State-of-the-Art Best Practices for Observability and Monitoring of LLM and Multi-Agent Systems

Foundational Pillars of LLM and Multi-Agent System Observability

The emergence of Large Language Models (LLMs) and Multi-Agent Systems (MAS) represents a fundamental paradigm shift in software architecture. These systems are not deterministic; they are probabilistic, autonomous, and capable of dynamic decision-making. This inherent non-determinism renders traditional software monitoring techniques, such as Application Performance Monitoring (APM), insufficient for providing the necessary visibility to build, debug, and maintain reliable AI applications in production.[1] A new discipline, termed "Agentic Observability," has emerged to address these unique challenges by providing deep, actionable visibility into the internal workings, decisions, and outcomes of AI agents throughout their lifecycle.[1]

The Paradigm Shift from APM to Agentic Observability

Traditional APM tools were engineered for a world of predictable, well-defined applications where a given input reliably produces a specific output. They excel at tracking infrastructure health, latency, and throughput.[2] However, AI agents break this model. Their behavior is emergent, influenced by complex prompts, vast context windows, and interactions with external tools, making their execution paths difficult to predict.[1] This complexity is magnified in multi-agent systems, which may require up to 26 times the monitoring resources compared to single-agent applications due to the need to trace not only individual agent actions but also their coordination, handoffs, and cascading dependencies.[1] An error in one agent can propagate silently, causing downstream agents to operate on faulty data, a failure mode traditional APM is ill-equipped to detect.[1]

The core inadequacy of traditional monitoring stems from the "semantic gap"—the chasm between an agent's high-level intent, often expressed in natural language, and its low-level system actions, such as API calls or file system operations.[4] An application-level monitor might observe a successful tool call, while a system-level monitor sees a process writing to a file. Neither can semantically correlate these events to understand that a benign intention has been manipulated into a malicious or erroneous action, rendering them effectively blind.[4]

Agentic Observability bridges this gap by merging the strengths of traditional APM with the nuanced requirements of model and LLM performance monitoring, which include insights into predictions, data quality, and explainability.[1] This new approach provides a hierarchical visibility model, allowing teams to conduct root cause analysis by drilling down from a high-level application view, through individual user sessions and agent interactions, down to the granular level of individual traces and spans.[1] This hierarchical structure is not merely a visualization enhancement; it is a fundamental requirement for diagnosing failures in non-deterministic, compositional systems where the "why" of a failure is as important as the "what."

Core Telemetry Signals: Traces, Metrics, and Logs in the AI Context

Agentic Observability builds upon the three foundational pillars of traditional observability—metrics, logs, and traces—but adapts them to capture the unique dimensions of AI systems.[2]

A **trace** represents the complete, end-to-end journey of a single request as it traverses the system.[8] Each discrete operational step within that journey—such as a database lookup, an LLM inference call, or a tool execution—is captured as a

span.[9] For an AI agent processing a user query, a complete trace might capture the initial user input, the agent's generated plan, all external tool calls, the full context sent to the LLM, the LLM's reasoning process, and the final generated response.[10] This detailed, connected view of events transforms an otherwise opaque "black box" agent into a transparent "glass box," which is essential for debugging, optimization, and building trust in the system.[11]

Metrics provide quantitative, numerical data about system performance and resource utilization.[6] While traditional metrics like CPU, memory, and network usage remain relevant, the most critical metrics for AI systems are those that measure cost, performance, and quality.[10] These are often conceptualized as an adaptation of the "Four Golden Signals" of Site

Reliability Engineering: Latency, Traffic, Errors, and Saturation.[7]

Logs offer a detailed, time-stamped, and textual narrative of discrete events within the system.[7] In the context of AI agents, logging must be comprehensive, capturing not only system-level events but also the semantic content of interactions. This includes user interaction logs, full LLM interaction logs (capturing the exact prompt and completion), tool execution logs, and, critically, agent decision-making logs. These decision logs provide an audit trail of how an agent arrived at a conclusion, which is indispensable for debugging complex reasoning paths, identifying biases, and ensuring responsible AI practices.[10]

Critical AI-Specific Metrics: Cost, Latency, Token Usage, Quality, and Drift

To effectively manage LLM-powered systems, teams must move beyond traditional operational metrics and focus on a new set of AI-specific key performance indicators (KPIs) that directly impact business outcomes and user experience.[8]

Token Usage and Cost: Large language models are typically billed on a per-token basis for both input (prompt) and output (completion).[10] Consequently, monitoring token usage is a direct and essential proxy for operational cost. A robust observability platform must capture

prompt_tokens, completion_tokens, and total_tokens for every LLM invocation.[13] This granular data allows teams to pinpoint the most token-intensive and expensive operations within a trace, enabling targeted optimization efforts such as prompt refinement, response caching, or the use of smaller, more efficient models for simpler tasks.[8]

Latency: Inference latency, the time an agent takes to generate a response, is a critical factor for user satisfaction.[7] High latency can lead to poor user engagement and abandonment.[10] Latency should be measured at two levels: the end-to-end duration of the entire trace and the duration of each individual span within the trace.[11] This dual-level analysis is crucial for identifying performance bottlenecks, which could be caused by a slow model, an inefficient database query, or a high-latency external tool call.[8]

Response Quality: This is a complex, multi-faceted metric that assesses the semantic correctness and utility of an agent's output. It seeks to answer questions like: Was the response accurate and factually correct? Was it relevant to the user's query? Was it helpful? Did it hallucinate or invent information?.[10] Measuring quality requires a combination of automated and human-centric approaches. This includes tracking explicit user feedback (e.g., thumbs-up/down ratings), analyzing implicit user signals (e.g., a user immediately rephrasing a question, indicating the initial response was unsatisfactory), and employing automated evaluation techniques, such as using another LLM to "judge" the quality of a response against predefined criteria.[2]

Model Drift: AI models are not static; their performance can degrade over time as the real-world data they encounter in production diverges from the data they were trained on. This phenomenon is known as model drift.[10] Monitoring for drift involves tracking statistical changes in response patterns, shifts in output quality metrics, or an increase in negative user feedback over time.[10] Early detection of drift is a key indicator of long-term system health and signals that a model may need to be retrained or fine-tuned with more recent data to maintain its accuracy and effectiveness.[10]

The emphasis on metrics like quality and drift signifies a profound evolution in what system monitoring entails. The focus is shifting from observing purely *operational* signals (e.g., "Is the CPU usage high?") to observing *semantic* signals (e.g., "Is the agent's answer factually correct?"). This requires the observability pipeline itself to become more intelligent, capable of understanding and evaluating the meaning and correctness of the data flowing through it.

