

AgentTelemetry: A Grounded Taxonomy and Empirical Evaluation of Observability Primitives 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 the observability infrastructure needed to diagnose their failures remains fundamentally inadequate. OpenTelemetry’s GenAI semantic conventions now cover LLM invocation, tool execution, retrieval, and agent lifecycle, but leave five critical agent orchestration phases—planning, reasoning, safety checking, inter-agent delegation, and memory management—without any span-level representation.

We present AGENTTELEMETRY, an open-source, OTel-native observability library that introduces a *grounded taxonomy* of nine agent-specific span kinds—AGENT, LLM_CALL, TOOL_CALL, PLANNING, REASONING, RETRIEVAL, GUARD_RAIL, DELEGATION, and MEMORY—derived from systematic analysis of seven production agent frameworks.

We make four contributions: (1) the taxonomy itself, empirically validated through an ablation study showing that removing any single span kind makes at least one fault type undetectable; (2) a privacy-preserving multi-agent context propagation scheme extending W3C Trace Context; (3) a controlled fault-injection study across 14 fault types, 6 LLM models, 7 frameworks, and 4 observability conditions demonstrating that AGENTTELEMETRY achieves a Fault Detection Rate (FDR) of 1.000 under controlled fault injection compared to 0.429 for vanilla OpenTelemetry and 0.000 without telemetry; and (4) a pip-installable library (3,700+ LOC, 78 tests) with adapters for seven frameworks, including an

end-to-end validated LangChain integration producing correct spans from real AgentExecutor traces. A case study on 112 SWE-bench Lite instances (37% of the benchmark, across all 12 repositories) reveals that reasoning loops account for 75% of agent failures (95% CI: [66%, 82%])—a failure mode invisible to vanilla OTel but precisely characterized by AGENTTELEMETRY’s novel REASONING span kind. A telemetry-guided intervention that detects and breaks these loops improves the patch rate by +11 pp over a no-intervention control (3-seed mean, $p < 0.05$), demonstrating that the observability is actionable. Instrumentation overhead is 11.7 μ s median ($p99 < 30 \mu$ s) per span, negligible relative to LLM API latencies of 500 ms–10 s.

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

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

ACM Reference Format:

Krishna Chaitanya Balusu. 2026. AgentTelemetry: A Grounded Taxonomy and Empirical Evaluation of Observability Primitives for Autonomous AI Agent Systems. In . ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

Code and data: <https://github.com/agenttelemetry/agenttelemetry>

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—frequently delegating sub-tasks to other agents [4, 6]. Frameworks such as LangChain, CrewAI, AutoGen, and LlamaIndex 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*.

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/nnnnnnnn.nnnnnnnn>

Production teams cannot reliably answer fundamental diagnostic questions:

- **Why did the agent choose this tool?** Decision attribution requires tracing from a tool invocation back through the LLM call that selected it and the reasoning chain that informed it.
- **Where did the hallucination originate?** Hallucination tracing requires comparing LLM output against retrieved documents, which demands semantic distinction between retrieval and generation spans.
- **How much did this task cost across all agents?** Cost attribution requires agent identity propagation combined with per-call token and pricing metadata.

OpenTelemetry (OTel) [11] has become the standard for distributed tracing in microservice architectures. The OTel GenAI semantic conventions define operation types for LLM calls (chat, text_completion), retrieval (retrieval, embeddings), tool execution (execute_tool), and agent lifecycle (create_agent, invoke_agent). However, these conventions address individual operations, not the higher-order *orchestration phases* that characterize autonomous agent execution. Five such phases—planning, reasoning, safety checking, inter-agent delegation, and memory management—have no representation in OTel GenAI, leaving them as opaque INTERNAL spans indistinguishable from any other application logic. This semantic opacity makes automated fault detection substantially more difficult.

We present AGENTTELEMETRY, an open-source, OTel-native library that fills this gap with four contributions:

1. **A grounded taxonomy of nine agent-specific span kinds**, derived from systematic analysis of seven production frameworks and validated through an ablation study proving each kind is necessary for detecting at least one fault type (§3).
2. **A lightweight multi-agent context propagation scheme** extending W3C Trace Context with agent identity baggage and three-level privacy controls (§4).
3. **A controlled fault-injection evaluation** across 14 fault types, 6 models, 7 frameworks, and 4 observability conditions, demonstrating FDR = 1.000 versus 0.429 for vanilla OTel and 0.000 without telemetry (§6).
4. **An open-source, pip-installable library** (3,700+ LOC, 78 tests) with seven framework adapters—one reference implementation providing full fault coverage, plus six adapter-validated stubs—using three distinct instrumentation strategies (§5).

2 Background and Motivation

2.1 Agent Execution Model

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

Table 1. Agent concepts mapped to observability primitives. • = overlapping with OTel GenAI semantic conventions; ▷ = extends OTel GenAI; ★ = novel, no OTel GenAI equivalent.

Agent Concept	OTel GenAI	AgentTelemetry	
Agent lifecycle	create/invoke_agent	AGENT	▷
LLM invocation	chat / text_compl.	LLM_CALL	•
Tool execution	execute_tool	TOOL_CALL	•
Doc. retrieval	retrieval, embeddings	RETRIEVAL	•
Task decomposition	(none)	PLANNING	★
Chain-of-thought	(none)	REASONING	★
Safety/policy check	(none)	GUARD_RAIL	★
Inter-agent handoff	(none)	DELEGATION	★
Memory read/write	(none)	MEMORY	★
Token count	gen_ai.usage.*	llm.input/output_tokens	•
Cost estimation	(none)	llm.cost	★
Agent identity	gen_ai.agent.*	agent.name, .role	▷

agents, 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.

2.2 The OpenTelemetry Gap

OpenTelemetry provides trace and span identifiers, context propagation, and a rich exporter ecosystem. However, its semantic layer was designed for network-centric microservices. The OTel GenAI semantic conventions [11] (all currently at Development stability) have made significant progress: they define eight gen_ai.operation.name values—chat, text_completion, embeddings, retrieval, execute_tool, generate_content, create_agent, and invoke_agent—covering LLM invocation, vector retrieval, tool execution, and agent lifecycle.

Despite this breadth, the GenAI conventions address only *individual operation types* and leave a critical gap in *agent orchestration phases*. Table 1 summarizes: five of nine AGENTTELEMETRY span kinds—PLANNING, REASONING, GUARD_RAIL, DELEGATION, and MEMORY—have no OTel GenAI equivalent at all. A sixth (AGENT) extends OTel’s invoke_agent with framework, role, and orchestration semantics. Without these distinctions, a planning step and a summarization call are both opaque chat spans; a safety check and a reasoning chain are both undifferentiated INTERNAL spans.

3 A Grounded Taxonomy of Agent Span Kinds

3.1 Derivation Methodology

The nine span kinds were derived through a structured qualitative analysis loosely modeled on open coding [18]: we examined framework APIs, generated candidate span kinds

from observed execution phases, iteratively merged and refined candidates across frameworks, and stopped when no new kinds emerged. The process proceeded in four stages.

Stage 1: API inventory. For each of the seven frameworks, we catalogued every public API entry point that represents a distinct execution phase in an agent loop. Table 2 lists the 25 entry points examined. Callback-based frameworks (LangChain, CrewAI, LlamaIndex) expose named hooks (e.g., `on_retriever_start`, `register_before_tool_call_hook`); thin SDK wrappers (Anthropic, OpenAI) expose a single `create()` method whose response contains typed blocks (`tool_use`, `tool_calls`); orchestration frameworks (AutoGen) expose agent lifecycle methods (`BaseChatAgent.run()`, `ChatCompletionClient.create()`).

Stage 2: Open coding. Each API entry point was assigned a candidate span kind label describing the execution phase it represents. This produced 14 initial candidates: `AGENT`, `LLM_CHAT`, `LLM_COMPLETION`, `TOOL_CALL`, `PLANNING`, `REASONING`, `REFLECTION`, `RETRIEVAL`, `GUARD_RAIL`, `DELEGATION`, `AGENT_COMM`, `READ_MEMORY`, `WRITE_MEMORY`, and `HUMAN_INPUT`.

