Unified Agentic Interfaces is All You Need for AI Agent Observability

Yanpeng Hu
huyp@shanghaitech.edu.cn
ShanghaiTech University
China

Yusheng Zheng
yzhen165@ucsc.edu
UC Santa Cruz
USA

Abstract

Production AI agent systems expose a fundamental observability mismatch: traditional monitoring assumes deterministic code with stable interfaces, while agents generate non-deterministic outputs through rapidly-evolving logic. Current solutions (APM tools, LLM-centric monitoring, and framework-specific SDKs) create fragmented silos that cannot capture semantic reasoning, resist tampering, or scale across multi-vendor ecosystems. We argue that *agent observability fundamentally requires autonomous intelligence: LLM-powered observability agents are necessary to interpret semantic failures, adapt to evolving agent behaviors, and perform cross-layer causal reasoning at production scale–capabilities that humans and rule-based systems cannot provide.* This paper presents a vision for unified agentic interfaces through a two-plane architecture: a Data Plane capturing telemetry at stable system boundaries (syscalls, network, inference endpoints, human feedback), decoupling observability from agent internals; and a Cognitive Plane deploying autonomous observability agents for semantic understanding at production scale. We identify open research challenges across system-level capture, semantic reconstruction, and standardization required to realize this vision.

1 The Agentic Observability Challenge

1.1 The Transformation: From Deterministic Code to Autonomous Reasoning

AI-powered agentic systems are fundamentally changing how we build software infrastructure [16, 37]. Frameworks like AutoGen [39], LangChain [8], Claude Code [6], and gemini-cli [28] orchestrate large language models (LLMs) to autonomously execute complex workflows, including debugging production incidents, analyzing multi-modal data pipelines, coordinating distributed deployments, and making real-time operational decisions [35].

Consider a concrete example: an automated code review agent receives a pull request, analyzes the diff against project style guides, queries a vector database for similar past bugs, spawns subprocess tools to run linters and tests, coordinates with a security agent to check for vulnerabilities, and finally posts structured feedback. This workflow involves multiple LLM calls, external tool invocations, inter-agent communication, and persistent state, all orchestrated autonomously with minimal human intervention.

Yet despite rapid adoption in development environments, production deployment at scale faces three fundamental challenges that expose a critical gap in existing observability paradigms:

Semantic Failures Replace Deterministic Errors. In our code review example, the agent might hallucinate a security vulnerability that doesn't exist, enter an infinite loop requesting more context, or forget critical style guidelines mid-review. Unlike traditional crashes (segfaults, exceptions), these failures are *semantic*, meaning they are plausible but incorrect outputs that require understanding agent *intent* to detect. As practitioners observe, building multi-agent systems without observability "feels like debugging a black box" where developers are "essentially flying blind" without visibility into decisions and data flows [25, 29]. More critically, prompt injection attacks [40] can compromise agents to evade their own logging, hiding malicious behavior. Traditional metrics (CPU, latency, 5xx errors) cannot capture these failure modes.

Opaque Multi-Layer Costs. The code review workflow incurs costs at every layer, including token usage for LLM calls, vector database queries, API calls to GitHub, and subprocess execution for linters. When agents spawn recursive sub-tasks or enter reasoning loops, costs can spiral unpredictably. Without unified visibility across this multi-layer stack, runaway expenses remain invisible until discovered in post-incident analysis.

Fragmented Multi-Vendor Instrumentation. Our code review agent's execution spans multiple administrative domains, including LLM serving (OpenAI API), agent orchestration (LangChain), vector storage (Pinecone), tool execution (subprocess calls), and inter-agent communication (custom APIs). Each layer has its own SDK, logging format, and instrumentation requirements, creating incompatible telemetry silos. Multi-agent coordination exacerbates costs: production deployments report up to 15ÃŮ higher token consumption compared to single-agent workflows [5], while error propagation across agent handoffs creates cascading failures invisible to per-agent monitoring. When the security agent coordination fails, debugging requires correlating logs across five different systems with no unified trace.