State-of-the-Art Observability in Multi-Agent Frameworks

The leading multi-agent frameworks—LangGraph, AutoGen, and CrewAI—each embody a distinct architectural philosophy, which in turn dictates their approach to observability. Understanding these differences is crucial for selecting the right framework and implementing an effective monitoring strategy. The choice reflects a fundamental trade-off between the seamlessness of a deeply integrated ecosystem, the broad compatibility of an open standard, and the ultimate flexibility of a platform-agnostic design.

LangGraph: Leveraging LangSmith for Stateful Graph Tracing and Human-in-the-Loop Monitoring

As a core component of the LangChain ecosystem, LangGraph is designed for first-class, deeply integrated observability through its companion platform, LangSmith.[16] This pairing represents an "Integrated Ecosystem" approach, offering a powerful and seamless developer experience at the potential cost of being tightly coupled to a specific vendor's tools.

Pattern: Stateful Graph Instrumentation
LangGraph models agentic workflows as stateful graphs, where nodes represent units of work (e.g., an LLM call, a tool execution) and edges represent the transitions between them. The primary observability pattern is to instrument each node and edge, which allows LangSmith to generate a visual and logical map of the agent's execution path.18 This provides unparalleled clarity into the complex, often cyclical, reasoning processes of agents.
How: Enabling Tracing via Callbacks and Environment Variables
Integration is straightforward. Developers enable tracing by setting standard environment variables (LANGSMITH_TRACING_V2=true, LANGSMITH_API_KEY, LANGSMITH_PROJECT).19 For more granular control, tracing can be applied selectively using LangSmith's tracing_context context manager or by passing a CallbackHandler directly into the graph's invocation configuration.[19]

Example: Enabling LangSmith Tracing for a LangGraph Invocation

Python

```python
# examples/langgraph_langsmith_tracing.py
import os
from langgraph.graph import StateGraph
from langchain_core.messages import HumanMessage
from langchain_openai import ChatOpenAI
from langfuse.langchain import CallbackHandler as LangfuseCallbackHandler # Example for
another tool
from langchain_community.callbacks.manager import get_openai_callback # Example for cost
tracking

# NOTE: For LangSmith, tracing is often enabled via environment variables.
# os.environ = "true"
# os.environ = "YOUR_API_KEY"
# os.environ = "My-Agent-Project"

# For programmatic control, you can pass callbacks in the config.
# This example shows how a Langfuse handler would be passed; the principle is the same for
LangSmith.
langfuse_handler = LangfuseCallbackHandler()

# Define the state and graph (simplified)
class AgentState(dict):
    messages: list

graph_builder = StateGraph(AgentState)
llm = ChatOpenAI(model="gpt-4o")

def call_model(state):
    response = llm.invoke(state["messages"])
    return {"messages": state["messages"] + [response]}

graph_builder.add_node("llm_call", call_model)
graph_builder.set_entry_point("llm_call")
graph_builder.set_finish_point("llm_call")
graph = graph_builder.compile()

# Invoke the graph with tracing configuration
# LangSmith automatically picks up traces if env vars are set.
# For other tools or explicit control, use the 'callbacks' config.
config = {"callbacks": [langfuse_handler]}
initial_state = {"messages": [HumanMessage(content="Explain LangGraph observability.")]}

# The invocation will be traced in the configured platform (LangSmith or Langfuse)
for s in graph.stream(initial_state, config=config):
    print(s)
```

Code Snippet based on concepts from [19]

Why: Advanced Debugging and Human Oversight
The tight integration between LangGraph and LangSmith enables unique and powerful features. "Time-travel" debugging allows developers to inspect an agent's state at any point in the graph, rewind it, and manually intervene to steer the agent down a different path.16 Furthermore, LangGraph's architecture natively supports human-in-the-loop checkpoints, allowing for moderation, quality control, and explicit approval of critical agent actions, all of which are visualized and managed within LangSmith.16

AutoGen: Implementing OpenTelemetry for Event-Driven Agent Architectures

AutoGen, developed by Microsoft, has evolved into an asynchronous, event-driven framework and has embraced OpenTelemetry (OTel) as its standard for observability.[22] This represents a "Standardized Interoperability" philosophy, prioritizing open standards and compatibility with a wide range of backend systems over a single, proprietary solution.

Pattern: Filtered, Hierarchical OTel Tracing
The core challenge in observing AutoGen is managing the high volume of internal telemetry it produces. A naive OTel integration can result in noisy, indecipherable traces.23 The established best practice is to implement a smart filtering layer that reduces trace volume by up to 80% while retaining the most meaningful information.23 Traces should then be organized into a logical hierarchy that reflects the agent's workflow, typically with three levels: a top-level
Workflow span for the entire user request, Phase spans for major stages like initialization and execution, and Operation spans for individual tool calls or LLM invocations.[23]

How: Custom Span Processors and OTel SDK Configuration
Achieving this requires a custom tracing module that configures the OpenTelemetry SDK with a custom span processor. This processor implements the filtering logic, including pattern-based include/exclude rules and inspection of span attributes, to discard irrelevant internal telemetry.23
Architecture Diagram: AutoGen Observability with Custom OTel Filtering

```
+-------------------+     +------------------------+     +--------------------+

| AutoGen Agent App |----->| OTel SDK with Custom |----->| OTel Collector |
| (High-volume raw | | Span Processor (Filter) | | (Optional) |
| telemetry) | +------------------------+     +--------------------+
+-------------------+ | |

| (Filtered, structured traces) |
                              v                     v
                       +------------------+    +------------------------+

| Jaeger / Zipkin | | Datadog / New Relic / |
| (Local Viz) | | Other Backends |
                       +------------------+    +------------------------+
```

Diagram based on concepts from [23]

Why: Flexibility and Ecosystem Compatibility
By adopting OpenTelemetry, AutoGen ensures that its users are not locked into a single observability vendor. They can export telemetry data to any OTel-compatible backend, from open-source tools like Jaeger for local development to enterprise platforms like Datadog for

production monitoring.23 The official documentation also highlights third-party tools like AgentOps, which build upon this OTel foundation to provide specialized features like session replays, cost management, and security monitoring tailored for AutoGen.25

CrewAI: Integrating Third-Party Platforms for Role-Based Workflow Observability

CrewAI adopts a "Flexible Agnosticism" philosophy. It does not promote a single, first-party observability solution or mandate a specific standard like OpenTelemetry. Instead, it provides a flexible architecture with hooks that encourage a vibrant ecosystem of third-party integrations.[26] This approach offers maximum choice to the developer but also places the responsibility of selecting, integrating, and maintaining the observability stack squarely on them.

