Chapter 10. Monitoring in Production

Whether you're a product owner, machine learning (ML) engineer, or site reliability engineer (SRE), once agents hit production, you need to see what they're doing and why. Shipping agentic systems is only the halfway point. The real challenge begins once your agents are operating in dynamic, unpredictable, high-stakes environments. Monitoring is how you learn from reality—how you catch failures before they escalate, identify regressions before users notice, and adapt systems in response to real-world signals.

Unlike traditional software, agents behave probabilistically. They depend on foundation models, chain together tools, and respond to unbounded user inputs. You can't write exhaustive tests for every scenario. That's why monitoring becomes the nervous system of your deployed agent infrastructure.

Monitoring isn't just about detecting problems. It's the backbone of a tight feedback loop that accelerates learning and iteration. Teams that monitor well learn faster, ship safer, and improve reliability with every deployment.

In this chapter, we focus on open source monitoring. While there are excellent commercial platforms like Arize AX, Langfuse, and WhyLabs, we'll concentrate here on tooling you can self-host and extend freely. Our reference stack includes:

OpenTelemetry

: For instrumenting agent workflows

Loki

: For log aggregation and search

Tempo

: For distributed traces

Grafana

: For visualization, alerts, and dashboards

We'll walk through how to integrate each of these with a LangGraph-based agent system, then show how the pieces fit together into a feedback loop that closes the gap between observation and improvement.

Monitoring Is How You Learn

Understanding root causes of agent failures—from software bugs and foundation model variations to architectural limits—is essential for proactive maintenance and system adaptability. Each type demands targeted detection, analysis, and fixes to maintain stability in production.

The best agent systems improve over time through feedback. Traditional monitoring reacts to crashes or throughput dips, but for agents, it's foundational: revealing emergent issues in probabilistic behaviors and guiding development amid uncertainty.

Agent failures are subtle—a tool succeeds but cascades errors, an LLM output sounds fluent yet misleads, or a plan partially works but misses the goal. These mismatches rarely crash systems; monitoring must expose them swiftly, making production observability nonoptional.

Failures aren't just incidents—they're test cases. Every time an agent breaks in production, that scenario should be captured and turned into a regression test. But the same is true for success: when an agent handles a complex case well, that trace can become a golden path worth preserving. By exporting both failure traces and exemplar successes into your test suite, you create a living CI/CD corpus that reflects real-world conditions. This practice helps "shift left" your monitoring strategy—catching issues earlier in development, and ensuring that new agent versions are continuously validated against the actual complexity of production behavior.

A key challenge in monitoring probabilistic systems like agents is distinguishing true "failures" (systematic issues requiring fixes) from expected variations (inherent nondeterminism where outputs differ but stay acceptable). A simple decision tree can guide this. Start with the output—does it meet success criteria (e.g., eval score > 0.8)? If yes, monitor trends but no action is needed. If no, check reproducibility (rerun 3–5 times; failure rate > 80% indicates systematic bug for engineering review). If it is not reproducible, assess confidence/variance (e.g., LLM score > 0.7, Kullback-Leibler

divergence < 0.2 from baseline). Within the bounds means expected variation (log for drift watch), and outside the bounds suggests anomalous failure (e.g., input drift via population stability index > 0.1, triggering mitigation like retraining or guardrails). This flowchart, applied in tools like Grafana, prevents overreaction to noise while catching real degradations early.

Effective monitoring spans infrastructure signals (latency, error rates, CPU) and semantic behaviors (intent grasp, tool selection, hallucination, task abandonment). Was the user's intent understood? Was the right tool selected? Did the system produce hallucinated content? Did the user abandon the task halfway through? These are not questions traditional monitoring systems are built to answer, but they are critical to ensuring agents remain trustworthy, helpful, and aligned.

Build a layered feedback loop: instrument runtime events (tool calls, generations, fallbacks) with context, streaming to backends like Loki (logs), Tempo (traces), and Grafana (visualization/alerting). Append evaluation signals—hallucination scores or drift indicators—via external critics in real time.

It's worth emphasizing that all of this can—and should—be part of the same observability pipeline used for production services. The same Prometheus instance that tracks service health can also track agent success rates. The same Grafana dashboards used by SREs can include semantic error rates, model latency distributions, and tool usage graphs. There is no need for a separate monitoring stack; agents benefit from the same rigor and visibility as any other critical software service.

Of course, observability data often contains sensitive content. Logs may include user messages, tool inputs, or intermediate LLM generations. To maintain compliance and user privacy, teams should configure separate monitoring clusters with strict role-based access control (RBAC). Sensitive data can be routed to isolated backends with encryption-at-rest and access auditing, ensuring that debugging and performance analysis remain possible without compromising trust or compliance obligations. It's also common practice to redact, hash, or mask personally identifiable information (PII) from observability logs before export. OpenTelemetry provides hooks for data scrubbing during span export, enabling fine-grained control over what leaves the boundary of the application.

Ultimately, monitoring turns metrics into action—helping teams spot what's critical and respond fast. The following sections show how open source tools build this loop, accelerating development, robustness, and reliability in live environments.

