, LLM_CALL, REASONING) combined with semantic attributes (relevance scores, cost accumulators) and structured events (WARNING, COST_THRESHOLD) enable rapid root-cause diagnosis that is impractical with generic logging. Cross-agent fault propagation and per-iteration cost attribution are fundamentally impossible without structured multi-agent telemetry.

5.4 Threats to Validity

Our evaluation has several limitations. The micro-benchmarks were conducted on a single machine (Apple Silicon); overhead may vary on different architectures. The real-agent experiment uses a single framework (Anthropic Claude SDK); while the instrumentor architecture is framework-agnostic, overhead characteristics may differ across frameworks. The debugging case study uses simulated fault injection with controlled latencies rather than production failures. We did not evaluate multi-machine distributed tracing, though the W3C-compatible context propagation is designed for this scenario.

6 Related Work

System-Level Agent Monitoring. AgentSight [1] uses eBPF to monitor TLS-encrypted LLM traffic and correlate kernel events with agent intent. While it provides deep system-level visibility, it requires Linux kernel access often unavailable in containers and cannot capture application-level semantics (agent role, task decomposition). AGENTTELEMETRY operates at the application level; the two approaches are complementary.

Runtime Safety and Constraints. NeMo Guardrails [2] intercepts and validates agent behavior; DSPy Assertions [3] embed constraints into LLM pipelines. Both enforce policies but do not produce structured, exportable telemetry for dashboards or SLO monitoring.

Agent Evaluation. AgentBench [6], AgentBoard [4], and Agent-as-a-Judge [5] target offline evaluation, not continuous production monitoring.

Commercial Platforms. LangSmith [16] provides tracing for LangChain but is ecosystem-specific and proprietary. OpenLLMetry [17] extends OTel with LLM instrumentation but focuses on individual API calls without agent-level abstractions (task, reasoning, inter-agent communication). AGENTTELEMETRY is fully open-source, framework-agnostic, standards-based (OTLP), and provides nine agent-specific span kinds that capture the full complexity of multi-agent systems.

7 Conclusion

We have presented AGENTTELEMETRY, an open-source framework for unified observability of autonomous AI agent systems. By introducing agent-specific span kinds, semantic conventions, W3C-compatible context propagation, and privacy-first design, it bridges the gap between distributed tracing standards and LLM-based agent workloads. Its instrumentor-exporter architecture supports seven frameworks with export to any OTLP backend.

Future directions include proposing our semantic conventions as an OpenTelemetry Enhancement Proposal (OTEP), adaptive sampling that retains anomalous traces [14], real-time span streaming for long-running tasks, and applying causal inference [15] to automated root cause analysis.

References

- [1] Y. Zheng, Y. Hu, T. Yu, and A. Quinn. AgentSight: System-Level Observability for AI Agents Using eBPF. *arXiv preprint arXiv:2508.02736*, 2025.
- [2] T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, and J. Cohen. NeMo Guardrails: A Toolkit for Controllable and Safe LLM Applications. In *Proc. EMNLP (Demo Track)*, 2023.
- [3] A. Singhvi, M. Shetty, S. Tan, C. Potts, K. Sen, M. Zaharia, and O. Khat-tab. DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines. In *Proc. COLM*, 2024.

- [4] C. Ma, J. Zhang, Z. Zhu, C. Yang, Y. Yang, et al. AgentBoard: An Analytical Evaluation Board of Multi-Turn LLM Agents. In *Proc. ACL*, 2024.
- [5] M. Zhuge, C. Zhao, D. Ashley, et al. Agent-as-a-Judge: Evaluate Agents with Agents. *arXiv preprint arXiv:2410.10934*, 2024.
- [6] X. Liu, H. Yu, H. Zhang, et al. AgentBench: Evaluating LLMs as Agents. In *Proc. ICLR*, 2024.
- [7] S. Vijayvargiya, A. B. Soni, X. Zhou, Z. Z. Wang, N. Dziri, G. Neubig, and M. Sap. OpenAgentSafety: A Comprehensive Framework for Evaluating Real-World AI Agent Safety. *arXiv preprint arXiv:2507.06134*, 2025.
- [8] Y. Ruan, H. Dong, A. Wang, S. Pitis, et al. ToolEmu: Identifying Risks of LLM Agents with an LLM-Emulated Sandbox. In *Proc. ICLR*, 2024.
- [9] J. Leest, I. Gerostathopoulos, P. Lago, and C. Raibulet. Monitoring and Observability of Machine Learning Systems: Current Practices and Gaps. *arXiv preprint arXiv:2510.24142*, 2025.
- [10] L. Zhang, T. Jia, M. Jia, Y. Wu, A. Liu, Y. Yang, Z. Wu, X. Hu, P. S. Yu, and Y. Li. A Survey of AIOps in the Era of Large Language Models. *arXiv preprint arXiv:2507.12472*, 2025.
- [11] OpenTelemetry Authors. OpenTelemetry Specification, v1.30. <https://opentelemetry.io/docs/specs/otel/>, 2024.
- [12] R. Ding, C. Zhang, L. Wang, Y. Xu, M. Ma, W. Zhang, S. Qin, S. Rajmohan, Q. Lin, and D. Zhang. TraceDiag: Adaptive, Interpretable, and Efficient Root Cause Analysis on Large-Scale Microservice Systems. In *Proc. ACM ESEC/FSE*, 2023.
- [13] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *Proc. ACM ICSE*, 2022.
- [14] D. Batavia, I. Weber, et al. Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems. In *Proc. IEEE ICWS*, 2023.
- [15] L. Zheng, Z. Chen, J. He, and H. Chen. MULAN: Multi-modal Causal Structure Learning for Root Cause Analysis. In *Proc. NeurIPS*, 2024.
- [16] LangChain Inc. LangSmith: Platform for Building Production-Grade LLM Applications. <https://smith.langchain.com/>, 2024.
- [17] Traceloop. OpenLLMetry: Open-Source Observability for LLM Applications Built on OpenTelemetry. <https://github.com/traceloop/openllmetry>, 2024.

