AgentTrace: A Structured Logging Framework for Agent System Observability

Adam AlSayyad*, **Kelvin Yuxiang Huang***, **Richik Pal***

University of California, Berkeley
alsayyad@berkeley.edu, yxkelvinhuang@berkeley.edu, richik.pal@berkeley.edu

Abstract

Despite the growing capabilities of autonomous agents powered by large language models (LLMs), their adoption in high-stakes domains remains limited. A key barrier is security: the inherently nondeterministic behavior of LLM agents defies static auditing approaches that have historically underpinned software assurance. Existing security methods, such as proxy-level input filtering and model glassboxing, fail to provide sufficient transparency or traceability into agent reasoning, state changes, or environmental interactions. In this work, we introduce AgentTrace, a dynamic observability and telemetry framework designed to fill this gap. AgentTrace instruments agents at runtime with minimal overhead, capturing a rich stream of structured logs across three surfaces: operational, cognitive, and contextual. Unlike traditional logging systems, AgentTrace emphasizes continuous, introspectable trace capture, designed not just for debugging or benchmarking, but as a foundational layer for agent security, accountability, and real-time monitoring. Our research highlights how AgentTrace can enable more reliable agent deployment, fine-grained risk analysis, and informed trust calibration, thereby addressing critical concerns that have so far limited the use of LLM agents in sensitive environments.

Introduction

The deployment of autonomous agents built on large language models (LLMs) has shown early promise across a wide spectrum of domains, including software engineering, scientific analysis, and complex decision-making workflows (Schick et al. 2023; Weng 2023). However, despite the functional competence of these systems in isolated tasks, their adoption in safety-critical or high-integrity environments remains severely limited. A primary constraint is the absence of structured and dynamic observability frameworks that can account for the stochastic reasoning behaviors of LLM agents and enable reliable diagnosis, security assessment, and governance.

Conventional methods for securing AI systems, such as static input filtering, prompt hardening, and API boundary control, are insufficient for LLM-based agents that act through long-running, multi-step reasoning cycles in open-ended environments. These agents dynamically compose tool invocations, retrieve external knowledge, and revise goals during execution, producing behaviors that are difficult to trace or explain post hoc. Current security and auditing tools remain constrained by assumptions of determinism, procedural transparency, or bounded action space, assumptions that do not hold in LLM-based settings. As a result, the dominant paradigm has centered on proxy-level defenses (e.g., PromptArmor) and glassbox introspection of static prompts and outputs (Zou et al. 2023). These approaches offer limited insight into agent intent, decision provenance, or operational context, especially in the presence of tool-use chaining, memory operations, or multi-agent collaboration.

Crucially, the undeterministic nature of agentic reasoning introduces a novel challenge for security and governance: threats and failures can emerge not merely from malicious inputs or faulty tools, but from the emergent behavior of the agent's cognitive trajectory. As observed in recent red-teaming and adversarial prompting research (Liu et al. 2024; Zou et al. 2023), even well-scoped agents may deviate from expectations when their reasoning states are not explicitly monitored. To address this, a shift is required from static, perimeter-oriented security architectures toward dynamic, semantic observability of the agent's internal execution process.

Our contributions are as follows. We present **AgentTrace**, a structured, schema-based logging framework that instruments LLM agents at runtime without requiring code modifications. We introduce a three-surface taxonomy: cognitive, operational, and contextual, that enables multi-level introspection into an agent's reasoning, execution, and environment. Finally, we demonstrate how AgentTrace integrates with existing telemetry infrastructures such as OpenTelemetry to provide scalable, real-time observability. Through this design, AgentTrace establishes a foundation for transparent, accountable, and reproducible agentic systems, paving the way for future research in alignment, evaluation, and safety.

Related Work

Agent Observability and Tracing Frameworks. Recent agent-oriented observability tools instrument execution flows to support debugging and monitoring. *AgentOps* introduces a hierarchical span taxonomy that organizes reasoning, planning, workflow, task, tool, and LLM spans to

*Equal contribution.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

trace artifacts and processes throughout the agent lifecycle (Dong, Lu, and Zhu 2024). *LADYBUG* complements this with post-hoc debugging that combines execution tracing and LLM-aided self-reflection (Rorseth et al. 2025). However, these systems primarily target *single-surface* traces and lack a schema that unifies cognitive artifacts with operational and contextual signals. In contrast, AgentTrace introduces a *schema-based*, *multi-surface* observability model linking *operational*, *cognitive*, and *contextual* traces under a unified envelope, realized at runtime via lightweight instrumentation.