1.2 Why Existing Observability Paradigms Cannot Scale

These challenges reveal a fundamental mismatch between agent systems and existing observability approaches. Table 1 summarizes how agent observability differs qualitatively from traditional software monitoring. Three existing paradigms address parts of this problem, but none provides a complete solution:

Traditional APM Is Operationally Blind to Semantics. Classic application performance monitoring (APM) tools like Datadog and New Relic excel at detecting infrastructure failures such as crashes, 5xx errors, memory leaks, and latency spikes. But when our code review agent hallucinates a non-existent vulnerability, no error is thrown. CPU and memory remain normal. The only signal is *semantic incorrectness*, which requires understanding natural language intent and reasoning quality, capabilities APM systems were never designed to provide.

LLM-Centric Monitoring Stops at the Model Boundary. Existing LLM monitoring solutions (prompt safety filters, hallucination detectors) focus on single-turn model interactions. They monitor token generation quality at the inference endpoint. But our code review workflow involves multi-step reasoning, tool orchestration (spawning linters), cross-agent coordination (calling the security

Table 1. Traditional vs. Agentic Observability: A Comparative Framework

Aspect	Traditional Observability	Agentic Observability
Primary Goal	System health & performance	Behavioral correctness, safety, & trust
Core Pillars	Metrics, Events, Logs, Traces (MELT) [22]	MELT + **Evaluations** + **Governance**
Nature of Failures	Crashes, exceptions, latency spikes	"Quiet failures" (hallucinations, flawed logic, misuse of tools)
System Behavior	Deterministic & predictable	Non-deterministic & emergent
Key Question	"Is the system working?"	"Is the system thinking correctly and acting appropriately?"
Core Unit of Analysis	Service/request trace	Agent decision path/trajectory graph

agent), and persistent state (vector database lookups). LLM monitoring cannot observe subprocess execution, inter-agent messages, or the causal chain connecting user intent to final output.

LLM Serving Observability Optimizes Infrastructure, Not Behavior. LLM serving platforms monitor throughput, latency percentiles, GPU utilization, and SLO compliance, all infrastructure metrics for the inference layer. These say nothing about whether the agent followed instructions correctly, used tools appropriately, or achieved its goal within cost constraints. Serving observability ensures the model runs efficiently; agent observability ensures the agent behaves correctly.

1.3 Two Fundamental Gaps

The mismatch between agent systems and existing observability paradigms creates two critical technical challenges:

The Instrumentation Gap: Agent Code Is Unstable. Returning to our code review agent, suppose it initially uses `subprocess.run(["pylint"])` but later evolves to dynamically generate custom linter scripts. Application-level instrumentation (callbacks, middleware) that wraps the original `subprocess.run` call becomes obsolete. Worse, if the agent is compromised via prompt injection [40], it can modify its own logging code to hide malicious behavior. For example, it could write a bash script with exploit commands (not logged as harmful file I/O) and then execute it (appears as a normal tool call). In-process instrumentation cannot provide tamper-resistant audit trails.

The Semantic Gap: System Events Lack Intent. Conversely, observing only syscalls and network traffic shows *what* happened (process spawned, bytes sent) but not *why*. When our code review agent spawns `pylint`, syscall tracing records `execve("pylint", [...])`. But *why* did the agent run it? What reasoning led to this decision? Traditional observability frameworks [24, 33] lack semantic primitives such as attributes like `agent.goal`, `reasoning.step_id`, `tool.justification`, or anomaly detectors for semantic failures (contradictions, persona drift, instruction forgetting).

These gaps are complementary: application instrumentation provides semantics but is fragile and tamperable; system-boundary tracing is stable and tamper-resistant but semantically opaque. A complete solution must bridge both.

2 Current Solutions: A Fragmented Landscape

Having established the unique challenges of agent observability, we now survey existing solutions. Our analysis reveals a maturing ecosystem converging toward OpenTelemetry standards [9, 23], yet fundamentally limited by reliance on application-layer instrumentation that cannot address the instrumentation and semantic gaps.