Stage 3: Axial coding and merging. We merged candidates using two criteria: (a) *observational indistinguishability*—if two candidates cannot be reliably distinguished from framework-emitted signals alone, they are merged; and (b) *diagnostic redundancy*—if two candidates trigger the same fault-detection logic, the more specific one is absorbed. Five merges occurred:

- `LLM_CHAT` and `LLM_COMPLETION` → `LLM_CALL`: both represent an LLM invocation; the chat-vs-completion distinction adds no diagnostic value (same token/cost/latency detection logic applies).
- `REFLECTION` → merged into `REASONING`: both are introspective LLM steps; the distinction is not reliably observable across frameworks (LangChain emits both as `on_chain_start` with no flag separating reflection from reasoning).
- `AGENT_COMM` → renamed to `DELEGATION`: the observable signal is a task handoff between agents, not a generic message exchange; “delegation” captures the semantic more precisely.
- `READ_MEMORY` and `WRITE_MEMORY` → `MEMORY` with a `memory.operation` attribute: this follows the OTel convention of fewer span kinds with distinguishing attributes (cf. `db.operation` in OTel database conventions).
- `HUMAN_INPUT` → removed: human-in-the-loop interaction is not an agent execution phase; it is better modeled as an `AGENT` span with `agent.type=“human”`.

Stage 4: Saturation check. After processing the first five frameworks (LangChain, CrewAI, AutoGen, LlamaIndex, Anthropic SDK), the set of span kinds had stabilized at nine. The sixth framework (OpenAI SDK) and seventh (Custom manual API) introduced no new candidates—only new entry points mapping to existing kinds. We consider the taxonomy saturated at $N = 5$ frameworks with two confirmation passes.

A second coder independently classified all 25 entry points, achieving Cohen’s $\kappa = 0.904$ (almost perfect agreement; see §??).

Why nine and not fewer? Merging `PLANNING` into `REASONING` would lose the ability to distinguish task decomposition (“break this into 3 sub-queries”) from chain-of-thought (“the answer is X because Y”). The ablation study (§6.3) confirms both are needed: removing `PLANNING` makes planning-failure faults undetectable, while removing `REASONING` makes reasoning-faults undetectable.

Why nine and not more? We considered splitting `LLM_CALL` into `LLM_CHAT` and `LLM_COMPLETION`, but the same detection logic (token overflow, cost explosion, hallucination) applies to both. Adding the split would increase adapter complexity without improving fault coverage.

Cross-validation with competing taxonomies. Table 3 maps two recent agent taxonomies to ours. AgentDebug’s five competency categories [19] map directly: `memory`→`MEMORY`, `reflection`→`REASONING`, `planning`→`PLANNING`, `action`→`TOOL_CALL`, `system-level`→`AGENT`. The OTel GenAI semantic conventions define eight operation types, three of which overlap with our taxonomy (`LLM_CALL`↔`chat/text_completion`, `TOOL_CALL`↔`execute_tool`, `RETRIEVAL`↔`retrieval/embeddings`) and one that our `AGENT` kind extends (`invoke_agent`). Crucially, five of our nine kinds—`PLANNING`, `REASONING`, `GUARD_RAIL`, `DELEGATION`, and `MEMORY`—are entirely *novel*: no OTel GenAI operation or span type captures these agent orchestration phases. Our ablation study (§6.3) confirms all five are necessary for complete fault detection. MAST’s 14 failure modes [20] operate at a different abstraction level (failure classification, not execution phases) but 12 of 14 (85.7%) are detectable via our span kinds and fault types (Table 4). The two gaps—failure to ask for clarification (FM-2.2) and missing verification (FM-3.2)—are *omission* failures that require expected-span specifications beyond trace-level anomaly detection. Conversely, three of our fault types (cost explosion, tool failure, timeout) address operational failures absent from MAST’s behavioral taxonomy but critical for production monitoring.

3.2 The Nine Span Kinds

We represent agent span kinds as semantic attributes on OTel `INTERNAL` spans via the `agenttelemetry.span.kind` attribute. This design is OTel-native: all spans are standard OTel spans that export via OTLP to any compatible backend. Of the nine kinds (Table 5), three overlap with OTel GenAI operations (`LLM_CALL`, `TOOL_CALL`, `RETRIEVAL`), one extends an existing GenAI operation (`AGENT`), and **five are novel contributions** with no GenAI equivalent: `PLANNING`, `REASONING`, `GUARD_RAIL`, `DELEGATION`, and `MEMORY`. These five capture the orchestration phases that distinguish autonomous agents from simple LLM API wrappers.

Operational semantics of `PLANNING` vs. `REASONING`. `PLANNING` spans wrap LLM calls that produce *structured action plans*—task decompositions, sub-query lists, or

Table 2. Traceability matrix: framework API entry points mapped to final span kinds. 25 entry points across 7 frameworks produce 9 span kinds.

Framework	API Entry Point	Candidate	Final Kind
LangChain	on_chain_start (agent/executor)	Agent	AGENT
	on_chain_start (other chains)	Reasoning	REASONING
	on_llm_start	LLM_Chat	LLM_CALL
	on_chat_model_start	LLM_Chat	LLM_CALL
	on_tool_start	Tool_Call	TOOL_CALL
	on_retriever_start	Retrieval	RETRIEVAL
	on_agent_action	Planning	PLANNING
CrewAI	before_llm_call_hook	LLM_Call	LLM_CALL
	before_tool_call_hook	Tool_Call	TOOL_CALL
AutoGen	ChatCompletionClient.create()	LLM_Call	LLM_CALL
	BaseChatAgent.run()	Agent	AGENT
LlamaIndex	span_enter (llm/predict/chat)	LLM_Compl.	LLM_CALL
	span_enter (retrieve/retriever)	Retrieval	RETRIEVAL
	span_enter (sub_question)	Planning	PLANNING
	span_enter (query/queryengine)	Agent	AGENT
Anthropic	span_enter (default)	Reasoning	REASONING
	Messages.create()	LLM_Call	LLM_CALL
OpenAI	Response tool_use blocks	Tool_Call	TOOL_CALL
	Completions.create()	LLM_Call	LLM_CALL
Custom	Response tool_calls	Tool_Call	TOOL_CALL
	start_agent_span_ctx()	Agent	AGENT
Custom	start_planning_span()	Planning	PLANNING
	start_guardrail_span()	Guard_Rail	GUARD_RAIL
	start_delegation_span()	Delegation	DELEGATION
	start_memory_span()	Memory	MEMORY

Table 3. Cross-validation: mapping competing taxonomies to AGENTTELEMETRY span kinds. “–” indicates no equivalent. ★ = novel contribution absent from OTEL GenAI.

Our Kind	AgentDebug	OTel GenAI	Novelty
AGENT	system-level	invoke_agent	Extended
LLM_CALL	–	chat, text_compl.	Overlap
TOOL_CALL	action	execute_tool	Overlap
PLANNING	planning	–	★
REASONING	reflection	–	★
RETRIEVAL	–	retrieval, embed.	Overlap
GUARD_RAIL	–	–	★
DELEGATION	–	–	★
MEMORY	memory	–	★

step sequences. Adapters identify these via framework callbacks (e.g., LangChain’s `on_agent_action`, LlamaIndex’s `sub_question` span tag) or explicit `plan()` calls in the manual API. REASONING spans wrap *chain-of-thought steps within a single action*—intermediate inference, reflection, or self-critique. Adapters identify these via callbacks such as `on_llm_start` within an agent loop or `on_chain_start` for non-executor chains. In ReAct-style agents, the adapter creates one LLM_CALL span for the combined LLM output; the parent span is PLANNING when the output contains an Action: block, and REASONING

when the output contains only Thought: blocks. This distinction is diagnostically necessary: the ablation study (§6.3) confirms that removing PLANNING makes planning-failure faults undetectable, while removing REASONING makes reasoning-loop faults undetectable.

3.3 Cross-Framework Validation

No single framework natively exposes all nine execution phases. Table 6 shows the mapping from framework concepts to span kinds. LangChain covers LLM calls, tools, and retrieval but lacks inter-agent delegation; CrewAI provides multi-agent task handoffs but requires hooks for tool calls; AutoGen captures agent messaging but has no explicit planning phase. The taxonomy is framework-agnostic by construction: it captures *what* the agent does (which execution phase), not *how* (which framework API).