System-Level Telemetry and Distributed Tracing. System-centric approaches provide horizontal visibility into API calls and service dependencies, often via kernel/OS boundary tracing and OpenTelemetry-style pipelines. *AgentSight*, for example, correlates LLM prompts with kernel events using eBPF to bridge intent and execution at system boundaries, and has been applied to boundary tracing and anomaly detection (Zheng et al. 2025). Yet these methods are largely semantics-agnostic to agent intent and internal reasoning, offering limited *causal* linkage between what the agent infers and what the system executes. **AgentTrace** complements them by *embedding cognitive semantics into the telemetry stream*: cognitive spans are *nested* within operational and contextual spans and exported through standard backends, preserving interoperability while enabling reasoning-aware, end-to-end traces.

Cognitive Interpretability and Textual Traceability. Work on cognitive observability and agentic interpretability models reasoning traces and human–agent alignment. *Watson* surfaces implicit reasoning errors in LLM-powered agents without altering agent architecture (Rombaut et al. 2024), while concurrent work frames explanation as interactive mental-model building with LLM-driven proactive clarification (Kim et al. 2025). These efforts enhance understanding of internal reasoning but remain decoupled from runtime observability and structured, composable telemetry. **AgentTrace** bridges this gap by treating cognition as a first-class telemetry surface: reasoning steps, plans, and reflections are captured in a machine-readable schema and *causally linked* to operational actions and contextual I/O at runtime, enabling unified, schema-consistent analysis across surfaces."

Methodology

Schema

We present a principled, schema-based methodology for capturing rich, interpretable traces of autonomous LLM-agent behavior. At the core of our logging framework is a formalized representation of logs as transformations of runtime events into structured records:

$$L(S{:}E{:}C) \rightarrow R,$$

where S denotes the surface (cognitive, operational, or contextual), E is the event content, C represents metadata context, and R is a structured record that satisfies four critical properties: consistency (schema-compliant representation), causality (temporal fidelity), fidelity (faithful to the agent's internal and external behavior), and interoperability (analysis-ready, framework-agnostic). This schema design builds on recent efforts in AI observability frameworks (Goyal et al. 2024) and structured introspection mechanisms for LLM agents (Rombaut et al. 2024), extending them with a formal schema for high-fidelity, surface-level trace capture. However, our schema uniquely emphasizes semantically enriched introspection in LLM agents, encompassing not just control flow and system I/O, but also the agent's cognitive deliberations and interactions with external APIs and data stores.

Surface Taxonomy and Extraction Procedure

We operationalize the schema across three disjoint but composable surfaces of agent execution, each instrumented through techniques designed for transparent, non-intrusive logging.

Operational Surface: Method-Level Execution Tracing

The operational surface captures all explicit agent method calls, argument structures, return values, and execution timing. Through Python introspection and function wrapping, we automatically intercept all public methods on the agent class. Each method invocation produces a pair of events - start and complete - enriched with span-level metadata such as argument count, result type, and execution duration. Events are written to both structured JSONL logs and OpenTelemetry spans, preserving trace and span relationships for end-to-end visibility.

This approach is functionally equivalent to distributed tracing as used in service-oriented architectures (Sigelman et al. 2010), but adapted for fine-grained agent-level observability. All logs conform to a fixed schema to enable downstream consumption, and trace linkage ensures coherent propagation across multiple layers of abstraction.

Cognitive Surface: LLM Interaction Introspection

The cognitive surface is designed to capture the internal deliberations of the agent's reasoning engine, primarily interactions with LLMs. These include raw prompts, completions, extracted reasoning chains (e.g., Chain-of-Thought), and confidence estimates. When supported by the LLM API (e.g., OpenAI or Anthropic), this surface also parses semi-structured outputs to extract `<thinking>` segments, step-by-step reasoning, and structured JSON fields such as plan or reflection.