Pattern: Span-per-Action Tracing
CrewAI's architecture is centered on role-playing agents that collaborate to complete tasks. The recommended observability pattern is to treat each significant agent action—such as sending a message, delegating a task, or using a tool—as an individual span within a larger trace that represents the entire crew's "kickoff" process.18 This allows for detailed analysis of both individual agent performance and the overall collaborative workflow.
How: SDK-Based Integrations
Observability is implemented by installing and initializing the SDK of a chosen third-party platform. The CrewAI framework is designed to be easily instrumented by these SDKs.
Example: Integrating CrewAI with W&B Weave

Python

```python
# examples/crewai_weave_tracing.py
import os
import weave
from crewai import Agent, Task, Crew, Process

# 1. Initialize Weave (requires W&B account and `wandb login`)
# This automatically patches CrewAI to capture traces.
weave.init(project_name="crewai_observability_demo")

# Set up your LLM API key
# os.environ = "YOUR_API_KEY"
# 2. Define your agents and tasks as usual
researcher = Agent(
  role='Senior Research Analyst',
  goal='Uncover cutting-edge developments in AI and data science',
  backstory="You are a renowned analyst at a top tech think tank.",
  verbose=True,
  allow_delegation=False
)
writer = Agent(
  role='Tech Content Strategist',
  goal='Craft compelling content on tech advancements',
  backstory="You are a skilled writer known for making complex topics accessible.",
  verbose=True
)

task1 = Task(
  description='Investigate the latest trends in multi-agent system observability.',
  expected_output='A 300-word summary of the top 3 trends.',
  agent=researcher
)
task2 = Task(
  description='Write an engaging blog post based on the research summary.',
  expected_output='A 500-word blog post with a catchy title.',
  agent=writer
)

# 3. Instantiate and run the crew
crew = Crew(
  agents=[researcher, writer],
  tasks=[task1, task2],
  process=Process.sequential
)

# The kickoff() call will be automatically traced by Weave.
# A link to the trace will be printed in the console.
result = crew.kickoff()
print(result)
```

Code Snippet based on concepts from [27]

Why: Maximum Choice and Specialized Solutions
The agnostic approach of CrewAI empowers developers to choose the best tool for their

specific needs. For teams already using Weights & Biases for model experimentation, the seamless W&B Weave integration is a natural fit.27 For those who prefer an open-source, OTel-native approach, OpenLIT provides a one-line setup.29 For developers seeking a comprehensive LLM engineering platform with prompt management, Langfuse is an excellent option, integrated via the OpenInference SDK.30 This flexibility allows teams to align their observability tooling with their existing MLOps stack and preferences.

Observability for AI-Assisted Development Environments: The Case of Cursor

AI-powered code editors like Cursor introduce a unique set of observability challenges. Unlike agents that operate within a sandboxed application, Cursor's agent directly interacts with a developer's local and remote codebase, executing commands and modifying files.[31] This necessitates a dual approach to observability: monitoring the code the agent

produces (internal observability) and monitoring the agent's *actions* on the system (external observability). The latter has driven the development of cutting-edge, instrumentation-free techniques that represent a significant evolution in security and monitoring for autonomous systems.

Internal Observability: Applying OpenTelemetry to Agent-Generated Code

The code generated by Cursor becomes an integral part of an application and must therefore adhere to the same production-grade observability standards as human-written code.[34] The best practices for instrumenting this code are well-established and center on the OpenTelemetry standard.

Pattern: Standard OpenTelemetry Instrumentation
Cursor's own internal documentation, framed as a rule for its agent, explicitly promotes the use of OpenTelemetry for distributed tracing, metrics, and structured logging within the Go microservices it helps build.34 This includes several key practices:

- **Distributed Tracing:** Starting and propagating tracing spans across all service boundaries (HTTP, gRPC, database calls, external APIs) to create a complete picture of a request's lifecycle.
- **Structured Logging:** Emitting logs in a machine-readable format like JSON and using log

correlation by injecting trace_id and span_id into every log message. This allows for seamless navigation between traces and logs in observability backends.
- **Key Metrics Monitoring:** Tracking the "golden signals" of application health: request latency, throughput (requests per second), error rate, and resource usage.[34]

This demonstrates a critical principle: AI-assisted code generation does not absolve developers of the responsibility to build observable systems. The generated code must be instrumented and monitored with the same rigor as any other production service.

External Observability: Boundary Tracing with eBPF for Instrumentation-Free Monitoring

While instrumenting the generated code is necessary, it is not sufficient for fully understanding the behavior of an autonomous agent like Cursor. The agent itself operates as a black box, and its ability to execute arbitrary commands presents a significant security and reliability challenge.[4] This has led to the development of an "outside-in" monitoring paradigm.

Pattern: Boundary Tracing via eBPF
A recent academic paper introduces a framework named AgentSight, which is designed to observe agents like Cursor from outside their application code, thus providing a tamper-proof view of their actions.4 This technique, called "boundary tracing," leverages eBPF (extended Berkeley Packet Filter), a powerful Linux kernel technology that allows for safe and efficient monitoring of system-level events without modifying application code.4
The core insight behind boundary tracing is that while an agent's internal logic is complex and its codebase may be inaccessible, its interactions with the outside world occur at stable, unavoidable system interfaces: the network stack for API calls and the kernel for system operations (e.g., file access, process execution).[4] By monitoring these boundaries, AgentSight can bridge the "semantic gap" between the agent's intent and its actions.

How: Correlating Network and Kernel Events
The AgentSight architecture uses eBPF for two primary functions:
1. **TLS Interception:** It intercepts TLS-encrypted network traffic between the agent and the LLM backend. This allows it to extract the agent's high-level semantic intent by capturing the raw prompts and model responses.
2. **Kernel Event Monitoring:** It monitors kernel events, such as syscalls, to observe the agent's low-level system effects—every file it reads, writes, or deletes, and every process it spawns.

A real-time correlation engine then links the semantic intent from the network stream with the system actions from the kernel stream, creating a unified, causally-linked trace.[4] This powerful, instrumentation-free approach can detect malicious behaviors like prompt injection attacks, resource-wasting reasoning loops, and security vulnerabilities that traditional, instrumentation-based methods might miss entirely.[4] This shift from cooperative, "inside-out" instrumentation to adversarial, "outside-in" observation represents a necessary evolution in security posture for the age of autonomous agents.

Implementing End-to-End Tracing with OpenTelemetry for GenAI

OpenTelemetry (OTel) has emerged as the undisputed industry standard for observability, providing a vendor-agnostic framework for generating, collecting, and exporting telemetry data.[11] For Generative AI applications, adopting OTel is a critical best practice that ensures interoperability, avoids vendor lock-in, and provides a standardized language for describing complex agentic workflows.