4 Multi-Agent Context Propagation

4.1 Agent Identity in Distributed Traces

Standard W3C Trace Context propagates traceparent and tracestate headers across service boundaries. This works for microservices communicating via HTTP or gRPC, but autonomous agents introduce new challenges: (1) agent identity—when Agent A delegates to Agent B, we need to know which agent owns each span; (2) delegation chains that may contain cycles (A→B→A); and (3) privacy boundaries where different agents operate under different content capture policies.

We extend the W3C model with two OTEL baggage keys: `agenttelemetry.agent_id` and `agenttelemetry.parent_agent_id`. These use the standard W3CBaggagePropagator—no new wire format is needed. When Agent A delegates to Agent B, A injects its `agent_id` as `parent_agent_id`; B sets its own `agent_id` upon extraction. The delegation relationship is recorded in DELEGATION spans with `source_agent` and `target_agent` attributes, enabling cycle detection.

4.2 Privacy-Preserving Telemetry

A `PrivacyLevel` enum controls attribute filtering at the span level:

The default is `METADATA_ONLY`: diagnostics without content exposure. Privacy filtering is implemented as a span attribute filter, not a separate propagation protocol. The privacy level can differ per agent in a multi-agent system, respecting data minimization.

5 Implementation

AGENTTELEMETRY is a pip-installable Python package (3,700+ LOC, 78 tests, MIT license) built on the OpenTelemetry SDK.

5.1 Architecture

All nine span kinds are represented as string values in the `agenttelemetry.span.kind` attribute on standard OTEL INTERNAL

Table 4. Mapping MAST failure modes [20] to AgentTelemetry fault types. 12 of 14 MAST modes (85.7%) are detectable; the two gaps are omission failures requiring expected-span specifications. Three of our fault types (cost explosion, tool failure, timeout) address operational failures absent from MAST’s behavioral taxonomy.

MAST Failure Mode	Code	Our Fault Type(s)	Detecting Span Kind(s)	Coverage
<i>(i) Specification Issues</i>				
Disobey Task Specification	FM-1.1	wrong_tool, hallucination	TOOL_CALL, LLM_CALL	Full
Disobey Role Specification	FM-1.2	agent_misroute, guardrail_bypass	AGENT, GUARD_RAIL	Full
Step Repetition	FM-1.3	infinite_loop, reasoning_loop	TOOL_CALL, REASONING	Full
Loss of Conversation History	FM-1.4	context_overflow, memory_corruption	LLM_CALL, MEMORY	Full
Unaware of Termination	FM-1.5	infinite_loop, planning_failure	TOOL_CALL, PLANNING	Full
<i>(ii) Inter-Agent Misalignment</i>				
Conversation Reset	FM-2.1	memory_corruption, context_overflow	MEMORY, LLM_CALL	Partial
Fail to Ask for Clarification	FM-2.2	—	—	Gap
Task Derailment	FM-2.3	wrong_tool, planning_failure	TOOL_CALL, PLANNING	Partial
Information Withholding	FM-2.4	circular_delegation	DELEGATION	Partial
Ignored Other Agent’s Input	FM-2.5	agent_misroute, circ. deleg.	AGENT, DELEGATION	Partial
Reasoning-Action Mismatch	FM-2.6	wrong_tool, hallucination	TOOL_CALL, REASONING	Full
<i>(iii) Task Verification</i>				
Premature Termination	FM-3.1	timeout, planning_failure	LLM_CALL, PLANNING	Partial
No or Incomplete Verification	FM-3.2	—	—	Gap
Incorrect Verification	FM-3.3	guardrail_bypass, hallucination	GUARD_RAIL, LLM_CALL	Full

Table 5. The nine agent-specific span kinds with representative semantic attributes. ★ = novel (no OTel GenAI equivalent).

#	Span Kind	Key Attributes
1	AGENT	agent.name, agent.framework, agent.role
2	LLM_CALL	llm.model, llm.input_tokens, llm.output_tokens, llm.cost
3	TOOL_CALL	tool.name, tool.input, tool.output, tool.status
4	PLANNING ★	planning.strategy, planning.step_count
5	REASONING ★	reasoning.chain
6	RETRIEVAL	retrieval.source, retrieval.query, retrieval.doc_count
7	GUARD_RAIL ★	guardrail.name, guardrail.result
8	DELEGATION ★	delegation.source_agent, delegation.target_agent
9	MEMORY ★	memory.operation, memory.key

spans. This design ensures that OTLP export, Jaeger visualization, and any OTel-compatible backend work with no modifications.

Table 6. Framework coverage: which agent execution phases each framework natively exposes (✓) vs. requires adapter mapping (◊).

Span Kind	LangChain	CrewAI	AutoGen	LlamaIndex	Anthropic	OpenAI	Custom
AGENT	✓	✓	✓	✓	◊	◊	✓
LLM_CALL	✓	✓	✓	✓	✓	✓	✓
TOOL_CALL	✓	◊	✓	✓	✓	✓	✓
PLANNING	◊	◊	◊	✓	◊	◊	✓
REASONING	✓	✓	✓	◊	◊	◊	✓
RETRIEVAL	✓	◊	◊	✓	◊	◊	✓
GUARD_RAIL	◊	◊	◊	◊	◊	◊	✓
DELEGATION	◊	✓	✓	◊	◊	◊	✓
MEMORY	◊	◊	◊	◊	◊	◊	✓

Table 7. Privacy levels and the data they capture.

Level	Captured Data
NONE	Structural only: span kind, timing, status
METADATA_ONLY	+ model name, token counts, tool names, costs
FULL	+ prompt/completion text, tool I/O content

The library comprises three modules: **Core** (span definitions, privacy filtering, context propagation, exporters, tracer factory), **Adapters** (seven framework-specific instrumentors), and **Analysis** (decision attribution, hallucination tracing, cost aggregation, anomaly detection).

Table 8. Adapter strategies, frameworks, and code complexity.

Strategy	Frameworks	LOC
Callback handlers	LangChain, CrewAI, LlamaIndex	903
Monkey-patching	Anthropic SDK, OpenAI SDK, AutoGen	795
Manual API	Custom agents	315

5.2 Three Adapter Strategies

Building adapters for seven frameworks, we identified three distinct instrumentation strategies (Table 8):

Callback handlers register event listeners via framework-provided extension APIs (e.g., LangChain’s `BaseCallbackHandler`, CrewAI’s hook decorators, LlamaIndex’s `SpanHandler`). This is the least invasive strategy. **Monkey-patching** wraps key methods (e.g., `Messages.create()` for Anthropic) at runtime, necessary for thin SDK clients without callback systems. **Manual API** provides context managers (`start_agent_span()`) for explicit instrumentation. All framework imports occur inside `_instrument()`, avoiding import-time dependencies.

5.3 Analysis Modules

Four analysis modules operate on exported span data: **CostAggregator** sums token costs by model, agent, and trace; **DecisionAttributor** links each `TOOL_CALL` to its parent `LLM_CALL` and nearby `REASONING` spans; **AnomalyDetector** detects circular delegation, infinite retries, cost explosion, and context overflow; **HallucinationTracer** compares `LLM_CALL` output against `RETRIEVAL` content using lightweight token-overlap heuristics suitable for trace-level screening, not production-grade NLI-based detection; it flags candidate ungrounded claims for human review rather than issuing definitive hallucination verdicts.

6 Evaluation

We evaluate `AGENTTELEMETRY` through six research questions:

- RQ1** Does AgentTelemetry improve fault detection over vanilla OTel?
- RQ2** Are all nine span kinds necessary?
- RQ3** What is the instrumentation overhead?
- RQ4** Can AgentTelemetry answer motivating diagnostic questions?
- RQ5** Does AgentTelemetry work with real LLM APIs across providers?
- RQ6** Does AgentTelemetry enable diagnosis and remediation of real failures?

6.1 Experimental Setup

Task. A standardized research assistant agent that receives a question, plans search queries, retrieves documents, calls tools, reasons over results, checks safety constraints, and generates an answer. This exercises all nine span kinds.

Models. Six LLMs across three capability tiers and two providers: Haiku 4.5, Sonnet 4.6, Opus 4.6 (Anthropic); GPT-4o-mini, O3-mini, GPT-4o (OpenAI).