Span metadata is derived from instrumented LLM API calls, and cognitive spans are nested within operational traces to maintain full trace context.

Extraction relies on a set of generalizable strategies: marker-based pattern detection, XML tag parsing, and JSON field extraction. This design supports multiple reasoning formats and enables comparative analysis across different model outputs and prompt templates.

Contextual Surface: External System I/O

The contextual surface tracks all outbound agent interactions with external systems, including HTTP APIs,

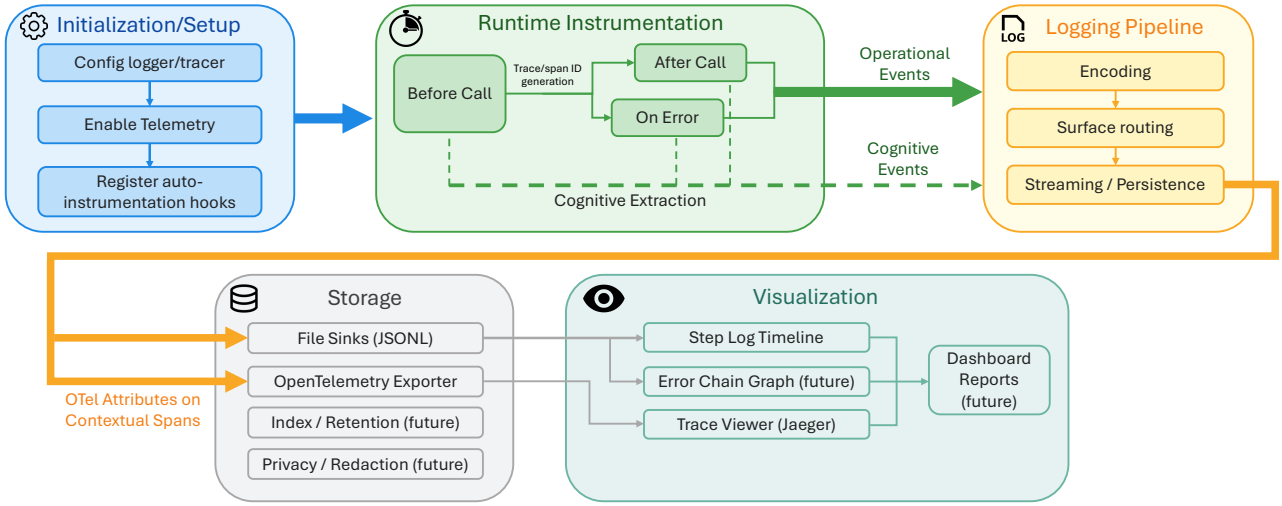

Figure 1: **AgentTrace system flow.** End-to-end runtime architecture showing initialization (logger setup, OpenTelemetry enablement, and auto-instrumentation hooks), runtime instrumentation with trace/span ID generation and cognitive extraction, logging pipeline for event encoding and routing, and downstream storage and visualization. Contextual spans are enriched with OpenTelemetry attributes via auto-instrumentation.

SQL/NoSQL databases, cache layers, vector stores, and file systems. Rather than requiring agent authors to manually log these operations, we leverage OpenTelemetry's auto-instrumentation capabilities to monkey-patch standard libraries (e.g., requests, sqlalchemy, redis) at runtime. However, we also include the option for manual instrumentation in order to capture more granular and cohesive log structures.

Each contextual interaction produces a span enriched with resource-specific metadata: URLs and headers for HTTP, query structure and row counts for SQL, and key/value operations for cache or vector DBs. These events are stored exclusively as OTel spans to avoid redundancy with file-based logging. Temporal nesting under the same trace context enables causal analysis across layers of computation and I/O.

This surface bridges the agent's cognitive operations and their environmental grounding, providing a unified view of how internal plans translate to external effects (cf. Paxton et al. 2023).

Unified Representation and Trace Semantics

All three surfaces emit logs that conform to a shared envelope schema. Each log event includes:

- a UUID identifier
- surface type (cognitive, operational, or contextual)
- trace ID and span ID
- precise UTC timestamp
- event body (structured per surface)

Logs are stored in two complementary formats:

- JSONL files (line-delimited JSON for offline inspection, streaming, or replay)
- OpenTelemetry spans (for real-time distributed tracing and integration with tools like Jaeger or Tempo)

This dual-path storage ensures both low-latency local debugging and scalable remote observability. Logs are append-only and schema-validated at write time to preserve consistency and support batch analytics and visualization.