Before diving into instrumentation details, it's helpful to define what you actually want to observe. Effective monitoring begins with choosing the right metrics—those that reveal not just whether the system is up, but whether it's working as intended.

Table 10-1 is a practical taxonomy of metrics, organized by layer of abstraction, that can guide what to collect, visualize, and alert on.

Table 10-1. Taxonomy of metrics

	Metric	Purpose	Example action
Infrastructure	CPU/memory usage	Monitor system health and scaling pressure	Autoscale or optimize memory-intensive tools
	Uptime/availability	Track service availability and failure recovery	Trigger incident response
	Request latency (P50, P95, P99)	Ensure responsiveness under load	Engage in tune caching or retry logic
Workflow level	Task success rate	Determine how often agents complete intended workflows	Investigate failures or update prompts
	Token usage	Measure the token consumption at the workflow level	Rapid increases or decreases can indicate issues
	Tool call success/failure rate	Detect degraded integrations or misuse of tools	Patch wrappers or fall back automatically
	Tool use rate limit exceeded	Track instances where agent tool invocations surpass predefined call limits within specified time windows	Adjust limits or adjust invocation frequency
	Retry frequency	Identify instability or flakiness in plans or tools	Debounce retries or refine planning logic
	Fallback frequency	Surface failures in primary workflows	Improve robustness or escalate to human

	Metric	Purpose	Example action
Output quality	Token usage (input/output)	Track verbosity, cost, and generation efficiency	Prune long prompts or switch model tier
	Hallucination indicator	Measure semantic accuracy of generated content	Introduce grounding or LLM critique steps
	Embedding drift from baseline	Detect distribution shifts in user inputs or task framing	Adjust workflows or fine-tune model
User feedback	Requery/rephrasing rate	Measure whether users are understood on first try	Improve intent classification
	Task abandonment rate	Identify workflows that confuse or frustrate users	Simplify flows or add clarification prompts
	Explicit ratings (thumbs up/down)	Collect qualitative assessments of system helpfulness	Use it to triage outputs for evaluation

Each of these metrics can be logged via OpenTelemetry, aggregated in Prometheus or Loki, visualized in Grafana, and (where appropriate) linked to traces in Tempo. The goal is not to collect everything, but to collect what is necessary to detect meaningful change—and to do so in a way that supports rapid diagnosis and continuous improvement.

Monitoring Stacks

Selecting the right monitoring is an important decision that can either accelerate or impede the pace of development for your agent system. Observability must capture not only traditional infrastructure metrics (e.g., latency, uptime) but also semantic insights like hallucination rates, tool

efficacy, and distribution shifts in user inputs. The current landscape emphasizes open source tools that integrate seamlessly with frameworks like LangGraph, CrewAI, and AutoGen, supporting distributed tracing, logging, and alerting while handling the probabilistic nature of foundation models. Many companies already have established enterprise plans for managed logging stacks (e.g., Splunk, Datadog, or New Relic), and foundation models or agents don't necessarily require an entirely new monitoring solution. In most cases, it's wise to extend your existing stack—leveraging its familiarity, scalability, and integrations—unless you have strong needs for specialized features like evaluations that are native to foundation models or lightweight self-hosting. We'll explore several equivalent open source options in the following subsections, highlighting features, integrations, and trade-offs to help you choose or adapt based on your environment.

Grafana with OpenTelemetry, Loki, and Tempo

This stack offers high composability, making it a flexible choice for teams building custom observability around agents:

Setup and integration

Initialize OpenTelemetry (OTel) in your LangGraph application to export spans (e.g., for tool calls or LLM generations) and metrics (e.g., token usage). The logs route to Loki for structured querying, while traces go to Tempo for end-to-end visibility. Grafana pulls from both, establishing dashboards that correlate agent behavior (e.g., planning latency) with system health. Example: wrap a LangGraph node with OTel spans to track `tool_recall` metrics, exporting to Tempo for querying failed sessions.

Key features

The key features are the real-time dashboards for semantic metrics (e.g., hallucination scores via custom plug-ins); alerting on anomalies like retry spikes; and strong community for AI extensions (e.g., 2025 Grafana plug-ins for LLM drift detection). It's scalable for production, with low overhead when self-hosted.

Trade-offs

Pros include flexibility (mix-and-match components) and no vendor lock-in; cons are the multitool setup (requires managing Loki/Tempo

separately) and a steeper learning curve for noninfra teams. This is ideal for enterprises extending existing infra monitoring to agents.

ELK Stack (Elasticsearch, Logstash/Fluentd, Kibana)

The ELK Stack is a mature option emphasizing powerful search and analytics, often extended from existing enterprise setups for AI workloads:

Setup and integration

Use OTel collectors to send agent traces/logs to Elasticsearch (via Logstash for ingestion). Kibana provides the UI for querying and dashboards. For LangGraph, instrument nodes to log structured events (e.g., JSON with tool params), leveraging Elasticsearch's ML jobs for anomaly detection on agent outputs. Example: query "hallucination events where confidence < 0.7" across sessions, correlating with user feedback.