Frameworks. Seven adapters: LangChain, CrewAI, AutoGen, LlamaIndex, Anthropic SDK, OpenAI SDK, and a Custom manual API. The Custom adapter serves as the *reference implementation* with full fault coverage across all 14 fault types. The remaining six are adapter-validated stubs that exercise the instrumentation layer (span creation, attribute propagation, export) through manual instrumentation rather than running actual framework code paths; this validates adapter correctness without requiring framework installation but does not test framework-internal callbacks.

Reproducibility. All benchmarks use deterministic mock LLM clients that return responses based on input hash, ensuring reproducibility without API keys. The fault injector records ground truth for every injected fault. All code, configurations, and mock data are open-source.

6.2 RQ1: Fault Detection Rate

Method. We define 14 fault types spanning all nine span kinds (Table 9). Each fault is injected at rate 1.0 into the mock client or the application’s span creation logic. We measure binary Fault Detection Rate (FDR): 1 if the injected fault is detected in the trace, 0 otherwise.

Conditions. Five observability levels: (a) `no_telemetry`—no instrumentation; (b) `vanilla_otel`—standard OTel `INTERNAL` spans without agent-specific attributes; (c) `otel_genai`—OTel GenAI semantic conventions (`gen_ai.*` attributes, 4 operation types); (d) `metadata_only`—`AGENTTELEMETRY` with `METADATA_ONLY` privacy; (e) `full_capture`—`AGENTTELEMETRY` with `FULL` privacy.

Scale. 14 faults $\times$ 5 conditions $\times$ 7 frameworks $\times$ 6 models = 2,940 configurations, all executed with zero errors. We further validate with 5 random seeds at 5 injection rates (10%–100%) for statistical rigor.

Results. Table 10 presents the FDR by condition for the reference implementation (custom framework) averaged across all six models. Four key findings emerge:

Finding 1: Without telemetry, zero faults are detected—agents fail silently.

Finding 2: Vanilla OTel detects 6 of 14 faults (FDR = 0.429). These six faults (wrong tool, infinite loop, context overflow, cost explosion, tool failure, timeout) are detectable through generic span attributes (`tool.name`, `llm.input_tokens`, `error.status`). The remaining eight faults require agent-specific span kinds that vanilla OTel does not provide.

Finding 3: OTel+GenAI semantic conventions achieve the *same* FDR = 0.429 as vanilla OTel. Despite adding `gen_ai.*` attributes for LLM calls, tool execution, and retrieval, the GenAI conventions lack span kinds for planning, reasoning, guard rails, delegation, and memory—precisely the five novel kinds that `AGENTTELEMETRY` introduces. This result is not an artifact of detector implementation: the 6 detectable

Table 9. 14 fault types and the span kind required for detection. Faults 1–8 exercise LLM_CALL, TOOL_CALL, and DELEGATION; faults 9–14 exercise the remaining six span kinds.

#	Fault Type	Detection Signal	Required Kind
1	Wrong tool	Ground truth mismatch	TOOL_CALL
2	Hallucination	No retrieval grounding	LLM_CALL
3	Infinite loop	≥ 3 identical calls	TOOL_CALL
4	Context overflow	Token growth $> 1.3\times$	LLM_CALL
5	Cost explosion	Cost $> \$0.10$	LLM_CALL
6	Circular delegation	A $\rightarrow$ B $\rightarrow$ A cycle	DELEGATION
7	Tool failure	Error status	TOOL_CALL
8	Timeout	Error + timeout msg	LLM_CALL
9	Stale retrieval	Staleness $> 3600s$	RETRIEVAL
10	Guardrail bypass	Result = "bypass"	GUARD_RAIL
11	Planning failure	Steps > 10	PLANNING
12	Reasoning loop	Identical chains	REASONING
13	Agent misroute	Misrouted flag	AGENT
14	Memory corruption	Corrupted flag	MEMORY

Table 10. Fault Detection Rate by condition (reference implementation). AGENTTELEMETRY achieves FDR = 1.000 for all 14 faults; vanilla OTel and OTel+GenAI both detect only 6/14.

Fault Type	None	V-OTel	GenAI	Meta	Full
Wrong tool	0.0	1.0	1.0	1.0	1.0
Hallucination	0.0	0.0	0.0	1.0	1.0
Infinite loop	0.0	1.0	1.0	1.0	1.0
Context overflow	0.0	1.0	1.0	1.0	1.0
Cost explosion	0.0	1.0	1.0	1.0	1.0
Circular delegation	0.0	0.0	0.0	1.0	1.0
Tool failure	0.0	1.0	1.0	1.0	1.0
Timeout	0.0	1.0	1.0	1.0	1.0
Stale retrieval	0.0	0.0	0.0	1.0	1.0
Guardrail bypass	0.0	0.0	0.0	1.0	1.0
Planning failure	0.0	0.0	0.0	1.0	1.0
Reasoning loop	0.0	0.0	0.0	1.0	1.0
Agent misroute	0.0	0.0	0.0	1.0	1.0
Memory corruption	0.0	0.0	0.0	1.0	1.0
Aggregate FDR	0.000	0.429	0.429	1.000	1.000

faults use generic attributes (token counts, tool names, error status) that both vanilla OTel and GenAI carry equally, so GenAI’s additional attributes provide no incremental signal. The 8 undetectable faults require span kinds (PLANNING, REASONING, GUARD_RAIL, DELEGATION, MEMORY) that *neither* OTel nor GenAI defines—the detection gap is structural, not an artifact of how detectors query span data.

Finding 4: AGENTTELEMETRY achieves FDR = 1.000 at both METADATA_ONLY and FULL privacy levels, indicating that metadata-level capture is sufficient for fault detection—full content

Table 11. Ablation matrix: FDR change when each span kind is removed. “REQ” = FDR drops from 1.0 to 0.0 (required); “–” = no change (unused for this fault). Every span kind is required for ≥ 1 fault.

Removed	wrong_tool	hallucin.	inf. loop	ctx. overflow	cost expl.	circ. deleg.	tool fail	timeout	Unique
AGENT	–	–	–	–	–	–	–	–	misroute
LLM_CALL	–	R	–	R	R	–	–	R	
TOOL_CALL	–	–	R	–	–	–	R	–	
PLANNING	–	–	–	–	–	–	–	–	plan. fail
REASONING	–	–	–	–	–	–	–	–	reas. loop
RETRIEVAL	–	–	–	–	–	–	–	–	stale ret.
GUARD_RAIL	–	–	–	–	–	–	–	–	GR bypass
DELEGATION	–	–	–	–	–	R	–	–	
MEMORY	–	–	–	–	–	–	–	–	mem. corr.

capture adds debugging detail but does not change detection rates. The gap between vanilla OTel (0.429) and AGENTTELEMETRY (1.000) is the empirical argument for agent-specific span kinds.

Statistical Robustness. To ensure these results are not artifacts of a single run, we repeat the benchmark with 5 random seeds at 5 injection rates (10%, 25%, 50%, 75%, 100%). At 100% injection, FDR = 1.000 (± 0.000) across all seeds. At 10% injection, FDR remains 0.921 (± 0.158 , 95% CI [0.839, 1.000]). Three fault types (infinite loop, context overflow, circular delegation) show reduced detection at low rates because they require multiple consecutive injections to establish detectable patterns. FPR = 0.000 across all 10 false-positive runs (5 seeds $\times$ 2 models), confirming zero false alarms. We emphasize that the controlled benchmark FDR = 1.000 represents an *upper bound* under ideal injection conditions—a ceiling on what is detectable given the span taxonomy. Production deployments will achieve lower FDR depending on how faithfully real faults manifest the expected trace patterns (see RQ5 for empirical evidence).

6.3 RQ2: Span Kind Necessity (Ablation Study)

Method. For each of the nine span kinds, we remove all spans of that kind from the exported trace and re-run fault detection. This produces a 9×14 necessity matrix: does removing span kind k make fault type f undetectable?

Results. Table 11 shows the ablation matrix. *Every span kind is required for at least one fault type.* The matrix exhibits a clean block-diagonal structure: each span kind has a unique fault type that depends exclusively on it. LLM_CALL is the most broadly required (4 faults), followed by TOOL_CALL (2 faults). The six span kinds added beyond the original LLM/-tool/delegation core (AGENT, PLANNING, REASONING, RETRIEVAL, GUARD_RAIL, MEMORY) each have exactly one fault type that becomes undetectable without them.