Instrumentation Strategies: Auto-Instrumentation vs. Manual Span Creation

OpenTelemetry offers two primary methods for instrumenting an application, and the most effective strategy often involves a hybrid of both.[37]

Auto-Instrumentation: This approach involves using language-specific libraries that automatically "patch" common frameworks and libraries—such as the OpenAI SDK, LangChain, or HTTP clients—to emit spans for their operations without requiring any changes to the application's source code.[11] This is the fastest way to achieve baseline visibility and is highly effective for capturing interactions with third-party components.

Manual Span Creation: For an application's core business logic, manual instrumentation provides granular control over the generated telemetry. By explicitly creating spans around specific functions or blocks of code, developers can enrich traces with custom attributes that provide deep, business-specific context.[11] These attributes can include identifiers like

user_id or session_id, configuration parameters like model_version, or intermediate computational results, all of which are invaluable for debugging complex issues.[11]

Example: Hybrid OTel Instrumentation in a Python RAG Application

Python

```python
# examples/otel_hybrid_instrumentation.py
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import ConsoleSpanExporter, SimpleSpanProcessor

# --- SDK Setup (typically done once at application startup) ---
trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    SimpleSpanProcessor(ConsoleSpanExporter())
)
tracer = trace.get_tracer(__name__)

# Assume 'openai' and 'chromadb' libraries are auto-instrumented elsewhere.
# Auto-instrumentation would automatically create spans for the calls below.
import openai
import chromadb

client = openai.OpenAI()
db_client = chromadb.Client()

# --- Manual Instrumentation for Business Logic ---
def process_user_query(query: str, user_id: str):
    # Create a manual span for the overarching business logic
    with tracer.start_as_current_span("process_user_query") as parent_span:
        # Add business-specific attributes
        parent_span.set_attribute("app.user.id", user_id)
        parent_span.set_attribute("app.query.text", query)

        # 1. Retrieve context (this call would be auto-instrumented)
        collection = db_client.get_collection("documents")
        retrieved_docs = collection.query(query_texts=[query], n_results=2)
        context = " ".join(retrieved_docs['documents'])

        # 2. Generate response (this call would be auto-instrumented)
        response = client.chat.completions.create(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": f"Context: {context}"},
                {"role": "user", "content": query}
            ]
        )
        final_answer = response.choices.message.content
        parent_span.set_attribute("app.response.text", final_answer)

        return final_answer

# Run the function
process_user_query("What is OpenTelemetry?", "user-12345")
```

Code Snippet based on concepts from [11]

The OpenTelemetry Collector for GenAI: Architecture and Configuration

For any production-grade deployment, using the OpenTelemetry Collector is a non-negotiable best practice.[24] The Collector is a vendor-agnostic proxy that acts as a centralized hub for all telemetry data.[36]

Its architecture is based on pipelines, which consist of three main components [37]:

1. **Receivers:** Ingest telemetry data from various sources. The most common receiver for custom applications is the OTLP receiver, which listens for data sent over the standard OpenTelemetry Protocol.
2. **Processors:** Modify or filter data as it flows through the pipeline. Processors can be used to batch data for efficiency, add metadata, or sample traces to control costs.
3. **Exporters:** Send the processed data to one or more observability backends. The Collector can be configured to send the same data stream to multiple destinations simultaneously—for example, to a Jaeger instance for engineering teams and a Prometheus instance for operations teams.

Using a Collector decouples the application from the specific observability backend, allowing teams to change or add backends with a simple configuration change in the Collector, rather than redeploying the application.[24]

Applying GenAI Semantic Conventions for Standardized Traces

Semantic conventions are standardized schemas for telemetry attributes, defining common names, types, and meanings for various operations.[40] Adhering to these conventions is paramount, as it ensures that telemetry data is consistent and automatically understandable by any compliant observability platform.

OpenTelemetry is actively developing a set of semantic conventions specifically for Generative AI applications.[41] While still evolving, these conventions provide a standard vocabulary for describing the components of an LLM or agentic workflow. Consistently applying these attributes is a critical best practice.

Key GenAI semantic conventions include [43]:

- **LLM Operations:** Attributes like gen_ai.system (e.g., "OpenAI"), gen_ai.request.model (e.g., "gpt-4o"), gen_ai.prompt, gen_ai.completion, and detailed token counts (gen_ai.usage.prompt_tokens, gen_ai.usage.completion_tokens).
- **Vector Database Operations:** Attributes such as db.system (e.g., "ChromaDB"), db.vector.query.top_k, and attributes for query results including the document ID, score, and distance.
- **LLM Frameworks and Agents:** An attribute like traceloop.span.kind (proposed by OpenLLMetry) to classify the type of operation (e.g., workflow, task, agent, tool), which provides crucial structural context to the trace.

The standardization process for these conventions is ongoing, which has created a "semantic convention lag." Specialized observability platforms often provide their own stable, opinionated implementations of these conventions, offering value by abstracting away the evolving standard. However, aligning custom instrumentation with the emerging official conventions is the most future-proof strategy for ensuring long-term interoperability.

Comparative Analysis of Production-Grade Observability Platforms

The LLM and agent observability market is rapidly maturing, offering a diverse landscape of tools that range from open-source, developer-centric platforms to comprehensive, enterprise-grade APM suites. Selecting the right platform requires a clear understanding of a team's specific use case, existing infrastructure, and organizational priorities. The optimal choice often depends on whether the primary need is for deep, iterative development and debugging, or for unified, operational monitoring in a broader enterprise context.

Open-Source vs. Managed Solutions: Use-Case Boundaries for Langfuse and Datadog

Langfuse has established itself as a leading open-source LLM engineering platform, designed by and for developers building AI applications.[44] Its core strengths lie in features that accelerate the inner development loop:

- **Collaborative Debugging:** Traces can be shared via URL, and team members can add comments directly to specific spans, facilitating issue resolution.[45]
- **Integrated Prompt Management:** Langfuse provides a UI for versioning, testing, and deploying prompts, decoupling prompt engineering from the application code.[45]
- **Granular Analytics:** It offers detailed dashboards for tracking cost, latency, and quality metrics, allowing for precise analysis of user sessions and model performance.[21]
- **Data Control:** As an open-source tool, it can be self-hosted, giving organizations complete control over their sensitive telemetry data.[44]

Datadog LLM Observability is an extension of Datadog's market-leading enterprise APM platform.[15] Its primary value proposition is the unification of LLM telemetry with traditional infrastructure and application metrics in a single pane of glass.[47] This is particularly compelling for organizations already invested in the Datadog ecosystem. Key features include:

- **Unified Monitoring:** Correlates LLM traces with APM traces, logs, and infrastructure metrics, providing a holistic view of application health.[15]
- **Out-of-the-Box Evaluations:** Provides built-in checks for quality (e.g., failure to answer, off-topic responses) and security (e.g., PII detection, prompt injection attempts).[15]
- **Enterprise-Grade Operations:** Leverages Datadog's robust alerting, dashboarding, and incident management capabilities.[15]

Use-Case Boundary:

- **Langfuse** is optimally suited for **AI/ML development teams** whose primary focus is on the iterative process of building, debugging, and refining LLM applications. Its prompt management and collaborative features are tailored for this workflow.
- **Datadog** is the ideal choice for **enterprise operations and SRE teams** who need to monitor the performance, cost, and reliability of LLM applications as one component within a larger, complex production environment.