Key features

Key features include advanced full-text and vector search for LLM outputs (e.g., embedding-based drift detection); built-in ML for predictive alerts (e.g., forecasting tool failure rates); and scalability for massive log volumes with clustering.

Trade-offs

The pros are superior search and analytics (e.g., fuzzy matching on prompts, better for long-tail failures) and enterprise-grade scalability. The cons are higher resource demands (Elasticsearch is memory-intensive) and a more complex deployment (multiple services). It is best for teams with existing ELK investments, extending it for agent-specific semantic logging without starting from scratch.

Arize Phoenix

Phoenix focuses on LLM tracing and evaluation, providing a debug-oriented extension for agent monitoring in existing environments:

Setup and integration

Use Phoenix's Python SDK to instrument LangGraph (e.g., trace LLM calls with evals). It supports OTel export for hybrid use. Example: visualize agent traces with auto-scorers for accuracy, exporting to notebooks for analysis.

Key features

Key features include structured tracing with evals (e.g., RAG quality, hallucination detection); Jupyter integration for ML workflows; and 2025 enhancements for multiagent coordination metrics.

Trade-offs

The pros are it is specialized for evals/debugging (faster insights on agent quality) and lightweight for prototyping. The con is that it is limited to traces/evals (supplement for full logs/metrics) and more dev-oriented than ops. It is great for research/ML teams adding agent insights to managed enterprise stacks.

SigNoz

SigNoz is a unified, OTel-native platform that combines metrics, traces, and logs in a single tool, suitable for streamlined extensions of basic monitoring setups:

Setup and integration

SigNoz ingests OTel data directly, with auto-instrumentation for Python (e.g., LangGraph). Add spans for agent steps (e.g., planning latency) and query via its UI. Example: trace a multistep agent flow, filtering by `token_usage > 1000` to spot inefficiencies, with built-in evals for LLM quality.

Key features

It has integrated AI-powered insights (e.g., anomaly detection on agent traces); custom dashboards for LLM metrics (e.g., prompt drift); and lightweight self-hosting with ClickHouse backend for efficiency.

Trade-offs

The pros include a simpler setup (single app), lower overhead for small teams, and strong OTel support with AI extensions (e.g., 2025

updates for hallucination auto-scoring). The cons are that it has a less extensible ecosystem (fewer plug-ins) and that visualization is functional but not as advanced. It is well suited for startups or ML-focused teams extending lightweight monitoring without heavy infra additions.

Langfuse

Langfuse specializes in foundation model and agent observability, making it easy to extend existing stacks with semantic-focused tracing for agents:

Setup and integration

Integrate via SDK in LangGraph (e.g., wrap nodes with Langfuse tracers). It captures prompts, outputs, and evals (e.g., custom scorers for coherence). Example: log a full agent session, auto-evaluate for hallucination, and export traces for regression testing.

Key features

It has LLM-native metrics (e.g., token cost tracking, A/B testing for prompts); session replay for debugging; and it is self-hostable with database backends like PostgreSQL.

Trade-offs

The pros are that it is tailored for agents/LLMs (as built-in evals save custom work) and easy for dev teams (focus on app-level insights). The cons are that it has a narrower scope (weaker on infra metrics like CPU; pair with Prometheus) and it is less scalable for non-LLM telemetry. It is ideal for extending enterprise logging with agent-specific features without overhauling the core stack.

Choosing the Right Stack

All these open source stacks are viable equivalents—start by assessing your current setup. If you have an enterprise-managed solution, extend it with OTel instrumentation for agent signals unless you need LLM-specific features like auto-evals (favor Langfuse/Phoenix) or advanced search (ELK). For greenfield projects, Grafana or SigNoz offer broad coverage. Evaluate based on team expertise, data volume, and integration needs—many can hybridize

(e.g., OTel to multiple backends). Table 10-2 shows these trade-offs at a glance.

Table 10-2. Comparison of monitoring and observability stacks

Stack	Key strength	Best for	Trade-off (versus Grafana)
Grafana + Loki/Tempo	Composability and visualization	Enterprise ops	More components to manage
ELK Stack	Advanced search/analytics	Large-scale logs	Higher resource use
Phoenix	Tracing and debugging	Dev iteration	Limited production scale
SigNoz	Unified and lightweight	Startups/ML teams	Less extensible
Langfuse	Foundation model/agent-specific evals	Semantic monitoring	Narrower infra coverage

While the observability landscape offers multiple strong options—each with unique strengths in scalability, ease of use, or LLM-specific features—this competition drives innovation and ensures teams can find a fit for their needs, whether extending enterprise stacks or starting fresh. For our examples in the following sections, we'll focus on OTel with Grafana, Loki, and Tempo, as it provides a highly composable, open source foundation that's widely adopted and integrates seamlessly with agent frameworks like LangGraph, enabling us to demonstrate core concepts without vendor lock-in. With a stack selected, the next step is instrumentation—embedding telemetry directly into your agent runtime to capture meaningful signals, as explored in the next section.