2.1 Methodology: Ecosystem Survey

We surveyed agentic observability tooling as of early 2025, examining both industrial deployments and academic research. Our analysis covers two areas: (1) **Industrial tools**, which are production-ready solutions providing SDKs, proxies, or specifications for agent framework integration (Table 2), and (2) **Academic research**, which includes foundational work on agent monitoring, interpretability, and evaluation that informs current tooling design.

2.2 Key Findings: The Limits of Current Approaches

Industrial Tools. Analysis of 18 production systems (Table 2) reveals critical limitations: all tools require application-level instrumentation, creating maintenance burden and tampering vulnerabilities; despite OpenTelemetry adoption by five tools [23], agent-specific attributes remain unstandardized; while tools like TruLens and Arize Phoenix provide LLM-powered evaluation, none explain why decisions were made through reasoning trace reconstruction; and no tool observes kernel syscalls, TLS payloads, or subprocess execution directly.

Academic Research. Four research threads address complementary aspects: agent monitoring frameworks [12, 32] focus on trajectory logging but assume instrumented runtimes; mechanistic interpretability [19] reveals model internal reasoning through attention analysis; evaluation methodologies [27] develop semantic correctness metrics but operate on logged outputs rather than system-boundary telemetry; and system-level observability [41] using eBPF to capture kernel events and TLS payloads, though this approach only observes post-hoc without real-time intervention, lacks standardized schemas for multi-agent coordination, and cannot observe ML inference stack internals.

These findings reflect a deeper problem: existing approaches inherit design assumptions from two incompatible paradigms, neither suited for production agentic systems.

2.3 Redefining Agent Observability for Production Systems

Current approaches define observability narrowly, missing critical production requirements. Academic definitions focus on individual agent internal consistency (beliefs, intentions, actions) but ignore

#	Tool / SDK (year)	Integration path	What it provides	License / model	Notes
1	**LangSmith** [20] (2023)	Add `import langsmith` to LangChain/LangGraph apps	Request/response traces, prompt & token stats, evaluations	SaaS, free tier	Tight LangChain integration; OTel export beta
2	**Helicone** [17] (2023)	Reverse-proxy or Python/JS SDK	Logs OpenAI-style calls, cost/latency dashboards	OSS (MIT) + hosted	Proxy model requires no code changes
3	**Traceloop** [34] (2024)	One-line SDK import âĘŠ OTel	OTel spans for prompts, tools, sub-calls	SaaS, free tier	Standard OTel data compatibility
4	**Arize Phoenix** [3] (2024)	`pip install`, OpenInference tracer	Local UI + vector store for traces, automatic evals	Apache-2.0	Includes open-source UI for debugging
5	**Langfuse** [21] (2024)	SDK or OTel OTLP	Nested traces, cost metrics, prompt management	OSS (MIT) + cloud	Popular for RAG/multi-agent projects
6	**WhyLabs LangKit** [38] (2023)	Text metrics wrapper	Drift, toxicity, sentiment, PII detection	Apache-2.0 core	Focuses on text-quality metrics
7	**PromptLayer** [31] (2022)	Decorator or proxy	Prompt chain timeline, diff & replay	SaaS	Early solution, minimal code changes
8	**Literal AI** [4] (2024)	Python SDK + UI	RAG-aware traces, eval experiments	OSS + SaaS	Targets chatbot product teams
9	**W&B Weave/Traces** [7] (2024)	`import weave` or SDK	Links to W&B projects, captures code/IO	SaaS	Integrates with existing W&B workflows
10	**Honeycomb GenAI** [18] (2024)	Send OTel spans	Heat-maps on prompt spans, latency	SaaS	Built on mature trace store
11	**OTel GenAI Conv.** [9] (2024)	Spec + Python lib	Standard span names for models/agents	Apache-2.0	Provides semantic conventions
12	**OpenInference** [2] (2023)	Tracer wrapper	JSON schema for traces	Apache-2.0	Specification (not hosted service)
13	**AgentOps** [1] (2024)	Proxy injection into LLM calls	Time-travel debugging, multi-framework support (CrewAI, AutoGen)	OSS	Session replay across agent frameworks
14	**TruLens** [36] (2024)	Wrapper (`TruLlama`) or SDK	Multi-turn session tracking, custom feedback functions	OSS + hosted	Evaluation-focused with feedback loops
15	**Phospho** [30] (2024)	Log ingestion API	Clustering/labeling of LLM outputs, anomaly detection	OSS	Post-hoc NLP analytics on collected data
16	**MLflow** [11] (2024)	`mlflow.autolog()` for LLMs	Experiment tracking, artifact logging (prompts/outputs)	Apache-2.0	General MLOps extended to generative AI
17	**Maxim AI** [26] (2024)	One-line SDK integration	Agent trajectory visualization, cost/latency analytics	SaaS	Polished dashboard for production monitoring
18	**Guardrails.AI** [15] (2023)	Input/output validators	Real-time safety checks (toxicity, PII), auto-retry on violations	OSS (Apache-2.0)	Observability through safety enforcement