A Formal Data Model

Formal Definition. We define an AgentSpan as a tuple:

$$s = (t, s_{id}, p_{id}, name, k, t_0, t_1, \sigma, A, E)$$

where $t \in \{0, 1\}^{128}$ is the trace identifier, $s_{id} \in \{0, 1\}^{64}$ is the span identifier, $p_{id} \in \{0, 1\}^{64} \cup \{\perp\}$ is the optional parent span identifier, $name$ is a human-readable label, $k \in \mathcal{K}$ is the span kind drawn from the AgentSpanKind enumeration, t_0 and t_1 are nanosecond-precision timestamps, $\sigma \in \{\text{UNSET}, \text{OK}, \text{ERROR}, \text{TIMEOUT}\}$ is the terminal status, $A : \text{String} \rightarrow \text{Value}$ is a partial function mapping attribute keys to values, and E is an ordered sequence of AgentEvent instances.

Span Lifecycle. An AgentSpan progresses through five phases:

1. **Creation.** The tracer allocates the span, assigns s_{id} , inherits t and p_{id} from the active span stack, and records $t_0 = \text{now}()$.
2. **Attribute Setting.** The caller attaches metadata via `set_attribute(key, value)`. Attributes are additive and last-writer-wins.

3. **Event Recording.** Discrete occurrences (errors, guardrail triggers, cost thresholds) are captured via `add_event()`.
4. **Completion.** The context manager records t_1 and promotes σ to OK if still UNSET. For LLM_CALL spans, cost is auto-estimated.
5. **Export.** The span is serialized and dispatched to all registered exporters.

Semantic Attribute Conventions. Three attribute namespaces are defined:

agent.* Agent identity: `agent.name`, `agent.framework`, `agent.framework_version`, `agent.role`, `agent.task`.

llm.* LLM metadata: `llm.model`, `llm.provider`, `llm.input_tokens`, `llm.output_tokens`, `llm.total`

agent; stacked bar charts for Token Usage by Model; and pie charts for Cost Breakdown by Model.

Agent Trace Detail Dashboard. The trace detail dashboard enables deep inspection of individual traces using Grafana Tempo, Prometheus, and Loki. Panels include per-trace summary statistics, a span waterfall timeline, sortable span detail tables, LLM Call and Tool Call detail views, a node graph for multi-agent topology, and event/error log panels.

D Extended Case Study Narratives

Scenario 1: Hallucinated Retrieval. The researcher agent's RAG vector search returns documents with relevance score 0.12 (threshold 0.5). The AGENTTELEMETRY trace reveals the fault immediately: the RETRIEVAL span carries `retrieval.relevance_score=0.12` and a WARNING event. The downstream LLM_CALL span shows `llm.context_quality=degraded`. Following the parent-child chain from the writer's TASK span back through the researcher to the RETRIEVAL span localizes the root cause. Without structured telemetry, only the writer's incorrect output is visible, and diagnosis requires manual correlation across three agents' logs.

Scenario 2: Tool Timeout with Retry. A `web_search` tool call times out after 30 s; the researcher retries with a simplified query, receiving only 2 results instead of 10. The trace shows: a TOOL_CALL span with `status=ERROR` and

`tool.error=TimeoutError`, a retry WARNING event, a second TOOL_CALL with reduced results, and an LLM_CALL with `context_quality=degraded`.

Scenario 3: Cost Overrun. The researcher enters a self-critique loop iterating 8 times (expected 2–3), each iteration making an LLM_CALL with growing token counts. Each iteration is captured as a nested REASONING span with `reasoning.iteration` and `reasoning.cumulative_cost_usd` attributes. A `COST_THRESHOLD` event fires at the exact iteration where cost exceeds \$1.00.

E Extended Related Work

Agent Safety. ToolEmu [8] emulates tool execution for automated safety testing. OpenAgentSafety [7] evaluates agent safety across 350+ tasks, finding unsafe behavior in over 51% of unsafe and 12% of safe tasks. These findings underscore the need for runtime observability beyond development-time evaluation.

ML System Monitoring. Leest et al. [9] documented significant gaps in ML monitoring practice, motivating standardized tools for ML-specific observability.

AI Ops and Distributed Tracing. TraceDiag [12] and DeepTraLog [13] analyze distributed traces for fault diagnosis in microservices. AGENTTELEMETRY's OTLP export enables these tools to be applied directly to agent telemetry data.