OTel Instrumentation

The first step in building an effective monitoring loop is instrumentation. Without high-quality signals embedded directly into your agent runtime, you're flying blind. OTel provides the foundation for structured, interoperable

telemetry across traces, metrics, and logs—and it integrates well with LangGraph-based agent systems.

LangGraph is structured as a graph of asynchronous function calls. Each node in the graph represents a functional step in an agent workflow—perhaps planning, calling a tool, or generating a response with an LLM. Because each step is already isolated and explicitly declared, it's straightforward to instrument each one with OTel spans. These spans create a structured trace that records not just when steps started and ended, but what they were trying to do and how they performed.

For each node, we recommend starting a span at the beginning of the function and annotating it with relevant metadata. For example, in a tool-calling node, you might capture the tool name, the specific method called, the latency of the response, the success or failure status, and any known error codes. In nodes where the LLM generates output, the span can include prompt identifiers, token counts, model latency, and flags for hallucination risk or confidence scores.

This instrumentation does not require major architectural changes. OTel's Python SDK can be initialized once at startup, and spans can be created and closed using simple context managers. The distributed tracing context is automatically propagated across async calls, making it easy to correlate end-to-end behavior even in complex, branched agent flows. Here is a simplified example of how to wrap a LangGraph node with a trace span:

```
from opentelemetry import trace
tracer = trace.get_tracer("agent")

async def call_tool_node(context):
    with tracer.start_as_current_span("call_tool", attr
        "tool": context.tool_name,
        "input_tokens": context.token_usage.input,
        "output_tokens": context.token_usage.output,
    }):
        result = await call_tool(context)
        return result
```

Spans can include events (like fallback triggers or retries), nested subspans (to measure downstream API calls), and exception capture for automatic error tagging. These traces are exported in real time to backends like Tempo or Jaeger and visualized alongside logs and metrics in Grafana.

In addition to traces, OTel can emit structured logs and runtime metrics. For example, you can record the number of times a specific tool is invoked, the average response time per planning node, or the percentage of failed tasks per model version. These metrics are invaluable for creating dashboards and alerts that track long-term performance and detect early signs of degradation.

Instrumentation must be thoughtfully scoped. Too much detail becomes noisy; too little makes root cause analysis difficult. The key is to attach just enough context at each step—user request IDs, session metadata, agent configuration state, skill names, and evaluation signals—so that when something goes wrong, the trail of evidence is coherent, complete, and easily searchable.

Tempo acts as the trace backend. Every span you instrument in LangGraph—each tool call, plan generation, or fallback—is part of a distributed trace. Tempo stores these traces in a highly scalable fashion and supports deep querying. For instance, you can filter all traces where the planning step took longer than 1.5 seconds, or where a particular tool call failed with a given error code. This enables precise debugging of subtle issues that emerge only under real-world, multistep execution conditions.

Loki, by contrast, serves as your log aggregation layer. It captures structured logs—often in JSON format—from across your agent infrastructure. Each LangGraph node can emit structured log events during its execution: when a user query is received, when a tool is invoked, when an LLM produces an ambiguous response, or when a fallback path is triggered. Logs can be annotated with span and trace IDs, making it easy to correlate logs and traces from the same user session or agent workflow. While Loki is a great fit for structured logs, teams requiring full-text search, role-based views, or higher ingestion throughput may also consider Elasticsearch or commercial options like Datadog logs or Honeycomb.

Grafana unifies both of these data streams into a single pane of glass. It provides the visualization layer where logs from Loki and traces from Tempo can be explored side by side. Within Grafana, you can construct dashboards that show live trace data, drill down into individual requests, and correlate

structured logs with performance metrics. You can also build custom alerting rules—for example, flagging when error rates for a particular agent spike above a threshold, or when tool response latency crosses a defined boundary.

Together, OTel, Tempo, Loki, and Grafana form a complete, open source observability stack for agent systems. They enable deep inspection of behavior, fast root cause analysis, historical trend evaluation, and proactive anomaly detection. This integration is what transforms raw telemetry into operational intelligence—and operational intelligence into a development accelerant. This observability enables real-time debugging, trend analysis, and continuous learning—all of which are essential to the safe and scalable deployment of intelligent agents in production.

Visualization and Alerting

Once your LangGraph agents are instrumented with OTel and streaming logs and traces into Loki and Tempo, the final and most impactful layer is visualization and alerting—made possible with Grafana. Grafana is more than just a dashboarding tool; it's the operational frontend for observability, where signals become stories and metrics become actions.

Grafana connects seamlessly with Loki and Tempo as native data sources. For traces, Grafana's Tempo integration enables you to browse full execution traces for individual agent runs. This includes viewing span hierarchies that represent the sequence and timing of steps an agent took—from receiving a user query to selecting a plan, calling tools, and composing the final output. You can filter traces by latency, status, span name, or any custom attribute you've attached in your LangGraph nodes. This is invaluable for debugging multistep agent behaviors, especially when performance degrades or edge-case bugs arise.

For logs, the Loki plug-in enables querying structured log events emitted during agent execution. Grafana's log panels enable you to visualize real-time logs across all agents; filter by agent name, user session, error type, or trace ID; and correlate logs with related traces. Because logs and traces share common metadata—such as request or session IDs—Grafana lets you jump directly from a spike in log volume or error messages to the exact trace that triggered them.