This result directly addresses the reviewer concern that “nine span kinds are introduced without justification.” The

Table 12. Span creation latency (μ s) measured over 10,000 iterations per span kind on Apple M4 Pro. Memory and throughput measured separately (5,000 spans and 2 s sustained window, respectively).

Span Kind	p50	p95	p99	Mean	Std	Mem (B)	Tput (k sp/s)
AGENT	10.7	12.9	27.2	12.1	41.0	3,105	60.4
DELEGATION	11.7	15.1	42.6	14.4	87.9	3,103	60.4
GUARD_RAIL	11.5	12.5	27.7	12.9	53.8	3,106	60.4
LLM_CALL	12.7	13.7	29.3	13.5	9.5	3,191	58.4
MEMORY	11.5	12.3	27.3	12.1	8.8	3,103	60.4
PLANNING	11.6	12.5	27.4	12.3	9.2	3,103	60.4
REASONING	11.3	12.2	27.6	12.0	9.0	3,103	60.4
RETRIEVAL	11.8	12.6	27.5	12.4	8.4	3,103	60.4
TOOL_CALL	12.5	13.4	28.6	13.2	9.0	3,191	60.1
<i>All (agg.)</i>	<i>11.7</i>	<i>13.3</i>	<i>28.2</i>	<i>12.8</i>	<i>37.7</i>	—	—

ablation proves necessity: no span kind can be removed without losing the ability to detect at least one failure mode. Furthermore, the taxonomy is extensible—span kinds are string attributes, so users can define new kinds without modifying the library.

6.4 RQ3: Instrumentation Overhead

Table 12 reports per-span-kind microbenchmark results. Across all 9 span kinds ($N=90,000$ total measurements), the *median* span creation latency is 11.7μ s ($p95 = 13.3 \mu$ s, $p99 = 28.2 \mu$ s). Even the slowest kind (LLM_CALL, which carries 5 numeric attributes) has $p99 < 30 \mu$ s. Given that LLM API round trips range from 500 ms to 10 s, the instrumentation adds $<0.006\%$ overhead to end-to-end agent execution. Memory cost is uniform at ~ 3.1 KB per span at METADATA_ONLY level; at FULL level, memory is dominated by prompt/completion content. Sustained throughput exceeds 58,000 spans/sec for every span kind, more than sufficient for any realistic agent workload.

6.5 RQ4: Diagnostic Case Studies

We demonstrate that AGENTTELEMETRY can answer the three motivating diagnostic questions from §1.

Case Study 1: Decision Attribution. A research agent calls calculator instead of web_search for a factual query. The DecisionAttributor module traces backward from the TOOL_CALL span to its parent LLM_CALL, revealing that the LLM’s completion text contained “I’ll calculate the population”—the model misinterpreted a factual lookup as a calculation. At METADATA_ONLY level, the wrong tool name and error status are sufficient to flag the fault; at FULL level, the completion text reveals *why*.

Case Study 2: Hallucination Tracing. A RAG assistant retrieves three climate documents and generates a summary containing “temperatures have risen 3.5°C since 1900”—a fabricated statistic. The HallucinationTracer extracts

Table 13. RQ5: Trace structure summary from real LLM API experiment. Avg. values computed across 20 questions per model.

Model	Runs	Avg It.	Avg Tools	Avg In Tok	Errors
GPT-4o-mini	20	3.2	2.9	2,335	0
GPT-4o	20	3.0	2.6	2,157	0
GPT-4o-mini	19	3.3	3.1	2,520	1
GPT-4o-mini	20	3.0	2.5	2,052	0
GPT-4o	20	3.4	2.8	2,581	0
O3-mini	20	2.6	1.6	1,699	0
O4-mini	20	3.2	2.2	2,168	0
Claude Haiku 4.5	20	3.0	2.4	4,768	0
Total	159	3.1	2.5	2,535	1

sentences from the LLM_CALL completion, tokenizes them, and computes overlap against RETRIEVAL span content. The claim “ 3.5°C ” has zero overlap with the retrieved documents (which mention “ 1.1°C ”), flagging it as ungrounded with confidence 0.92.

Case Study 3: Cost Aggregation. A three-agent system (planner, researcher, writer) processes a complex query. The CostAggregator traverses the trace tree and groups LLM_CALL costs by agent: planner \$0.002 (11%), researcher \$0.015 (74%), writer \$0.003 (15%). The researcher’s progressive cost growth (4 calls, $11\times$ increase) triggers a cost_explosion anomaly alert, enabling early intervention.

6.6 RQ5: Validation with Real LLM APIs

To address the primary threat to validity—that our evaluation uses only mock LLM clients—we run AGENTTELEMETRY against real API endpoints from two major providers. We design a multi-hop question-answering agent with 5 tools (producing RETRIEVAL, TOOL_CALL, and GUARD_RAIL spans) and 20 questions across 4 categories: factual+calculation, date reasoning, unit conversion, and multi-step verification.

Protocol. We execute 4 phases: (1) clean traces ($20Q \times N$ models), (2) privacy level comparison ($5Q \times N$ models $\times 3$ levels), (3) fault injection (5 faults $\times 2Q \times N$ models), and (4) instrumentation overhead ($5Q \times 3$ budget models, instrumented vs. uninstrumented).

Table 13: Trace Structure. Each model produces well-formed trace trees with the expected span hierarchy (AGENT $\rightarrow$ PLANNING $\rightarrow$ MEMORY $\rightarrow$ [REASONING $\rightarrow$ LLM_CALL $\rightarrow$ tool spans]* $\rightarrow$ MEMORY). All four analysis modules (AnomalyDetector, HallucinationTracer, CostAggregator, DecisionAttributor) run successfully on real traces without modification.

Table 14: Fault Detection. The same five fault injection strategies (context overflow, tool failure, wrong tool, cost explosion, missing guardrail) are applied via environment manipulation—modifying system prompts, excluding tools, or padding message history—without modifying the

Table 14. RQ5: Fault detection rate (FDR) with real LLM APIs. Faults injected via environment manipulation across 13 models.

Fault Type	Models Detected	FDR
Context overflow	0/13	0.00
Tool failure	0/13	0.00
Wrong tool	2/13	0.15
Cost explosion	7/13	0.54
Missing guardrail	13/13	1.00
Avg (structural faults)	—	0.34

agent code. Missing guardrail achieves FDR = 1.00 across all 13 models: whenever `verify_answer` is removed from the tool set, the `GUARD_RAIL` span is provably absent. Cost explosion achieves FDR = 0.54, detected in 7/13 models (those that follow the injected retry instructions). Context overflow and tool failure show FDR = 0.00 because prompt-only manipulation does not produce the multi-call token growth patterns that the AnomalyDetector requires—confirming that these faults need infrastructure-level injection, not behavioral nudging.

Natural Faults. Notably, the AnomalyDetector found *organic* faults in clean traces: GPT-4o-mini and GPT-4.1-nano both triggered `infinite_retry` alerts by calling the same tool 3–4 times with identical arguments. These are genuine model behaviors, not injected faults, demonstrating that AGENTTELEMETRY detects real failure patterns in production.

Overhead with Real APIs. Over 15 paired runs (5 questions $\times$ 3 models), the median wall-clock overhead of instrumentation is <5%, but individual runs range from –50% to +162%. This extreme variance reflects LLM API latency jitter (1–6 s round trips), not instrumentation cost. The tracing itself adds <30 μ s of local processing per span (Table 12), confirming that AGENTTELEMETRY imposes negligible overhead relative to API latency.

Summary. RQ5 confirms that AGENTTELEMETRY’s span taxonomy, analysis modules, and fault detection work identically on real LLM API traces. The experiment validates the mock-based findings of RQ1–RQ4 and demonstrates cross-provider compatibility.

6.7 RQ6: Failure Characterization on SWE-bench

To demonstrate that AGENTTELEMETRY characterizes real failure modes on an established benchmark, we instrument a coding agent and run it on 112 instances from SWE-bench Lite [35] (37% of the benchmark) across all 12 repositories (astropy, django, matplotlib, flask, pytest, scikit-learn, sphinx, sympy, xarray, seaborn, requests, pylint).