Enterprise-Grade Agentic Observability: A Comparative Look at Maxim AI and Fiddler AI

Fiddler AI positions its "Agentic Observability" platform as a comprehensive **control layer** for building transparent, accountable, and compliant AI systems.[1] Its approach is heavily focused on governance and risk management.

- **Hierarchical Analysis:** Emphasizes a hierarchical view (session -> agent -> trace -> span) for intuitive root cause analysis in complex multi-agent systems.[1]
- **Guardrails and Trust Models:** Offers over 80 pre-built metrics for evaluating safety, faithfulness, and PII leakage, allowing teams to enforce policies and standards.[49]
- **Framework Support:** Provides native support for frameworks like LangGraph and Amazon Bedrock, along with OTel integration for broader compatibility.[5]

Maxim AI also targets the enterprise market but with a strong emphasis on **developer experience, interoperability, and real-time evaluation**.

- **Comprehensive Tracing:** Features robust, stateless SDKs for major frameworks (LangGraph, CrewAI, OpenAI Agents) and is fully OTel-compatible.[44]
- **Real-Time Evaluation:** Supports both automated (online) and human-in-the-loop evaluation workflows, enabling continuous quality monitoring in production.[18]
- **Trace Forwarding:** A key differentiator is its ability to act as an observability hub, ingesting traces and then forwarding them to other OTel-compatible platforms like Datadog or New Relic, allowing teams to unify their observability data without multiple instrumentation steps.[50]

Use-Case Boundary:

- **Fiddler AI** is best suited for organizations in **highly regulated industries** or those with a primary focus on AI governance, risk, and compliance (GRC). Its "control layer" and "Guardrails" framing directly addresses these needs.
- **Maxim AI** is an excellent fit for enterprises that prioritize **developer velocity, architectural flexibility, and integration** with a diverse set of existing tools. Its stateless, OTel-native architecture and trace forwarding capabilities provide maximum interoperability.

Specialized Tooling: W&B Weave, OpenLIT, and Portkey

Beyond the comprehensive platforms, a number of specialized tools address specific niches within the LLMOps lifecycle.

- **W&B Weave:** Hailing from Weights & Biases, Weave is tightly integrated with the MLOps workflow of experimentation and evaluation.[28] Its standout feature is the seamless, zero-config, automatic tracing for CrewAI applications, making it the default choice for teams already using the W&B ecosystem.[27]
- **OpenLIT:** This is a pure-play, open-source, OTel-native observability tool.[29] Its value proposition is its simplicity and completeness as a self-hostable solution. With a one-line initialization, it provides tracing and metrics for LLMs, VectorDBs, and even GPU usage, making it ideal for teams wanting a straightforward, open-standard solution without the complexity of a large enterprise platform.[29]
- **Portkey:** Portkey operates as an **AI Gateway**, providing observability as a feature of its core function, which is to manage and optimize API calls to various LLM providers.[26] By routing traffic through its proxy, it can offer capabilities like semantic caching, automatic retries and fallbacks, and load balancing, in addition to logging and tracing all requests.[8] It is the best choice for teams that need to manage a heterogeneous environment of multiple LLM providers and want to centralize control, optimization, and monitoring at the gateway level.

This diverse market landscape reflects the varying needs of teams building with AI. While some require all-in-one platforms that bundle observability with other LLMOps functions, others prefer best-in-class point solutions that adhere to open standards. The choice of tool is a critical architectural decision that depends on a team's scale, existing infrastructure, and primary objectives.

Table 1: Comparative Matrix of LLM/MAS Observability Platforms

Platform	Primary Use Case	Deployment Model	OpenTelemetry Support	Key Differentiator	Pricing Model
Langfuse	LLM App Development & Debugging	SaaS / Self-Hosted	Compatible	Integrated Prompt Management & Collaboration	Open-Source (Free), Cloud (Usage-based)

Datadog	Enterprise APM & Ops Monitoring	SaaS	Compatible	Unified view of LLM, APM, and infra metrics	Usage-based (Traces/Logs Ingested)
Maxim AI	Enterprise Agent Observability	SaaS / In-VPC	Native / Compatible	Stateless SDKs, Real-time Evals, Trace Forwarding	Custom (Enterprise)
Fiddler AI	AI Governance, Risk & Compliance	SaaS / In-VPC	Compatible	Hierarchical Analysis, Guardrails & Trust Models	Custom (Enterprise)
LangSmith	LangChain/ LangGraph Development	SaaS / Self-Hosted (Ent)	OTel Client Support	Deep integration with LangChain, "Time-travel" debugging	Free Tier, Plus (Per User), Enterprise
W&B Weave	MLOps Experimentation & Tracing	SaaS	No (Proprietary)	Seamless, automatic tracing for CrewAI	Free Tier, Usage-based
OpenLIT	OTel-Native Open Source Monitoring	Self-Hosted	Native	Simple, one-line setup for LLMs, VectorDBs, GPUs	Open-Source (Free)
Portkey	AI Gateway & LLM Ops	SaaS / Self-Hosted	Compatible	Unified API, Semantic	Free Tier, Usage-bas

				Caching, Load Balancing	ed (Requests)

Table data synthesized from.[1]

Advanced Monitoring, Evaluation, and Cost Optimization Strategies

Effective management of production-grade agentic systems requires moving beyond basic tracing and implementing a sophisticated, multi-layered strategy for monitoring, evaluation, and continuous optimization. This involves establishing real-time feedback loops, combining automated and human-centric quality assessments, and actively managing the significant operational costs associated with LLM inference.