system-level concerns like multi-layer costs and multi-agent coordination. Industrial tools provide model-centric input/output analysis, capturing inference-level telemetry but missing tool execution, inter-agent communication, and cross-layer causality. Production deployment demands system-level, multi-agent observability addressing cost transparency across all layers, tamper-resistant audit trails, multi-agent coordination visibility, and unified causal graphs linking intent to execution. This reveals why current solutions are inadequate: they optimize for single-agent, single-layer monitoring while production requires multi-agent, multi-layer, system-centric observability.

The Path Forward. Achieving production-grade agentic observability demands resolving two architectural tensions: (1) *Where to capture telemetry?* Application instrumentation is semantically rich but fragile and tamperable; system boundaries are stable and tamper-resistant but semantically opaque. (2) *Who analyzes telemetry?* Human operators understand intent but cannot scale to millions of events; rule-based systems scale but cannot interpret semantic failures. A complete solution must bridge both gaps simultaneously: capturing at stable interfaces while analyzing with autonomous intelligence. We now present a vision for such an architecture.

3 A Two-Plane Architecture for Agent Observability

Production agent deployment exhibits three characteristics making traditional observability infeasible: heterogeneity (execution spans multiple administrative domains including LLM providers, agent frameworks, runtimes, and third-party tools, each with incompatible SDKs, making application-level coordination untenable), dynamism (agents modify their own logic continuously through prompt evolution, dynamic tool synthesis, and runtime feedback, making static instrumentation obsolete within days while enabling self-modification to bypass logging), and scale (thousands of agents generate millions of semantically-rich events per hour, creating a cognitive gap between raw telemetry and actionable insight that exceeds human capacity). These characteristics create two inescapable requirements:

Requirement 1 (from heterogeneity and dynamism): Observability must decouple from application internals, capturing telemetry at stable system boundaries that remain invariant across vendor changes, framework updates, and agent self-modification. This necessitates a Data Plane providing zero-instrumentation capture at kernel, network, and TLS interfaces, creating a unified foundation independent of agent implementation details.

Requirement 2 (from scale and semantics): Understanding agent behavior at production scale requires autonomous intelligence, including systems that interpret natural language prompts, correlate multi-layer telemetry, infer causal relationships, and adapt to evolving agent behaviors. Only LLM-powered systems can bridge the semantic gap between raw syscalls and agent intent. This necessitates a Cognitive Plane where specialized observability agents monitor, diagnose, and remediate other agents.

Critically, these planes are interdependent. The Data Plane provides tamper-resistant, unified telemetry that the Cognitive Plane requires for trustworthy analysis. The Cognitive Plane provides semantic understanding that makes Data Plane events actionable. Neither can function effectively alone. They form an integrated architecture where system-boundary capture enables intelligent understanding, and intelligent understanding validates system-boundary events. We now detail each plane’s design.

3.1 The Data Plane: Unified, Zero-Instrumentation Telemetry Capture

The Data Plane addresses Requirement 1 by capturing telemetry at stable system boundaries that remain invariant despite heterogeneous frameworks, dynamic agent evolution, and multi-vendor fragmentation.