But Grafana's true power lies in building dashboards tailored to your agents' semantics and success criteria. As illustrated in Figure 10-1, a GenAI Observability dashboard can display key metrics like request rates, usage costs, token consumption, and request distributions for foundation models and vector databases. For example, you might build a dashboard showing the following:

- Token usage per agent per hour (to detect model verbosity regressions)
- P95 latency for tool calls and planning nodes
- Task success rate by workflow or prompt template version
- Fallback frequency by tool or skill
- Drift indicators based on embedding similarity of user queries over time

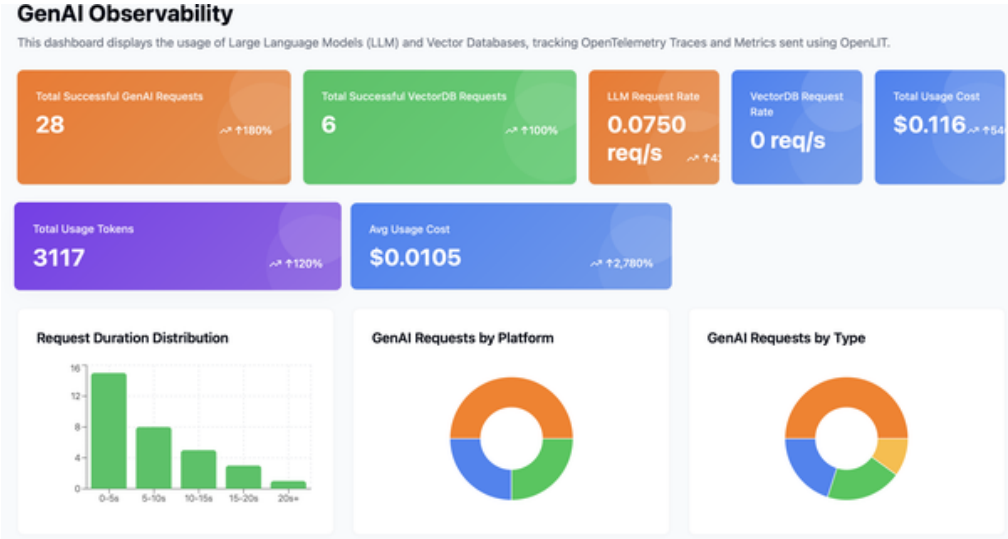

Figure 10-1. Grafana for AI Observability. This dashboard visualizes key metrics for foundation model and vector database usage, including request rates, success counts, costs, token consumption, request durations, top models by usage, and breakdowns by platform, type, and environment, providing actionable insights into agent performance and efficiency.

Each of these panels not only helps visualize system performance but also guides ongoing development. If a particular tool starts failing more often, or if token usage increases unexpectedly, these signals help prioritize debugging and optimization.

Grafana also supports custom alerts. You can define thresholds on any metric and trigger alerts via Slack, email, PagerDuty, or any other integration. For example, you might trigger alerts in the following circumstances:

- Hallucination rates exceed 5% in the last 30 minutes
- Retry loops occur more than three times in a single session
- Average response time for a critical tool increases by more than 50%

Alerts ensure your team is aware of regressions and anomalies in real time, even if no one is actively watching the dashboard. Combined with Loki logs and Tempo traces, these alerts help close the feedback loop rapidly.

Grafana's alerting system is highly extensible, integrating seamlessly with popular incident management tools like PagerDuty for escalating notifications to on-call teams—ensuring that high-severity issues, such as sudden spikes in hallucination rates or task failures, trigger structured response workflows with automated paging and acknowledgment. For more specialized error monitoring, Sentry can be layered in to capture and analyze exceptions within agent code, providing stack traces, breadcrumbs, and release health metrics that complement Grafana's dashboards; this is particularly useful for debugging probabilistic bugs in foundation model calls or tool invocations, with Sentry's SDK easily instrumented alongside OTel.

For teams seeking an all-in-one solution tailored to agentic systems, platforms like AgentOps.ai offer a streamlined alternative, combining tracing, metrics, evaluations, and alerting in a single package optimized for foundation models and agents. AgentOps.ai handles semantic monitoring (e.g., auto-scoring outputs for quality) and integrates with existing stacks, reducing setup overhead compared with composing Grafana components—though it may introduce vendor dependency. These options create flexibility: extend Grafana with PagerDuty/Sentry for robust alerting, or adopt AgentOps.ai for faster agent-specific insights, depending on your operational maturity and focus.

By integrating Grafana deeply into your agent development lifecycle, you create a living interface to your deployed systems. It becomes the shared canvas where product teams, engineers, and reliability staff can observe, debug, iterate, and improve. In the world of agent-based systems—where bugs are probabilistic and failure modes are emergent—this kind of unified visibility isn't just nice to have. It's essential.