Real-Time Monitoring and Alerting

The non-deterministic nature of AI agents means that failures can be subtle and emerge unexpectedly in production. Therefore, continuous, real-time monitoring and alerting are essential for detecting performance regressions, quality degradation, or anomalous behavior before they impact users.[2]

- **Alerting on Key Metrics:** Teams should configure real-time alerts for critical AI-specific metrics. This includes setting thresholds for spikes in P95 latency, sudden increases in error rates, and cost anomalies indicated by excessive token usage.[15] These alerts should be routed to appropriate channels like Slack or PagerDuty for immediate response.[18]
- **Monitoring for Drift and Quality Degradation:** Alerts should also be configured for drops in quality scores from online evaluators or a sustained increase in negative user feedback. This serves as an early warning system for model drift or emerging issues with prompt templates.[10]
- **Human-in-the-Loop Workflows:** A key best practice is to integrate human annotation and review directly into the monitoring workflow.[16] When automated systems detect low-quality outputs, negative user feedback, or potential safety violations, these specific traces can be automatically routed to a queue for review by subject-matter experts. This

"human-in-the-loop" process is invaluable for validating nuanced criteria, handling edge cases, and providing high-quality data for fine-tuning models or evaluators.[18]

Evaluation Strategies: Offline vs. Online

A comprehensive evaluation strategy must encompass both offline testing in a controlled environment and online monitoring of live production traffic. These two modes are complementary and create a powerful feedback loop for continuous improvement.[11]

- **Offline Evaluation:** This involves testing an agent against a curated, benchmark dataset (a "golden set") where the inputs and expected outputs are known.[11] This is typically integrated into a CI/CD pipeline to guard against regressions before deployment.[2] The primary challenge is ensuring the dataset remains representative of real-world user interactions.[11]
- **Online Evaluation:** This involves running automated evaluators on a sample of live production traffic to assess performance in real-time.[11] This is crucial for detecting issues that were not anticipated in the offline dataset, such as model drift or unexpected user behavior.[15]
- **LLM-as-a-Judge:** A powerful technique used in both offline and online evaluation is "LLM-as-a-Judge".[11] This involves using a separate, powerful LLM (like GPT-4o) to score an agent's output against criteria such as correctness, relevance, groundedness, and adherence to safety guidelines.[13] While highly flexible, these judges must be carefully calibrated and validated against human judgments to ensure their assessments are reliable.[13]

Cost, Latency, and Token Usage Optimization

The operational costs of LLM agents can escalate rapidly if not actively managed. Observability data is the key to identifying and acting on optimization opportunities.[10]

- **Identifying High-Cost Operations:** The first step is to use trace data to pinpoint the most token-intensive calls in any given workflow.[8] Dashboards that track cost per trace, per user, or per agent are essential for this analysis.[10]
- **Model Selection Strategy:** Not all tasks require the most powerful (and expensive) model. A key optimization strategy is to use smaller, faster, and cheaper models (Small Language Models or SLMs) for simpler tasks like intent classification or parameter

extraction, while reserving larger models for complex reasoning.[11] An evaluation framework is critical for comparing the performance-to-cost trade-offs of different models for a given task.[2]
- **Semantic Caching:** For frequently repeated queries, semantic caching can dramatically reduce both latency and cost. Instead of re-computing a response, a caching layer can store the results of previous LLM calls and serve them directly if a new, semantically similar query is received. This is a feature offered by AI gateways like Portkey.[44]
- **Prompt Engineering:** Inefficient prompts can significantly increase token consumption. Analyzing trace data can reveal prompts that are overly verbose or that lead the model to generate unnecessarily long responses. Refining these prompts can yield significant cost savings.[10]

Data Schemas and Standards for Interoperability

As the LLMOps ecosystem matures, the adoption of standardized data formats for telemetry and structured outputs becomes increasingly critical. Standards like JSON Schema and OpenTelemetry's semantic conventions are the foundation for building interoperable, maintainable, and scalable AI systems. They provide a common language that enables different components and tools to communicate effectively.

JSON Schema for Structured Outputs and Traces

Forcing an LLM to generate output that conforms to a specific JSON schema is a powerful technique for creating reliable, predictable applications. This ensures that the model's output is machine-parsable and can be validated against a predefined structure, which is essential for agentic tool use and data processing pipelines.[56]

Pattern: Pydantic-based Schema Definition and Parsing
Within the Python ecosystem, the recommended best practice is to use Pydantic models to define the desired output schema. LangChain, for example, provides a JsonOutputParser that can take a Pydantic model and automatically generate the necessary formatting instructions to inject into the prompt, guiding the LLM to produce a compliant JSON object.57
Example: Using Pydantic and LangChain for Structured JSON Output

Python

```python
# examples/langchain_structured_output.py
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

# 1. Define the desired data structure using Pydantic
class Joke(BaseModel):
    setup: str = Field(description="The setup or question part of the joke")
    punchline: str = Field(description="The punchline or answer to the joke")
    rating: int = Field(description="A rating of the joke's funniness from 1 to 10")

# 2. Set up a parser that uses the Pydantic object's schema
parser = JsonOutputParser(pydantic_object=Joke)

# 3. Create a prompt template that includes the auto-generated format instructions
prompt = PromptTemplate(
    template="Answer the user's query.\n{format_instructions}\nQuery: {query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)

# 4. Create and invoke the chain
model = ChatOpenAI(model="gpt-4o", temperature=0)
chain = prompt | model | parser
response = chain.invoke({"query": "Tell me a joke about software developers."})

print(response)
# Expected output:
# {'setup': 'Why do programmers prefer dark mode?',
#  'punchline': 'Because light attracts bugs!',
#  'rating': 8}
```

Code Snippet based on concepts from [56]

This pattern is not only useful for application logic but also for defining the structure of trace data itself. A well-defined JSON schema for trace inputs and metadata ensures that all telemetry data is consistent and can be reliably processed and queried by observability platforms.[58]