Building on the AgentOps taxonomy [12] that identifies key artifacts requiring observability (goals, plans, tool outputs), we extend this to system-level boundaries. The Data Plane captures telemetry at stable OS/hardware boundaries (syscalls, TLS protocols, GPU APIs) that evolve slowly under vendor guarantees, avoiding the brittleness of application code that changes continuously. Zero-instrumentation design eliminates SDK imports and code modifications, making observability independent of agent implementation.[1] Capture operates at kernel/hardware level where compromised agents cannot falsify telemetry, while programmable interfaces enable custom filters and correlation without system restarts. This design maintains low performance impact through zero-copy data paths while capturing semantically rich telemetry beyond traditional metrics: natural language prompts, reasoning traces, tool arguments, and causal relationships. The Data Plane organizes capture into four hierarchical layers:

Model Layer: Captures GPU/CPU metrics, framework hooks, and model internals through mechanistic interpretability [19] (attention patterns, layer activations), bridging infrastructure monitoring and semantic transparency.

Network Layer: Captures TLS-encrypted traffic between agents and external services using programmable interfaces like eBPF [42] to intercept prompts, reasoning traces, and API responses without proxies or SDK modifications.

System Layer: Captures kernel syscalls revealing tool execution, file access, and resource consumption through eBPF tracing [13, 14], providing tamper-resistant visibility into agent actions.

Human Layer: Captures human feedback, corrections, and interventions alongside agent responses, providing ground truth for evaluation and closing the observability loop.

Correlating events from heterogeneous sources into unified causal traces remains a key challenge. While AgentSight [41] demonstrates eBPF-based boundary tracing, system-level observability alone cannot observe model internals, enable real-time intervention, or attribute events in multi-agent scenarios. The Data Plane addresses these limitations through architectural unification across four complementary layers, bridging model-internal reasoning, semantic intent, system execution, and human oversight. This delivers framework neutrality, tamper resistance, cost transparency across layers, and multi-vendor compatibility. However, raw telemetry alone cannot explain why decisions were made or identify anomalous behavior, necessitating the Cognitive Plane.

3.2 The Cognitive Plane: Why Only Agents Can Observe Agents

The Data Plane provides comprehensive, tamper-resistant telemetry. But raw events such as `execve("pylint")`, TLS payload containing prompt text, or 5000 tokens consumed cannot answer critical questions: *Why* this decision? Is this behavior anomalous? How should we respond? Bridging telemetry to actionable insight requires intelligence. Building on Watson's concept of cognitive observability [32] and recent work on agentic interpretability [19], we argue that only autonomous, LLM-powered systems can provide this intelligence at production scale, for three fundamental reasons:

First, semantic failures require semantic understanding. Agent failures are not crashes but semantic incorrectness in natural language outputs. Detecting hallucinations or reasoning flaws requires understanding context-dependent correctness that rule-based systems cannot provide. Watson's cognitive observability [32] demonstrates how observability agents can retroactively infer reasoning traces, reconstructing why agents made specific decisions.

Second, dynamic evolution demands continuous learning. Agent behaviors evolve continuously as prompts change and new tools emerge. Static rulesets become obsolete within days. Observability agents learn from historical incidents, continuously updating their models of normalcy as production agents evolve, ensuring observability evolves as fast as the agents it monitors.

Third, multi-layer causal reasoning exceeds human capacity. Understanding failures requires correlating evidence across layers: a cost spike may trace to a reasoning loop caused by a database timeout from a network partition. Reconstructing such causal chains from millions of events demands hypothesis generation, cross-layer correlation, and counterfactual reasoning that humans cannot perform at scale but are natural for LLM-powered diagnoser agents.

The two-plane design enforces strict separation across three dimensions: trust boundaries (Data Plane operates at privileged system layers inaccessible to production agents, preventing compromised agents from falsifying observability data), technology stacks (Data Plane requires low-level systems programming while Cognitive Plane requires LLM orchestration), and evolution rates (Data Plane interfaces evolve slowly with OS stability while Cognitive Plane adapts rapidly to changing agent behaviors). The planes form an inseparable architecture where system-boundary capture provides trusted telemetry while autonomous intelligence transforms it into actionable insights.