Monitoring Patterns

Once an observability stack is in place—spanning instrumentation, logs, traces, dashboards, and alerts—the question becomes: how do we safely ship changes to agentic systems that are inherently probabilistic, adaptive, and hard to predict fully? The answer lies in adopting monitoring-aware development patterns that de-risk experimentation and create safety nets around production

changes. In this section, we explore several key patterns that teams can adopt to ensure their agents continue to evolve safely and responsively.

Shadow Mode

In shadow mode, a new or experimental version of an agent runs alongside the current production agent, processing the same inputs but without serving its outputs to users. This enables developers to log and trace the behavior of the new agent in real-world conditions without affecting user experience.

With OTel, you can instrument both the production and shadow agents and attach a shared request ID. Logs and traces from the shadow agent can then be labeled accordingly in Loki and Tempo, making it easy to compare behavior. You might look at differences in tool selection, latency, token usage, or hallucination frequency. These comparisons are especially useful when trialing new model versions, planning strategies, or prompting techniques.

Shadow mode enables safer innovation. It enables teams to answer: does the new agent do better or worse on live traffic? What breaks? What improves? And it lets you collect this data continuously, in parallel with normal operation.

Canary Deployments

Where shadow mode gathers information without exposure, canarying goes one step further. A canary deployment serves a new agent version to a small subset of real users—say, 1% or 5% of traffic—while the majority of users continue to interact with the baseline version.

Grafana dashboards are critical in this setup. By filtering all metrics and traces by version tag, you can directly compare success rates, latency, tool usage, and error counts between canary and baseline agents. Alerts can be configured to trigger if the canary shows significant regressions or anomalies.

If the canary behaves well, the deployment can be gradually expanded. If not, it can be rolled back immediately with minimal user impact. Canarying provides the operational safety needed to iterate quickly in production environments.

Regression Trace Collection

Every time an agent fails in production—whether through hallucination, planning error, or tool misuse—it creates an opportunity for learning. By automatically exporting these failure traces (from Tempo) or log snapshots (from Loki) into your test suite, you build a continuously updated regression corpus.

This turns production failures into training signals. A failed tool call or misaligned output becomes a new test case. Once a fix is implemented, rerunning this trace should pass. Over time, this strategy strengthens your evaluation set with real-world edge cases and helps prevent recurrence of the same failure modes.

Self-Healing Agents

Finally, monitoring can do more than detect failure—it can help agents recover from it. Agents that are designed to read their own telemetry in real time can implement fallback mechanisms when issues are detected.

For example, if a tool call fails repeatedly, the agent might reroute to a simpler fallback plan or ask the user for clarification. If latency spikes, the agent could skip optional reasoning steps. If hallucination scores are high, it could issue a disclaimer or defer to human review.

These self-healing behaviors are most effective when supported by detailed monitoring data. Each fallback decision can be logged and traced, enabling teams to analyze when and why fallbacks were triggered, and whether they helped resolve the issue.

User Feedback as an Observability Signal

While much of this chapter has focused on logs, traces, and metrics, user feedback offers a complementary lens—direct insight into how well the agent is meeting human expectations. Feedback can be implicit, such as users rephrasing their inputs, abandoning tasks, or hesitating during interactions. It can also be explicit, like a thumbs-down icon, a star rating, or a free-text comment. Both forms provide real-time signals that can and should be integrated into your monitoring stack.

In practice, implicit feedback metrics—such as task abandonment rate or requery frequency—can be logged and aggregated in Loki and visualized in Grafana just like any other performance metric. They offer early indicators of friction or confusion. Explicit feedback events, like low ratings, can be tied to specific traces in Tempo and trigger alerts when dissatisfaction spikes. Dashboards that combine user sentiment metrics with trace-based technical data enable teams to correlate performance issues with user frustration, giving a fuller picture of agent health.

Critically, user feedback can also drive improvement loops. For example, traces associated with low user ratings can be exported directly to the evaluation set for post hoc review. If multiple users abandon a specific flow, it may warrant revisiting the planning strategy or retraining the foundation model prompt. By integrating user signals into the broader observability and action framework, teams ensure their monitoring practices remain not only operationally effective but also user-centered.

Distribution Shifts

One of the subtler, yet most critical, challenges in monitoring agent-based systems is identifying and managing distribution shifts. These occur when the statistical properties of the agent's environment change over time—whether through evolving user language, new product terminology, changes in API responses, or even updates to the foundation model itself. While such shifts may not trigger explicit errors, they often manifest as degraded performance, misaligned outputs, or increased fallback usage.