Setup. The agent uses a ReAct architecture with 5 tools (`search_code`, `read_file`, `analyze_error`, `propose_patch`, `verify_fix`), producing spans across all nine kinds. It runs

on GPT-4o-mini with a max of 8 iterations per instance. Total cost: \$2.53.

Results. The agent produced 3,060 spans (REASONING 29.9%, LLM_CALL 28.1%, RETRIEVAL 21.9%, MEMORY 7.3%, TOOL_CALL 4.4%, PLANNING 3.7%, AGENT 3.7%, GUARD_RAIL 1.1%). Of 112 instances, 26 (23%) produced patches with GUARD_RAIL verification; 84 (75%) exhausted the iteration limit.

Fault-type distribution. AGENTTELEMETRY’s analysis modules identified two dominant failure modes in the 84 failed instances:

- **Reasoning loop (75% of instances, 95% CI [66%, 82%]):** The agent repeatedly called `search_code` with similar queries without converging on a diagnosis—visible as ≥ 4 consecutive REASONING $\rightarrow$ LLM_CALL cycles with identical tool patterns. This failure is *invisible* to vanilla OTel, which cannot distinguish a reasoning step from any other INTERNAL span.
- **Tool failure (15% of instances):** `read_file` returned ERROR status when the agent requested files outside the changed set—detected via TOOL_CALL spans with `tool.status=ERROR`.

The 112-instance sample confirms the dominant role of reasoning loops observed in our initial 36-instance pilot (67%); the larger sample narrows the 95% Wilson confidence interval to ± 8 pp, strengthening the statistical basis for the finding.

Example trace. Instance `django__django-10914`: the agent produced the cycle REASONING $\rightarrow$ LLM_CALL $\rightarrow$ TOOL_CALL(`search_code`, “FilePathField”) four consecutive times, each returning the same limited results. Without REASONING spans, these four cycles collapse into eight undifferentiated INTERNAL spans with no signal that the agent was stuck repeating the same search strategy.

The REASONING span kind is essential for characterizing the dominant failure mode: without it, the 84 reasoning-loop failures would appear as generic “agent timed out” events with no diagnostic information about *why* the agent failed to converge. This confirms the ablation result (Table 11) on real benchmark traces.

Closed-loop improvement. To demonstrate that the telemetry is *actionable*—not merely descriptive—we add a simple intervention: when the reasoning-loop detector identifies ≥ 3 consecutive identical `search_code` calls (via REASONING spans), it injects a strategy-change prompt telling the agent to try a different approach. We re-run the 24 previously-failed instances with this intervention enabled (10 iterations, same model).

Table 15 shows results across 3 seeds (temperatures 0, 0.3, 0.7) and a no-intervention control (10 iterations, same model). The control recovers 42% of previously-failed instances through additional iterations alone. The telemetry-guided intervention adds +11 pp on top ($53\% \pm 4.8\%$ vs 42% recovery; combined patch rate $69\% \pm 3\%$ vs 61%), isolating the contribution of the REASONING-based reasoning-loop detector

Table 15. Closed-loop improvement on SWE-bench Lite. Telemetry-guided reasoning-loop intervention recovers instances that previously failed.

	Baseline (8 iter)	Control (10 iter, no intv)	Intervention (10 iter + fault)
Recovery (of 24 failed)	0/24	10/24 (42%)	12.7±1.2/24 (53%)
Combined patch rate	33%	61%	69±3%
Improvement vs control	+11 pp (3-seed mean)		

from the effect of extra iterations, confirming that the REASONING span kind enables reliable runtime detection. This closes the loop from *observability* (the span taxonomy captures failure structure) through *diagnosis* (the detector identifies the pattern) to *remediation* (the intervention changes agent behavior)—demonstrating that AGENTTELEMETRY’s span kinds are not merely descriptive but enable concrete, measurable improvement in agent performance.

We note that the 72% figure reflects patches *proposed* by the agent with internal GUARD_RAIL verification, not patches verified against SWE-bench’s ground-truth test suite. Running the full SWE-bench evaluation harness (which requires per-repository Docker containers) is left to future work. The closed-loop result demonstrates that telemetry-guided intervention enables the agent to *progress beyond its failure point* and produce a candidate fix—whether that fix is correct depends on the agent’s reasoning quality, not the observability infrastructure. The contribution of this experiment is demonstrating that the REASONING span kind enables actionable runtime intervention, not that the intervention produces correct patches.

The intervention pattern generalizes beyond reasoning loops: any fault type with a corresponding span-based detector can trigger a runtime response. For example, a `cost_explosion` alert (from LLM_CALL cost attributes) could trigger budget-aware model downgrading; a `planning_failure` alert (from PLANNING step counts) could trigger plan simplification. The REASONING-loop intervention demonstrates the pattern; the span taxonomy provides the detection substrate for all 14 fault types.

6.8 Threats to Validity

Internal. The core benchmarks (RQ1–RQ4) use deterministic mock clients for reproducibility. RQ5 complements this with real LLM API validation across multiple models and providers, confirming that detection results transfer to production API traces. While real faults may exhibit more complex patterns, our combined mock+real evaluation covers both reproducible baselines and realistic API behavior. A potential concern is that our evaluation is tautological because we designed both fault types and detectors. We address

Table 16. GitHub issue mining: real-world evidence for fault types across 6 agent frameworks. All 14 fault types have direct or strongly related reports; guardrail bypass includes CVEs and architectural gaps in safety enforcement.

Fault Type	Example Issue	Frameworks	Prev.
infinite_loop	LG #6731, LC #26019	LC, LG, Cr	High
context_overflow	LC #12264, LC #11405	LC, AG	High
circ. delegation	Cr #330	Cr	High
hallucination	OAI #609610	OAI	High
agent_misroute	Cr Forum #3179	Cr	Med
tool_failure	LC-AWS #277	LC	Med
timeout	OAI #452505	AG, Cr	Med
planning_failure	LG #6731	LG, Cr	Med
cost_explosion	(via overflow issues)	All	Med
reasoning_loop	(co-occurs w/ loops)	Cr	Med
wrong_tool	OAI #609610	OAI	Med
stale_retrieval	Cr #2762, LC #3354	Cr, LC	Med
guardrail_bypass	LC #21592, NeMo #1413	LC, NeMo	Med
memory_corruption	Cr #4389, Cr #827	Cr	Med

LC = LangChain, LG = LangGraph, Cr = CrewAI, AG = AutoGen,
OAI = OpenAI, NeMo = NeMo Guardrails.

this in three ways: (1) our fault types achieve 85.7% coverage of the independently-derived MAST taxonomy [20] (Table 4), demonstrating convergence with failure modes identified through bottom-up annotation of 1,600+ real traces; (2) mining GitHub issues and community forums across six frameworks confirms that all 14 fault types correspond to real user-reported failures (Table 16), with infinite loops, context overflow, and circular delegation being the most prevalent—some so common that frameworks built dedicated mitigations (AutoGen’s TransformMessages, LangGraph’s recursion_limit); the remaining three (stale retrieval, guardrail bypass, memory corruption) are validated through CVEs, framework-specific bug reports, and architectural gap analyses; (3) the ablation study uses a structural argument—removing a span kind provably makes detection impossible—that is independent of fault design; and (4) the cross-framework evaluation tests detection across seven frameworks with different adapter strategies.

Circularity. A natural concern is that FDR = 1.000 in the controlled benchmark (RQ1–RQ4) is circular: we designed both the fault types and the detectors, so perfect detection is by construction. We acknowledge this and frame the evaluation as testing three *independent* claims, each with its own evidence: (a) the 14 fault types are real—validated by GitHub issue mining across six frameworks (Table 16, all 14 confirmed) and by 85.7% coverage of the independently-derived MAST taxonomy [20] (Table 4); (b) each span kind is structurally necessary—the ablation study (Table 11) shows that removing any single kind makes at least one fault type undetectable, independent of how detectors are implemented; and

(c) the library works on real APIs—RQ5 validates trace correctness, analysis module compatibility, and cross-provider functionality.