Example: JSON Schema for a LangChain Trace Input

JSON

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "LangChainTraceInputSchema",
  "type": "object",
  "properties": {
    "params": {
      "type": "object",
      "properties": {
        "content": {
          "type": "string",
          "description": "The primary user prompt or conversation text."
        },
        "user_id": {
          "type": "string",
          "description": "Unique identifier for the user."
        },
        "session_id": {
          "type": "string",
          "description": "Unique identifier for the conversation session."
        },
        "metadata": {
          "type": "object",
          "description": "Additional key-value metadata for filtering and analysis."
        }
      },
      "required": ["content"]
    }
  },
  "required": ["params"],
  "additionalProperties": true
}
```

Schema based on concepts from [19]

OpenTelemetry GenAI Semantic Conventions Reference

As detailed in Section 4, OpenTelemetry's semantic conventions provide the standardized vocabulary for creating interoperable traces. The table below serves as a practical reference for the most critical attributes defined in the emerging GenAI conventions, which should be applied to all instrumented spans to ensure they are correctly interpreted by observability backends.

Table 2: OpenTelemetry GenAI Semantic Conventions Reference

Attribute Name	Type	Description	Example Value	Applicable Span(s)
gen_ai.system	string	The AI system or provider being used.	OpenAI, Anthropic, Bedrock	LLM, Embedding
gen_ai.request.model	string	The specific model requested for the operation.	gpt-4o, claude-3-opus-20240229	LLM, Embedding
gen_ai.usage.prompt_tokens	int	The number of tokens in the input prompt.	128	LLM
gen_ai.usage.completion_tokens	int	The number of tokens in the generated completion.	256	LLM
gen_ai.request.temperature	double	The temperature setting for the generation request.	0.7	LLM

gen_ai.prompt	array[string]	The full prompt(s) sent to the model.	``	LLM
gen_ai.completion	array[string]	The full completion(s) received from the model.	``	LLM
db.system	string	The vector database system being used.	ChromaDB, Pinecone, Qdrant	Vector DB
db.vector.query.top_k	int	The number of nearest neighbors requested in a vector search.	5	Vector DB
traceloop.span.kind	string	The type of operation within an agentic framework.	agent, workflow, tool, task	All Framework Spans
traceloop.entity.name	string	The name of the specific framework component being executed.	ResearcherAgent, CodeInterpreterTool	All Framework Spans

Table data synthesized from the proposed OpenLLMetry conventions, which are informing the official OpenTelemetry standard.[43]

Conclusion

The landscape of observability for LLM and Multi-Agent Systems is undergoing a rapid and profound transformation, moving decisively away from the paradigms of traditional software monitoring. The analysis of recent best practices, frameworks, and tooling reveals several critical conclusions for architects and engineers building production-grade AI systems.

First, the core challenge has shifted from operational monitoring to **semantic observability**. The primary questions are no longer just about latency and error rates but about quality, correctness, safety, and cost. This necessitates a new class of metrics and evaluation strategies, such as LLM-as-a-Judge and the analysis of user feedback, which must be treated as first-class signals of system health.

Second, a clear **spectrum of architectural philosophies** has emerged among the leading agentic frameworks. LangGraph's integrated ecosystem with LangSmith offers unparalleled depth and developer experience for those within its walled garden. AutoGen's commitment to OpenTelemetry champions a future of standardized interoperability, albeit with greater implementation complexity. CrewAI's platform-agnosticism provides maximum flexibility but places the onus of integration and standardization on the developer. The choice between these models is a foundational architectural decision with long-term consequences for tooling, scalability, and vendor dependency.

Third, **OpenTelemetry is the undisputed standard** for ensuring future-proof interoperability. While specialized platforms offer powerful, out-of-the-box solutions, those built on an OTel-compatible foundation, such as Maxim AI and Fiddler AI, provide the most robust enterprise strategy. Adherence to the emerging GenAI semantic conventions is not merely a recommendation but a requirement for building systems that are maintainable and observable across a heterogeneous toolchain.

Finally, the rise of autonomous agents like Cursor, which operate directly on developer environments, is forcing a security-driven shift toward **"outside-in" monitoring**. Techniques like boundary tracing with eBPF represent the future of agent observability, providing an un-bypassable, instrumentation-free method for correlating high-level intent with low-level system actions.

Ultimately, building reliable, scalable, and trustworthy AI agent systems is not possible without a deep, hierarchical, and semantically aware observability strategy. The practices and tools outlined in this report provide a comprehensive blueprint for achieving this, enabling teams to move from prototype to production with confidence.

Cytowane prace

1. Agentic Observability Starts in Development: Build Reliable Agentic Systems - Fiddler AI, otwierano: września 22, 2025,

https://www.fiddler.ai/blog/agentic-observability-development
2. Agent Factory: Top 5 agent observability best practices for reliable AI | Microsoft Azure Blog, otwierano: września 22, 2025, https://azure.microsoft.com/en-us/blog/agent-factory-top-5-agent-observability-best-practices-for-reliable-ai/
3. A practical guide for AI observability for agents (2025 edition) - Vellum AI, otwierano: września 22, 2025, https://www.vellum.ai/blog/understanding-your-agents-behavior-in-production
4. AgentSight: System-Level Observability for AI Agents Using eBPF - arXiv, otwierano: września 22, 2025, https://arxiv.org/html/2508.02736v1
5. Fiddler Agentic Observability, otwierano: września 22, 2025, https://www.fiddler.ai/agentic-observability
6. What Is Observability? Key Components and Best Practices - Honeycomb, otwierano: września 22, 2025, https://www.honeycomb.io/blog/what-is-observability-key-components-best-practices
7. Monitoring and Observability: A Developer's Guide to Keeping Systems Healthy - Medium, otwierano: września 22, 2025, https://medium.com/@jesuva/monitoring-and-observability-a-developers-guide-to-keeping-systems-healthy-23d3ccfdf8d3
8. How LLM tracing helps you debug and optimize GenAI apps - Portkey, otwierano: września 22, 2025, https://portkey.ai/blog/llm-tracing-to-debug-and-optimize-genai-apps
9. Advanced Techniques for Monitoring Traces in AI Workflows - Coralogix, otwierano: września 22, 2025, https://coralogix.com/ai-blog/advanced-techniques-for-monitoring-traces-in-ai-workflows/
10. Why observability is essential for AI agents | IBM, otwierano: września 22, 2025, https://www.ibm.com/think/insights/ai-agent-observability