4 Open Research Challenges

The Data Plane faces challenges in correlating heterogeneous telemetry (GPU metrics, TLS payloads, syscalls, human feedback) across boundaries, ensuring tamper resistance through authenticated streams, and balancing privacy compliance with observability needs.

The Cognitive Plane must bridge low-level telemetry to high-level intent through causal inference, adaptive anomaly detection as agents evolve, probabilistic root-cause analysis for non-deterministic failures, and hierarchical evaluation across session boundaries.

Integration and standardization challenges include extending OpenTelemetry's GenAI conventions [9, 10] with agent-specific semantics (goals, reasoning steps, multi-agent coordination), developing evaluation frameworks with ground-truth failure datasets, and securing cross-organization telemetry sharing.

[1] While frameworks like LlamaIndex offer "one-click" SDK integration (e.g., `set_global_handler("arize_phoenix")`), these still require modifying agent codebases and remain vulnerable to tampering. The Data Plane's zero-instrumentation approach eliminates this dependency entirely.

References

[1] AgentOps. 2024. AgentOps: Time-Travel Debugging for AI Agents. https://www.agentops.ai/. Open-source toolkit for agent observability with multi-framework support.

[2] Arize AI. 2023. OpenInference: OpenTelemetry Instrumentation for LLM Applications. https://github.com/Arize-ai/openinference

[3] Arize AI. 2024. Home - Phoenix - Arize AI. https://phoenix.arize.com/

[4] Literal AI. 2024. Literal AI - RAG LLM observability and evaluation platform. https://www.literalai.com/

[5] Anthropic. 2024. Building Effective Agents. https://www.anthropic.com/research/building-effective-agents. Anthropic research on multi-agent system patterns and costs.

[6] Anthropic. 2025. Introducing Claude Code. https://www.anthropic.com/news/claude-code. Agentic coding tool announcement, Anthropic blog.

[7] Weights & Biases. 2024. Enterprise-Level LLMOps: W&B Traces - Wandb. https://wandb.ai/site/traces/

[8] Harrison Chase. 2023. LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain

[9] OpenTelemetry Community. 2024. OpenTelemetry Semantic Conventions for GenAI. https://opentelemetry.io/docs/specs/semconv/gen-ai/. Stable semantic conventions for LLM observability.

[10] OpenTelemetry Community. 2024. Semantic Conventions for Generative AI Systems. https://opentelemetry.io/docs/specs/semconv/gen-ai/

[11] Databricks. 2024. MLflow LLM Tracking. https://mlflow.org/docs/latest/llms/index.html. MLflow extension for generative AI experiment tracking.

[12] Liming Dong, Qinghua Lu, and Liming Zhu. 2024. AgentOps: Enabling Observability of LLM Agents. *arXiv preprint arXiv:2411.05285* (2024).

[13] eBPF Community. 2023. eBPF Documentation. https://ebpf.io/

[14] Brendan Gregg. 2019. *BPF Performance Tools*. Addison-Wesley Professional.

[15] Guardrails AI. 2023. Guardrails AI: Input/Output Validation for LLMs. https://www.guardrailsai.com/. Open-source framework for real-time safety checks and observability.

[16] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. *arXiv preprint arXiv:2402.01680* (2024).

[17] Helicone. 2023. Helicone / LLM-Observability for Developers. https://www.helicone.ai/

[18] Honeycomb. 2024. Observability for AI & LLMs - Honeycomb. https://www.honeycomb.io/ai-llm-observability

[19] Been Kim, John Hewitt, Neel Nanda, Noah Fiedel, and Oyvind Tafjord. 2025. Because we have LLMs, we Can and Should Pursue Agentic Interpretability. *arXiv preprint arXiv:2506.12152* (2025).