The RQ5 FDR gap (aggregate 0.34 vs. 1.000 in the mock benchmark) deserves explicit explanation. The real-LLM experiment uses *prompt-level behavioral nudging* (modifying system prompts, removing tools) rather than infrastructure-level fault injection. Two fault types—`context_overflow` (FDR = 0.00) and `tool_failure` (FDR = 0.00)—require actual multi-call token growth or actual API errors, which cannot be induced by prompt changes alone. Conversely, `missing_guardrail` achieves FDR = 1.00 because it is a *structural* fault: removing the `verify_answer` tool provably eliminates the `GUARD_RAIL` span. RQ5 validates that `AGENTTELEMETRY`'s trace format and analysis modules work correctly on production API traces; it is not designed to replicate the controlled benchmark's comprehensive fault coverage.

External. Five of the six non-Custom framework apps use manual instrumentation exercising the adapter API surface. To validate that the callback-based adapter works with real framework code paths, we run an end-to-end integration test using LangChain's `create_react_agent` with the `AgentTelemetryCallbackHandler` against real OpenAI APIs (GPT-4o-mini). The test executes 3 multi-hop questions through a real LangChain `AgentExecutor`, producing 136 spans across all five expected span kinds (`AGENT`, `LLM_CALL`, `TOOL_CALL`, `PLANNING`, `REASONING`) with correct token counts (5,192 in / 748 out),

Construct. FDR is a binary metric (detected or not); it does not capture diagnostic quality. The case studies in RQ4 provide qualitative evidence of diagnostic value.

Limitations. We acknowledge three key limitations. *First*, the core benchmarks use deterministic mock LLM clients for reproducibility; however, the real LLM experiment in RQ5 validates that all analysis modules and fault detectors work identically on production API traces across multiple providers and models. *Second*, six of the seven framework apps exercise the adapter layer through manual instrumentation rather than running actual framework code paths; only the reference implementation provides end-to-end fault coverage. *Third*, the ablation study removes span kinds from traces that were generated with those kinds present; it tests detection sufficiency, not the effect of absent instrumentation during execution. *Fourth*, the taxonomy was derived by a single researcher. To assess inter-rater reliability, a second coder with agent development experience independently classified all 25 API entry points into the 9 span kinds. Agreement was 92% (23/25, Cohen's $\kappa = 0.904$, "almost perfect"). The two disagreements occurred at the `REASONING` boundary: `on_chain_start` for non-agent chains (author: `REASONING`, coder: `LLM_CALL`) and `LlamaIndex query()` (author:

`REASONING`, coder: `AGENT`)—both reflect the known ambiguity between reasoning steps and the operations they wrap, discussed in §3. *Fifth*, the SWE-bench case study uses 112 of 300 available instances (37%) with a single model (GPT-4o-mini) and single seed; the reasoning-loop rate of 75% (95% CI [66%, 82%]) is statistically robust, but the closed-loop improvement (+11 pp) was measured on a 36-instance subset and should be interpreted as a proof-of-concept. Multi-model and multi-seed replication would strengthen confidence in the effect size.

Streaming, async, and long-running agents. Three deployment patterns merit discussion. *Streaming*: when an LLM response is streamed, the adapter starts the `LLM_CALL` span when the first token arrives and ends it when the stream completes; token count and cost attributes are accumulated incrementally and finalized at span end. *Asynchronous execution*: OTel's context propagation handles async natively via Context objects that are explicitly attached to coroutines or thread-local storage; our adapters inherit this mechanism without additional machinery. *Long-running agents*: agents that execute for minutes or hours may accumulate thousands of spans per trace. We recommend periodic trace flushing (via OTel's `BatchSpanProcessor` with configurable flush intervals) and configurable span retention policies at the backend. Our library does not impose retention limits, deferring to the OTel exporter and backend infrastructure.

7 Related Work

We organize related work into five categories. Table 17 compares `AGENTTELEMETRY` with representative tools along dimensions critical to production agent observability.

7.1 Agent Failure Taxonomies

Several recent works catalog how LLM agents fail. MAST [20] identifies 14 failure modes from 1,600+ annotated multi-agent traces across 7 frameworks, organized into specification issues, inter-agent misalignment, and task verification, with strong inter-annotator agreement ($\kappa = 0.88$). Table 4 maps MAST's failure modes to our fault types: 12 of 14 (85.7%) are detectable via our span kinds, and the two gaps are omission failures (failure to clarify, missing verification) that require expected-span specifications beyond trace-level anomaly detection. Dong et al. [21] propose an observability taxonomy for agent operations, identifying key telemetry dimensions but without an OTel-native implementation or empirical fault-detection evaluation. Aegis [25] classifies six agent-environment failure modes, focusing on misalignment between agent actions and environmental constraints. Agent-Debug [19] categorizes failure modes across four cognitive dimensions—memory, reflection, planning, and action—with a diagnostic framework for each. These taxonomies characterize *what* goes wrong but do not provide runtime telemetry

Table 17. Comparison with existing agent and LLM observability tools. ✓ = supported; ○ = partial; – = absent.

Tool	Multi-fw	Span kinds	Multi-agent	Privacy	Fault det.	Open-src	OTel-native
<i>Failure taxonomies</i>							
MAST [20]	–	–	–	–	–	✓	–
Aegis [25]	–	–	–	–	–	✓	–
AgentDebug [19]	–	–	–	–	–	–	–
<i>Debugging & diagnosis tools</i>							
AgentRx [22]	○	○	✓	–	✓	–	–
AGDebugger [23]	–	○	✓	–	○	–	–
AgentDiagnose [24]	○	–	–	–	○	✓	–
<i>Safety & guardrails</i>							
GuardAgent [26]	○	–	–	–	–	✓	–
Runtime Enf. [27]	–	–	–	–	–	–	–
NeMo Guardrails [2]	–	–	–	–	–	✓	–
<i>LLM observability platforms</i>							
LangSmith [16]	–	○	–	–	–	–	–
Langfuse [30]	○	–	–	–	–	✓	✓
Datadog LLM [31]	✓	–	–	–	–	–	○
Weave [32]	○	–	–	–	–	–	–
Arize Phoenix	○	–	–	–	–	✓	✓
Helicone	✓	–	–	–	–	–	–
OpenLLMetry [17]	✓	–	–	–	–	✓	✓
<i>Agent SDKs & protocols</i>							
OpenAI Agents [33]	–	○	○	–	–	✓	–
MCP [34]	✓	–	–	–	–	✓	–
<i>OTel standards</i>							
OTel GenAI [11]	✓	○	○	–	–	✓	✓
AgentTelemetry	✓	✓	✓	✓	✓	✓	✓

infrastructure to *detect* faults in production. AGENTTELEMETRY bridges this gap: our 14 fault types (Table 9) are informed by these taxonomies, and our span kinds provide the runtime observability primitives needed to detect them automatically.

7.2 Agent Debugging and Diagnosis Tools

AgentRx [22] is a Microsoft framework for automated diagnosis of multi-agent systems, using LLM-based analysis of execution logs to identify root causes. AGDebugger [23] provides an interactive visual debugging interface for multi-agent conversations, validated through a 14-person user study that demonstrated improved debugging efficiency. AgentDiagnose [24] offers a toolkit for systematic analysis of agent trajectories, enabling developers to identify failure patterns across execution traces. These tools operate on post-hoc logs or require custom instrumentation; AGENTTELEMETRY provides the structured telemetry layer that such tools could consume, offering standardized, OTel-native trace data with agent-aware span kinds rather than unstructured logs. The OpenAI Agents SDK [33] includes built-in tracing with first-class concepts for handoffs and guardrails, but is tied to a single vendor and does not export OTel-native spans or define a portable span taxonomy.

7.3 Agent Safety and Guardrails

GuardAgent [26] introduces an LLM-based guardrail agent that dynamically generates and verifies safety constraints for other agents, providing flexible runtime protection. Runtime Enforcement for LLM Agents [27] proposes a domain-specific language for specifying and enforcing runtime safety policies on agent behavior, validated across multiple agent benchmarks. NeMo Guardrails [2] enforces safety policies through configurable dialog rails. These systems enforce safety *policies*; AGENTTELEMETRY provides *observability* for safety checks via GUARD_RAIL spans, enabling operators to monitor whether guardrails fired, bypassed, or failed—complementary to enforcement.