Monitoring systems are your first line of defense against this kind of slow drift. Dashboards that track task success rates, tool invocation failures, and semantic metrics—such as token usage trends or hallucination frequency—can surface early signals. For quantitative detection, employ statistical tests like the Kolmogorov-Smirnov (KS) test to compare distributions of input features or outputs. The KS test is a nonparametric statistical test that compares the empirical cumulative distribution functions of two datasets to determine if they are drawn from the same underlying distribution, making it ideal for detecting shifts in continuous features like query lengths, latencies, or numerical metrics without assuming normality. It calculates the maximum vertical distance (KS statistic) between the distributions, along with a p-value for statistical significance; thresholds like KS > 0.1 (often paired with p-value

< 0.05) indicate meaningful divergence, triggering alerts for potential drift in agent inputs or outputs. In this code, SciPy's `ks_2samp` function is applied to sample arrays of historical and current data, printing a detection message if the statistic exceeds the threshold. Here's a small Python example using SciPy to detect drift in query lengths:

```
import numpy as np
from scipy import stats

# Historical and current query lengths (e.g., character
historical = np.array([10, 15, 20, 12])  # Baseline dat
current = np.array([25, 30, 28, 35])     # New data

ks_stat, p_value = stats.ks_2samp(historical, current)
if ks_stat > 0.1:
    print(f"Drift detected: KS statistic = {ks_stat}")
```

Kullback-Leibler (KL) divergence measures how one probability distribution diverges from another, often used to detect concept drift by quantifying shifts in token distributions (e.g., changes in word frequencies that might indicate evolving user language or new terminology). It is not symmetric KL(P||Q) ≠ KL(Q||P) and can signal when current data (Q) deviates significantly from historical baselines (P), with higher values indicating greater drift—e.g., a threshold > 0.5 might flag concept changes in embeddings. In this code, we normalize frequency vectors to probabilities, add a small epsilon to avoid `log(0)` errors, and compute the sum of *P * log(P/Q)*; the example assumes simplified token count arrays for historical and current data:

```
import numpy as np

def kl_divergence(p, q, epsilon=1e-10):
    p = p + epsilon
    q = q + epsilon
    p = p / np.sum(p)
    q = q / np.sum(q)
    return np.sum(p * np.log(p / q))

# Token frequency vectors (e.g., [word1, word2, ...] cc
historical_tokens = np.array([0.4, 0.3, 0.3])
current_tokens = np.array([0.2, 0.5, 0.3])
```

```
kl = kl_divergence(historical_tokens, current_tokens)
if kl > 0.5:
    print(f"Concept drift detected: KL = {kl}")
```

The population stability index (PSI) is a metric for detecting shifts in categorical or binned continuous variables (e.g., tool usage categories like “refund,” “cancel,” “modify”) by comparing percentage distributions between historical and current datasets, often divided into buckets for granular analysis. It sums natural_logarithm(actual_percent / expected_percent) across categories, where low PSI (< 0.1) means stability, 0.1–0.25 indicates minor drift (monitor), and > 0.25 signals major drift (intervene—e.g., retrain). This helps flag changes in patterns without assuming normality, making it suitable for agent metrics like invocation frequencies:

```
import numpy as np

def psi(expected, actual):
    expected_percents = expected / np.sum(expected)
    actual_percents = actual / np.sum(actual)
    psi_values = ((actual_percents - expected_percents)
        np.log(actual_percents / expected_percents))
    return np.sum(psi_values)

# Tool usage counts (e.g., ['refund', 'cancel', 'modify
historical = np.array([50, 30, 20])
current = np.array([20, 50, 30])

psi_value = psi(historical, current)
if psi_value > 0.25:
    print(f"Major drift: PSI = {psi_value}")
elif psi_value > 0.1:
    print(f"Minor drift: PSI = {psi_value}")
```

Sudden drops in accuracy (e.g., > 5–10% over a rolling 24-hour window), increases in task abandonment (> 15%), or surges in retries (> 20% session rate) are all potential indicators of input or concept drift. Embedding-based techniques, such as computing cosine similarity between current and historical query vectors, can also be used to compare new inputs against baselines (e.g., mean similarity < 0.8 triggers review), often implemented via libraries like Evidently AI for automated alerting in Grafana.

Responding to these shifts is part of building resilient systems. Transient changes may be addressed by tuning thresholds or updating parsing logic, while persistent shifts might require retraining workflows or adapting to new APIs. Feedback loops, such as logging and exporting degraded traces for analysis, help teams determine whether issues are temporary or systemic. As always, response strategies benefit from the real-time visibility provided by a strong observability stack—enabling teams to act before drift becomes failure.
Responding to these shifts is part of building resilient systems. Transient changes may be addressed by tuning thresholds or updating parsing logic, while persistent shifts might require retraining workflows or adapting to new APIs—guided by drift severity from the statistical measures (e.g., prioritize retraining if PSI > 0.25 persists over 48 hours). Feedback loops, such as logging and exporting degraded traces for analysis, help teams determine whether issues are temporary or systemic—perhaps via A/B testing post-detection to validate fixes. As always, response strategies benefit from the real-time visibility provided by a strong observability stack—enabling teams to act before drift becomes failure.

Metric Ownership and Cross-Functional Governance

As teams deploy agent-based systems, a subtle but serious organizational challenge emerges: who owns which metrics? In traditional software stacks, there's a clear split: infrastructure teams own latency and uptime, product teams own conversion or user success, and ML teams (if present) build models, and manage the health and performance of them, with responsibility for both the engineering and product implications. But agents powered by foundation models don't respect these boundaries—and neither should your monitoring strategy.

A foundation model response isn't just a model artifact—it's the product. A long chain of tool calls, retries, fallbacks, and generation steps isn't a backend quirk—it's the user experience. And a five-second plan generation delay isn't a model limitation—it's often a prompt or workflow design decision that someone made on the product team.