Implementation

To operationalize the proposed schema, we implemented AgentTrace as a modular runtime system that instantiates the three observability surfaces introduced in the Methodology section. This implementation translates the theoretical framework into concrete runtime mechanisms for capturing, structuring, and exporting agent logs.

Overview. AgentTrace is a lightweight Python package that (i) injects runtime instrumentation without modifying agent code, (ii) emits schema-consistent records across *operational*, *cognitive*, and *contextual* surfaces, and (iii) exports telemetry to an OpenTelemetry (OTel) backend for distributed tracing. Our design goals are non-intrusiveness, low overhead, and graceful degradation (i.e., falling back to local logging when remote export fails) when external telemetry is unavailable.

Python Module Layout and Initialization

AgentTrace exposes a minimal API:

- `init(...)` configures local sinks and toggles OTel export or auto-instrumentation.
- `instrument_agent(obj, name, methods=None)` wraps selected public callables for runtime tracing.
- `ALogger` records surface-specific events and optionally exports to OTel.

Initialization loads configuration, prepares append-only JSONL outputs, and, when enabled, activates OTel auto-instrumentation for common I/O libraries.

Algorithm 1: AgentTrace Runtime Instrumentation Wrapper

Input: agent instance A, name n, optional allowlist $\mathcal{M}$
Output: instrumented agent A

for each public method $m \in \texttt{select}(A, \mathcal{M})$ **do**
  let $f \leftarrow$ original implementation of m
  define wrapper w with preserved signature
  function $w(\mathbf{x})$
    $(\texttt{trace_id}, \texttt{span_id}) \leftarrow$ new IDs (or propagate)
    RECORDOPERATIONAL($\texttt{status=start}, n, m, \texttt{args=summary}(\mathbf{x})$)
    $t_0 \leftarrow$ now()
    try
      $y \leftarrow f(\mathbf{x})$
      $(y, \theta) \leftarrow \texttt{maybe_extract_cognitive}(y)$
      if $\theta \neq \varnothing$ **then** RECORDCOGNITIVE(θ)
      RECORDOPERATIONAL($\texttt{dur=}$now()$-t_0$, $\texttt{status=complete}, \texttt{result=summary}(y)$)
      return y
    catch exception e
      RECORDOPERATIONAL($\texttt{status=error}$, $\texttt{dur=}$now()$-t_0$, $\texttt{err=repr}(e)$)
      rethrow e
  end function
  replace m on A with w
end for
return A

Instrumentation via Decorator Injection

AgentTrace uses a runtime observer pattern. For each target method, it installs an in-place wrapper that:

1. emits a *start* event (method name, argument summary, timestamp),
2. records a *complete* event on success (duration, result summary), and
3. records an *error* event on exception while re-raising to preserve semantics.

Wrappers preserve function signatures, and each event carries a fresh `span_id` under a shared `trace_id`. Span nesting and context propagation enforce temporal causality across reasoning, tool, and workflow events.

Cognitive trace extraction. When completions include a delimited reasoning segment, AgentTrace returns the cleaned answer and logs the segment as a *cognitive* event; otherwise, results pass through unmodified.

Log Schema and Local Sinks

All surfaces share a common envelope with identifiers, timestamps, agent name, `surface` $\in \{\text{operational}, \text{cognitive}, \text{contextual}\}$, level, `trace_id`, and `span_id`. Surface payloads are concise: **Operational** includes method, status, duration, and result summary (optionally token or latency metadata); **Cognitive** stores thought, plan, and reflection excerpts with model and token counts; **Contextual** captures operation type, source, query or response summaries, and provenance. Crucially, contextual traces manage tool invocations and data access operations (reads and writes), linking agent reasoning with its external interactions. Operational and cognitive events are persisted to JSONL, while contextual events are primarily exported via OTel with optional file mirroring for offline workflows. All records are validated at emission time against the schema to maintain consistency.