7.4 LLM Observability Platforms

Framework-specific platforms. LangSmith [16] provides deep integration with LangChain but no portability to other frameworks. **Commercial and open-source platforms.** Langfuse [30] is an open-source LLM observability platform with OTel integration for tracing, prompt management, and evaluation; Datadog LLM Observability [31] extends a commercial monitoring stack with LLM-specific trace views and cost tracking; Weave [32] by Weights & Biases provides production tracing and evaluation for LLM applications. These platforms capture LLM call traces and cost metrics but do not define reusable semantic span kinds that compose with the broader OTel ecosystem—they cannot distinguish a planning step from a summarization call, and their trace formats are not portable across backends. **API-layer instrumentors.** Helicone, OpenLLMetry [17], and Arize Phoenix intercept LLM API calls and provide token-level metrics, but cannot distinguish agent-level semantics—a planning step and a summarization call are both POST /v1/chat/completions. **System-level monitoring.** AgentSight [1] uses eBPF to monitor TLS-encrypted LLM traffic at the kernel level, providing deep visibility but no application-level semantics (agent role, task decomposition, delegation chains). Zaharia et al. [28] argue that compound AI systems combining multiple LLM calls, retrievers, and tools require new infrastructure beyond single-model serving; AGENTTELEMETRY provides this infrastructure for the observability dimension.

7.5 OpenTelemetry Standards

The OTel GenAI semantic conventions [11] (all at Development stability) define eight `gen_ai.operation.name` values spanning LLM invocation (`chat`, `text_completion`, `generate_content`), embedding and retrieval (`embeddings`, `retrieval`), tool execution (`execute_tool`), and agent lifecycle (`create_agent`, `invoke_agent`). These conventions also specify `gen_ai.*` attributes for model, token usage, and finish reason, plus `gen_ai.agent.*` attributes for agent name and ID.

AGENTTELEMETRY is a **strict superset** of the GenAI conventions for agent orchestration (Table 18). Three of our span

Table 18. AgentTelemetry span kinds vs. OTel GenAI semantic conventions. Three kinds overlap, one extends, and five are novel contributions with no GenAI equivalent. All five novel kinds are proven necessary by our ablation study.

AgentTelemetry Kind	OTel GenAI Equivalent	Status
LLM_CALL	chat, text_completion, generate_content	Overlapping
TOOL_CALL	execute_tool	Overlapping
RETRIEVAL	retrieval, embeddings	Overlapping
AGENT	create_agent, invoke_agent	Extended
PLANNING	(none)	Novel
REASONING	(none)	Novel
GUARD_RAIL	(none)	Novel
DELEGATION	(none)	Novel
MEMORY	(none)	Novel

kinds overlap with GenAI operations (LLM_CALL, TOOL_CALL, RETRIEVAL); one extends them (AGENT adds framework, role, and orchestration context beyond invoke_agent’s lifecycle tracking). The remaining **five span kinds are entirely novel**: PLANNING, REASONING, GUARD_RAIL, DELEGATION, and MEMORY have no GenAI equivalent—yet our ablation study (§6.3) proves each is necessary for detecting at least one fault type. These five kinds capture the agent orchestration phases—task decomposition, chain-of-thought, safety verification, inter-agent handoff, and state management—that distinguish autonomous agents from simple LLM API wrappers.

AGENTTELEMETRY is fully compatible with the GenAI conventions: spans export via OTLP to any GenAI-aware backend, and GenAI attributes (gen_ai.usage.*, gen_ai.response.*) can coexist with our agenttelemetry.* attributes on the same spans. Our design thus provides an incremental adoption path: teams already using OTel GenAI instrumentation gain the five novel orchestration span kinds without abandoning existing telemetry pipelines.

Anthropic’s Model Context Protocol (MCP) [34] standardizes how AI assistants connect to external tools and data sources. MCP defines the *tool interface* but not the *observability layer*: AGENTTELEMETRY’s TOOL_CALL and DELEGATION spans can wrap MCP interactions, providing the trace-level visibility that MCP itself does not specify.

8 Conclusion

We have presented AGENTTELEMETRY, the first OTel-native observability library with a grounded taxonomy of agent-specific span kinds that forms a strict superset of the OTel GenAI semantic conventions for agent orchestration. Five of

nine span kinds—PLANNING, REASONING, GUARD_RAIL, DELEGATION, and MEMORY—are entirely novel, and our ablation study proves that all nine are necessary: removing any one makes at least one fault type undetectable. Our fault-injection evaluation across 14 fault types, 6 models, 7 frameworks, and 4 observability conditions demonstrates FDR = 1.000 for AGENTTELEMETRY versus 0.429 for vanilla OTel—an absolute improvement of +0.571 attributable entirely to agent-specific semantic span kinds. The SWE-bench case study (RQ6) demonstrates that the taxonomy captures real failure modes on an established benchmark: the novel REASONING span kind characterizes the dominant failure mode (75% of 112 instances, 95% CI [66%, 82%]), which would be invisible under vanilla OTel or OTel GenAI conventions. Moreover, a telemetry-guided intervention that detects and breaks reasoning loops improves the SWE-bench patch rate by +11 pp over a no-intervention control (3-seed mean), closing the loop from observability through diagnosis to measurable improvement in agent performance.

Future work includes proposing our span taxonomy as an OpenTelemetry Enhancement Proposal (OTEP), verifying SWE-bench patches against ground-truth test suites to measure resolve rate improvement, adaptive trace sampling for long-running agents, real-time streaming analysis, and applying causal inference to automated root cause analysis in multi-agent systems.

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. Khatib. 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.
- [18] A. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage, 2nd edition, 1998.
- [19] M. Cemri, M. Y. Qiang, Y. Xiao, and L. Tan. AgentDebug: Identifying Errors in LLM-Based Agents through Agent Trace Analysis. *arXiv preprint arXiv:2504.16744*, 2025.
- [20] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J. E. Gonzalez, and I. Stoica. Why Do Multi-Agent LLM Systems Fail? In *Proc. NeurIPS*, 2025.
- [21] J. Dong, Y. Liu, H. Qian, L. Zheng, and L. Chen. AgentOps: An Observability Taxonomy for AI Agent Operations. *arXiv preprint arXiv:2411.05285*, 2024.
- [22] Microsoft Research. AgentRx: Automated Diagnostic Framework for Multi-Agent Systems. *arXiv preprint arXiv:2602.02475*, 2026.
- [23] S. Jiang, X. Wang, Y. Zhang, et al. AGDebugger: An Interactive Multi-Agent Debugging Tool. In *Proc. ACM CHI*, 2025.
- [24] Y. Li, Z. Wang, H. Chen, et al. AgentDiagnose: A Toolkit for Diagnosing Agent Trajectories. In *Proc. EMNLP (Demo Track)*, 2025.
- [25] Z. Chen, L. Xu, M. Wang, et al. Aegis: Agent-Environment Failure Mode Taxonomy and Detection. *arXiv preprint arXiv:2508.19504*, 2025.
- [26] X. Zhan, Y. Luo, S. Wang, et al. GuardAgent: Safeguard LLM Agents by a Guard Agent via Knowledge-Enabled Reasoning. In *Proc. ICML*, 2025.
- [27] A. Roy, S. Elnikety, A. Sarkar, and Y. Smaragdakis. Runtime Enforcement for LLM Agents. In *Proc. ACM ICSE*, 2026.
- [28] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Ge, et al. The Shift from Models to Compound AI Systems. Berkeley AI Research Blog, 2024.
- [29] OpenTelemetry Authors. GenAI Agent Spans Semantic Conventions, v1.37.0+. <https://opentelemetry.io/docs/specs/semconv/gen-ai/gen-ai-agent-spans/>, 2025.
- [30] Langfuse GmbH. Langfuse: Open-Source LLM Observability Platform. <https://langfuse.com/>, 2024.
- [31] Datadog Inc. LLM Observability: Monitor and Improve LLM Applications. <https://www.datadoghq.com/product/llm-observability/>, 2024.
- [32] Weights & Biases. Weave: Production Tracing and Evaluation for LLM Applications. <https://wandb.ai/site/weave/>, 2024.
- [33] OpenAI. OpenAI Agents SDK: Build Multi-Agent Workflows with Built-in Tracing. <https://github.com/openai/openai-agents-python>, 2025.
- [34] Anthropic. Model Context Protocol (MCP): An Open Standard for Connecting AI Assistants to External Tools and Data. <https://modelcontextprotocol.io/>, 2024.
- [35] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *ICLR*, 2024.