[20] LangChain. 2023. Observability Quick Start - LangSmith - LangChain. https://docs.smith.langchain.com/observability

[21] Langfuse. 2024. Langfuse - LLM Observability & Application Tracing. https://langfuse.com/

[22] Bingchang Li, Xin Peng, Qilin Xiang, Jianwei Tian, Tingjian Wang, Chen Zhang, and Wenyun Zhao. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27, 25 (2022). doi:10.1007/s10664-021-10063-9 Industrial survey defining the three pillars of observability: metrics, logs, and traces.

[23] Guangya Liu and Sujay Solomon. [n. d.]. AI Agent Observability – Evolving Standards and Best Practices. https://opentelemetry.io/blog/2025/ai-agent-observability/. OpenTelemetry Blog, March 6, 2025.

[24] Charity Majors. 2017. Observability: A Manifesto. https://www.honeycomb.io/blog/observability-a-manifesto. Honeycomb blog post defining modern observability principles.

[25] Adnan Masood. 2024. Establishing Trust in AI Agents II: Observability in LLM Agent Systems. https://medium.com/@adnanmasood/establishing-trust-in-ai-agents-ii-observability-in-llm-agent-systems-fe890e887a08. Medium article on trust and observability in agent systems.

[26] Maxim AI. 2024. Maxim AI: LLM Observability Platform. https://www.getmaxim.ai/. SaaS platform for agent trajectory visualization and monitoring.

[27] Dany Moshkovich and Sergey Zeltyn. 2025. Taming Uncertainty via Automation: Observing, Analyzing, and Optimizing Agentic AI Systems. *arXiv preprint arXiv:2507.11277* (2025).

[28] Taylor Mullen and Ryan J. Salva. 2025. Gemini CLI: Your Open Source AI Agent. https://blog.google/technology/developers/introducing-gemini-cli-open-source-ai-agent/. Google Developers Blog, Jun 2025.

[29] Kirill Petropavlov. 2024. Observability in Multi-Agent LLM Systems: Telemetry Strategies for Clarity and Reliability. https://medium.com/@kpetropavlov/observability-in-multi-agent-llm-systems-telemetry-strategies-for-clarity-and-reliability-fafe9ca3780c. Medium article on multi-agent observability challenges.

[30] Phospho. 2024. Phospho: Text Analytics Platform for LLM Applications. https://phospho.ai/. Open-source platform for clustering and labeling LLM outputs.

[31] PromptLayer. 2022. Complete AI Observability Monitor and Trace your LLMs - PromptLayer. https://www.promptlayer.com/platform/observability

[32] Benjamin Rombaut, Sogol Masoumzadeh, Kirill Vasilevski, Dayi Lin, and Ahmed E. Hassan. 2025. Watson: A Cognitive Observability Framework for the Reasoning of LLM-Powered Agents. *arXiv preprint arXiv:2411.03455* (2025).

[33] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. In *Technical Report, Google Inc.* https://research.google/pubs/pub36356/ Google Technical Report dapper-2010-1. Foundational work on distributed tracing that inspired OpenTelemetry.

[34] Traceloop. 2024. Traceloop - LLM Reliability Platform. https://www.traceloop.com/

[35] Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. 2025. Multi-Agent Collaboration Mechanisms: A Survey of LLMs. *arXiv preprint arXiv:2501.06322* (2025).

[36] Truera. 2024. TruLens: Evaluation and Tracking for LLM Applications. https://www.trulens.org/. Open-source evaluation toolkit with feedback loops and session tracking.

[37] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[38] WhyLabs. 2023. LangKit: Open source tool for monitoring large language models. https://whylabs.ai/langkit

[39] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enable Next-Gen Large Language Model Applications. https://github.com/microsoft/autogen

[40] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics (ACL Findings)*. doi:10.48550/arXiv.2403.02691 arXiv:2403.02691.

[41] Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. 2025. AgentSight: System-Level Observability for AI Agents Using eBPF. *arXiv preprint arXiv:2508.02736* (2025). Introduces boundary tracing to bridge semantic gap between agent intent and system-level execution.

[42] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending Applications Safely and Efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 557–574.