11. AI Agents in Production: Observability & Evaluation | ai-agents-for ..., otwierano: września 22, 2025, https://microsoft.github.io/ai-agents-for-beginners/10-ai-agents-production/
12. What is LLM Observability? - IBM, otwierano: września 22, 2025, https://www.ibm.com/think/topics/llm-observability
13. How quality, cost, and latency are assessed by Agent Evaluation ..., otwierano: września 22, 2025, https://learn.microsoft.com/en-us/azure/databricks/generative-ai/agent-evaluation/llm-judge-metrics
14. How quality, cost, and latency are assessed by Agent Evaluation (MLflow 2), otwierano: września 22, 2025, https://docs.databricks.com/aws/en/generative-ai/agent-evaluation/llm-judge-metrics
15. LLM Observability - Datadog, otwierano: września 22, 2025, https://www.datadoghq.com/product/llm-observability/
16. LangGraph - LangChain, otwierano: września 22, 2025,

https://www.langchain.com/langgraph
17. LangGraph Platform - LangChain, otwierano: września 22, 2025, https://www.langchain.com/langgraph-platform
18. Observability for AI Agents: LangGraph, OpenAI Agents, and Crew AI - Maxim AI, otwierano: września 22, 2025, https://www.getmaxim.ai/articles/observability-for-ai-agents-langgraph-openai-agents-and-crew-ai/
19. Observability - Docs by LangChain, otwierano: września 22, 2025, https://docs.langchain.com/oss/python/langgraph/observability
20. Open Source Observability for LangGraph - Langfuse, otwierano: września 22, 2025, https://langfuse.com/guides/cookbook/integration_langgraph
21. Example - Trace and Evaluate LangGraph Agents - Langfuse, otwierano: września 22, 2025, https://langfuse.com/guides/cookbook/example_langgraph_agents
22. AutoGen - Microsoft Research, otwierano: września 22, 2025, https://www.microsoft.com/en-us/research/project/autogen/
23. Building Better Observability for AutoGen Multi-Agent Systems | by ..., otwierano: września 22, 2025, https://medium.com/@milenppavlov/building-better-observability-for-autogen-multi-agent-systems-b0c93b8a50e6
24. What is OpenTelemetry? A Developer's Guide to Observability and AI Integration | by Mark Henry | Sep, 2025 | Medium, otwierano: września 22, 2025, https://medium.com/@mark_henry_01/what-is-opentelemetry-a-developers-guide-to-observability-and-ai-integration-3174f6265e1a
25. Agent Monitoring and Debugging with AgentOps | AutoGen 0.2, otwierano: września 22, 2025, https://microsoft.github.io/autogen/0.2/docs/ecosystem/agentops/
26. Overview - CrewAI Documentation, otwierano: września 22, 2025, https://docs.crewai.com/observability/overview
27. Tracing your CrewAI application | genai-research – Weights & Biases, otwierano: września 22, 2025, https://wandb.ai/onlineinference/genai-research/reports/Tracing-your-CrewAI-application--VmlldzoxMzQ5MDcwNA
28. Weave Integration - CrewAI Documentation, otwierano: września 22, 2025, https://docs.crewai.com/en/observability/weave
29. OpenLIT Integration - CrewAI, otwierano: września 22, 2025, https://docs.crewai.com/observability/openlit
30. Observability for CrewAI with Langfuse Integration - Langfuse, otwierano: września 22, 2025, https://langfuse.com/integrations/frameworks/crewai
31. Cursor: The best way to code with AI, otwierano: września 22, 2025, https://cursor.com/
32. Background Agents | Cursor Docs, otwierano: września 22, 2025, https://cursor.com/docs/background-agent
33. Agents | Cursor Learn, otwierano: września 22, 2025, https://cursor.com/learn/agents
34. Rules for Observability | Cursor Directory, otwierano: września 22, 2025,

https://cursor.directory/rules/observability
35. Rules for Best Practices - Cursor Directory, otwierano: września 22, 2025, https://cursor.directory/rules/best-practices
36. What is OpenTelemetry?, otwierano: września 22, 2025, https://opentelemetry.io/docs/what-is-opentelemetry/
37. The AI Engineer's Guide to LLM Observability with OpenTelemetry, otwierano: września 22, 2025, https://agenta.ai/blog/the-ai-engineer-s-guide-to-llm-observability-with-opentelemetry
38. Guide to Monitoring LLMs with OpenTelemetry - Ghost, otwierano: września 22, 2025, https://latitude-blog.ghost.io/blog/guide-to-monitoring-llms-with-opentelemetry/
39. This enables easy building and running the Fiddler receiver within a custom OpenTelemetry collector - GitHub, otwierano: września 22, 2025, https://github.com/fiddler-labs/otel-collector-with-fiddler
40. Semantic Conventions - OpenTelemetry, otwierano: września 22, 2025, https://opentelemetry.io/docs/concepts/semantic-conventions/
41. OpenTelemetry semantic conventions 1.37.0, otwierano: września 22, 2025, https://opentelemetry.io/docs/specs/semconv/
42. Semantic conventions for generative AI systems | OpenTelemetry, otwierano: września 22, 2025, https://opentelemetry.io/docs/specs/semconv/gen-ai/
43. GenAI Semantic Conventions - traceloop, otwierano: września 22, 2025, https://www.traceloop.com/docs/openllmetry/contributing/semantic-conventions
44. 7 Tools and Best Practices to Observe AI Agents in Production (2025) | by Kuldeep Paul, otwierano: września 22, 2025, https://medium.com/@kuldeep.paul08/7-tools-and-best-practices-to-observe-ai-agents-in-production-2025-aa505d474fb3
45. What are the 9 Best LLM Observability Tools Currently on the Market ..., otwierano: września 22, 2025, https://www.zenml.io/blog/best-llm-observability-tools
46. LLM Observability Tools: 2025 Comparison - lakeFS, otwierano: września 22, 2025, https://lakefs.io/blog/llm-observability-tools/
47. Which monitoring and evaluation platforms work best for LLM systems?, otwierano: września 22, 2025, https://community.latenode.com/t/which-monitoring-and-evaluation-platforms-work-best-for-llm-systems/37458
48. OpenTelemetry - Datadog, otwierano: września 22, 2025, https://www.datadoghq.com/solutions/opentelemetry/
49. Fiddler AI: AI Observability, Model Monitoring, LLM Monitoring, and Agentic Observability, otwierano: września 22, 2025, https://www.fiddler.ai/
50. Agent Observability - Maxim AI, otwierano: września 22, 2025, https://www.getmaxim.ai/products/agent-observability
51. Top 5 Tools to Monitor AI Agents in 2025 - Maxim AI, otwierano: września 22, 2025, https://www.getmaxim.ai/articles/top-5-tools-to-monitor-ai-agents-in-2025/

52. Forwarding via Data Connectors - Maxim Docs, otwierano: września 22, 2025, https://www.getmaxim.ai/docs/tracing/opentelemetry/forwarding-via-data-connectors
53. OpenLIT | OpenTelemetry-native GenAI and LLM Application Observability, otwierano: września 22, 2025, https://openlit.io/
54. Portkey | 🦜️ LangChain, otwierano: września 22, 2025, https://python.langchain.com/docs/integrations/providers/portkey/
55. LangSmith - LangChain, otwierano: września 22, 2025, https://www.langchain.com/langsmith
56. Structured outputs - LangChain, otwierano: września 22, 2025, https://python.langchain.com/docs/concepts/structured_outputs/
57. How to parse JSON output - LangChain, otwierano: września 22, 2025, https://python.langchain.com/docs/how_to/output_parser_json/
58. How to map data from tracing project runs to dataset - LangSmith ..., otwierano: września 22, 2025, https://forum.langchain.com/t/how-to-map-data-from-tracing-project-runs-to-dataset/1568