That's why logs, traces, and evaluation signals from agents belong in the core observability platform, alongside service health and system metrics. If product

dashboards and model notebooks are the only place that agent metrics show up, you're missing the full picture—and likely masking systemic issues.

Latency is a perfect example. Teams often adopt the mindset that "foundation models are slow," and then inadvertently build latency into everything—from verbose prompts to unnecessary retries to bloated plans. Without rigorous, trace-based instrumentation, this drift goes undetected. Before long, the whole system feels sluggish—not because the infrastructure is underpowered, but because the product and ML teams normalized delay as inevitable.

The solution isn't to offload latency ownership to infra or UX to product. It's to build shared dashboards where teams can do the following:

- Product leads can see how planning latency and fallback rate correlate with task abandonment.
- ML engineers can monitor hallucination rates and drift alongside user feedback.
- Infra/SRE teams can alert on token spikes and tool flakiness that affect system reliability.

Each team must own part of the agent telemetry—and no one team can interpret it in isolation. To address the organizational challenges of metric ownership, teams can use a Responsibility Assignment Matrix (RACI chart) to clarify roles across functions. In a RACI chart, each task or metric is assigned one or more of the following: R (Responsible: does the work), A (Accountable: owns the outcome), C (Consulted: provides input), or I (Informed: kept updated).

Table 10-3 is a template tailored to agent monitoring, which you can adapt based on your team's structure, size, and specific metrics. This promotes cross-functional collaboration by ensuring no metric falls through the cracks while avoiding silos.

Table 10-3. RACI matrix of monitoring metrics and cross-functional responsibilities

Metric/activity	Product team	ML engineers	Infra/SRE team
Latency (e.g., planning or tool call delays)	A (owns user impact) / C (consults on UX thresholds)	R (optimizes prompts/models) / I (informed on regressions)	R (monitors infra causes) / C (consults on scaling)
Hallucination rates	C (provides user feedback context) / I (informed on trends)	A/R (owns detection/mitigation via evals)	I (informed for alerting setup)
Task success rate	A (owns product goals) / R (defines success criteria)	C (consults on model improvements)	I (informed for system reliability ties)
Token usage/cost	C (consults on business impact)	R (optimizes generations) / I (informed on spikes)	A (owns budgeting/scaling) / R (monitors infra efficiency)
Distribution shifts (e.g., input drift)	I (informed for product adjustments)	A/R (detects via embeddings/evals)	C (consults on data pipeline stability)
Fallback/retry frequency	C (consults on UX fallbacks)	R (refines planning logic)	A (owns reliability) / I (informed on patterns)
User feedback/sentiment	A/R (owns aggregation and prioritization)	C (consults on model ties)	I (informed for ops alerts)

Metric/activity	Product team	ML engineers	Infra/SRE team
Dashboard maintenance and triage rituals	C (provides product context)	C (provides ML insights)	A/R (owns platform and cross-team reviews)

A trace that shows a tool being called four times in a loop, followed by a long generation, a vague response, and user abandonment—that's not just an engineering detail. That's a product failure. And it's only visible when logs and spans are routed through a shared platform like Loki and Tempo, not hidden in disconnected metrics tabs.

To make this work, use the following practice:

- Use shared observability dashboards with version tags and semantic metrics. Highly effective teams don't debate which dashboard is more accurate—they work across functional boundaries to improve the experience for customers together.
- Tag spans and logs with product context (feature flag, user tier, workflow ID).
- Create cross-functional triage rituals, where product, infra, and ML review telemetry together—especially after launches or major regressions.
- Avoid double standards: don't hold foundation model latency to a different bar than other services. Slowness that impacts users is everyone's problem.

Agentic systems demand cross-functional observability. The monitoring stack isn't just for detecting outages—it's the interface through which engineering, ML, and product learn to speak the same language about what the system is doing, how well it's performing, and where it needs to evolve.

Conclusion

Monitoring agent-based systems is more than a safety check—it is the discipline that enables intelligent systems to thrive in real-world environments. In this chapter, we've seen that monitoring is not just reactive;

it is how teams learn from production, adapt to change, and accelerate progress.

From foundational instrumentation with OpenTelemetry, to real-time log and trace collection via Loki and Tempo, to dashboards and alerts in Grafana, we outlined how to build an open source feedback loop that surfaces issues before they become outages—and turns every deployment into an opportunity for refinement.

We explored practical techniques like shadow mode, canarying, fallback logging, and user sentiment tracking. We emphasized not only what to measure but also how to act. And we showed how monitoring helps detect not just failures but slow drifts in context, data, or behavior that can quietly undermine performance if left unchecked.

The path forward is clear: teams that build agent systems with observability in mind—who instrument, visualize, and learn from their agents in flight—gain a powerful edge. They iterate faster. They trust their metrics. They recover gracefully when things go wrong.

In a world where agentic systems are becoming core infrastructure, robust monitoring isn't optional—it's foundational. And those who master it will lead the way in creating intelligent, resilient, and trustworthy agents at scale.