OpenTelemetry Export and Auto-Instrumentation

When enabled, AgentTrace converts each event into an OTel span and exports via a batch processor. Attributes are populated defensively, i.e., with type checks and safe conversions: scalars are set directly, structured values are JSON-stringified when necessary, and unknown objects fall back to string representations. Exporter failures gracefully degrade to local JSONL.

Contextual I/O capture. OTel auto-instrumentation patches common HTTP, database, and cache libraries to emit contextual spans (URLs, queries, status, counts, latencies) without manual logging. In tracing UIs (e.g., Jaeger), method-level operational spans appear alongside contextual spans, providing end-to-end visibility complementary to local files.

Engineering Considerations

Non-intrusiveness. Decorators enable tracing without changing agent logic.
Low overhead. Typical success paths emit two events per call; export is batched asynchronously.
Robustness. Serialization and export are defensive; failures never block execution.
Composability. A stable, analysis-ready schema supports JSONL and OTel; contextual I/O is auto-instrumented with optional file mirroring.

Conclusion

In this paper, we present AgentTrace, a research framework that establishes the first open standard for structured agent logging through a schema-based protocol spanning cognitive, operational, and contextual traces. By transforming logging into a semantically rich, introspectable substrate, AgentTrace elevates observability from an engineering utility to a core enabler of agent safety, reproducibility, and accountability. This design closes critical gaps in existing observability systems by enabling fine-grained debugging, reliable failure attribution, and transparent governance of LLM-based agents.

Looking ahead, the structured and interpretable traces generated by AgentTrace pave the way for security and evaluation research. They allow for dynamic threat modeling, real-time risk detection, and post-hoc forensic analysis of adversarial or misaligned behaviors. Beyond security, these logs provide the groundwork for agent evaluation, enabling new metrics for reasoning stability, goal fidelity, and cross-agent behavioral benchmarking.

References

Dong, L.; Lu, Q.; and Zhu, L. 2024. AgentOps: Enabling Observability of LLM Agents. arXiv:2411.05285.

Goyal, S.; Hira, M.; Mishra, S.; Goyal, S.; Goel, A.; Dadu, N.; DB, K.; Mehta, S.; and Madaan, N. 2024. LLMGuard: Guarding Against Unsafe LLM Behavior. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI-24)*, 23790–23792. Demonstration Track.

Kim, B.; Hewitt, J.; Nanda, N.; Fiedel, N.; and Tafjord, O. 2025. Because we have LLMs, we Can and Should Pursue Agentic Interpretability. arXiv:2506.12152.

Liu, Y.; Lo, S.; Lu, Q.; Zhu, L.; Zhao, D.; Xu, X.; Harrer, S.; and Whittle, J. 2024. Agent Design Pattern Catalogue: A Collection of Architectural Patterns for Foundation Model Based Agents. *arXiv preprint arXiv:2405.10467*.

Rombaut, B.; Masoumzadeh, S.; Vasilevski, K.; Lin, D.; and Hassan, A. E. 2024. Watson: A Cognitive Observability Framework for the Reasoning of LLM-Powered Agents. arXiv:2411.03455.

Rorseth, J.; Godfrey, P.; Golab, L.; Srivastava, D.; and Szlichta, J. 2025. LADYBUG: an LLM Agent DeBUGger for data-driven applications. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*, 1082–1085. OpenProceedings.org. Demonstration Paper.

Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*.

Sigelman, B. H.; Barroso, L. A.; Burrows, M.; Stephenson, P.; Plakal, M.; Beaver, D.; Jaspan, S.; and Shanbhag, C. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. https://research.google.com/archive/papers/dapper-2010-1.pdf. Google Technical Report.

Weng, L. 2023. LLM Powered Autonomous Agents. Online blog post, https://lilianweng.github.io/posts/2023-06-23-agent/.

Zheng, Y.; Hu, Y.; Yu, T.; and Quinn, A. 2025. AgentSight: System-Level Observability for AI Agents Using eBPF. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*, SOSP '25, 110–115. ACM.

Zou, A.; Wang, Z.; Carlini, N.; Nasr, M.; Kolter, J. Z.; and Fredrikson, M. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. *CoRR*, abs/2307.15043. ArXiv:2307.15043.