Monitoring Signals for AI Agent Reliability: Beyond the Golden Signals of Microservices

Amar Chaudhari

July 9, 2026
Working paper

Abstract

AI agents are deployed on microservice infrastructure and monitored with microservice tools—but their failure modes belong to neither category. Production incidents show agents producing confident, fluent outputs that are semantically wrong, committing irreversible side effects that standard dashboards record as successes, and degrading in ways that are invisible until user complaints arrive. The root cause is structural: microservice observability rests on four assumptions that agents systematically violate—determinism, explicit failures, bounded action space, and cheap stateless retries. We propose twelve production-observable monitoring signals in four families, adapted from the offline reliability metrics of Rabanser et al. [1] to operate without ground-truth labels: S1 (outcome signals), S2 (trajectory signals), S3 (resource-envelope signals), and S4 (boundary signals). An instrumented case study—672 episodes, four experimental conditions, `gpt-5.4-nano-2026-03-17` agent with `gpt-5.4-mini-2026-03-17` judge—shows an input-drift incident that leaves traditional signals entirely flat while the boundary family fires (unauthorized-irreversible fraction rises $0 \to 0.17$) and outcome consistency degrades ($0.905 \to 0.810$). The study also reveals that LLM-judge proxy quality itself degrades under infrastructure faults (agreement $0.798 \to 0.667$), motivating a "monitor the monitor" protocol of periodic human validation. The proposed signals complement, not replace, the golden signals: they instrument the semantic and behavioral layer that golden signals leave dark.

1 Introduction

AI agents are increasingly deployed on the same infrastructure that runs microservices. They run in containers, sit behind load balancers, and emit traces to the same application-performance monitoring (APM) stacks that operators have tuned for years. By extension, they are monitored like microservices: latency, error rate, throughput, and saturation dashboards light green, and operators conclude the system is healthy. We argue that this conclusion is structurally unsound for agentic systems, and that a new class of production-observable monitoring signals is needed to fill the gap.

The gap is not theoretical. Consider three incidents from recent deployments. In July 2025, an AI coding agent on a commercial platform deleted a user's production database despite receiving explicit instructions to leave it intact; the incident was discovered after the fact, not flagged by any operational monitor [2]. OpenAI's Operator agent, in a widely reported test, authorized a grocery purchase of \$31.43 without seeking user confirmation—a real-money side effect that completed successfully from the service's perspective [3]. New York City's business-advisory chatbot was found to advise small businesses to violate employment law and to give materially different answers to the same question posed on different days [4]. In each case the infrastructure metrics were

nominal: latency was low, error rates were near zero, throughput was steady. *The service was healthy; the agent was not.*

The common thread is that traditional monitoring signals rest on four assumptions that AI agents systematically violate. Microservice observability assumes that components behave *deterministically* (so retries and canary releases are meaningful), that failures are *explicit* (so they surface as status codes, exceptions, or timeouts that can be counted), that each service's action space is *bounded* (so worst-case blast radius can be reasoned about at design time), and that failed operations are *cheap and stateless to retry* (so retry-on-error is the universal remediation strategy). Table 1 enumerates these assumptions and the specific ways agents break each one. We return to them in detail in Section 2.

This paper is complementary to the offline reliability framework of Rabanser et al. [1], which establishes a rigorous science of AI agent reliability through ground-truth benchmarks, K repeated runs per task, and controlled perturbation experiments. Their framework asks: *how reliable is this agent on a defined task distribution?* We ask the orthogonal operational question: *which reliability properties are observable in production, where there are no ground-truth labels, no two runs are identical, and traffic is neither controlled nor repeatable?* Where possible, the signals we propose are streaming, label-free adaptations of the metrics Rabanser et al. [1] define offline; we additionally identify signals that matter only in operation—escalation rates, cost-envelope drift, and behavioral drift onset—which have no offline analogue because they depend on deployment context rather than a fixed task distribution.

Contributions. We make four contributions:

1. An assumption-by-assumption analysis of why golden-signal monitoring is structurally insufficient for AI agents, identifying four violated assumptions and their operational consequences (§2).
2. A suite of twelve production-observable monitoring signals in four families (outcome signals, trajectory signals, resource-envelope signals, and boundary signals), each with a measurement protocol (§3, Table 2).
3. An instrumented case study of a concrete agent incident that traditional infrastructure dashboards fail to detect, while the proposed signals provide early warning (§4).
4. A reference monitoring architecture showing how the proposed signals integrate with existing observability infrastructure (§5).

Scope. We propose signals, not remediation strategies; what to do upon detecting a reliability event is out of scope. The perturbations modeled in our case study represent natural operational faults—prompt variation, upstream data drift, user-phrasing shift—not adversarial attack; robustness to adversarial inputs is a distinct and important problem that we do not address. We also do not address alignment or value specification: we take the agent's intended behavior as given and ask only whether it is being achieved. These boundaries mirror those stated by Rabanser et al. [1] for their offline framework, and we adopt them for the same reason: tractability.

2 What Microservice Observability Assumes

2.1 The Golden Signals and Their Foundations

Site reliability engineering codified four "golden signals" as the canonical lens for production monitoring: latency, traffic, errors, and saturation [5]. Complementary frameworks refine the same idea: the RED method (Rate, Errors, Duration) focuses on request-serving systems; the USE method (Utilization, Saturation, Errors) focuses on resources. Together they define what an operator looks at first when something goes wrong. Service-level objectives (SLOs) formalize these signals into contractual thresholds—an error budget that quantifies how much unreliability a system is permitted to accumulate before remediation is required. Distributed tracing [6] adds causal structure: a trace links spans across service boundaries, letting operators follow a request as it propagates through a call graph and identify which component introduced latency or triggered an exception.

These mechanisms are mature, well-understood, and effective for the systems they were designed to instrument. We do not argue they are wrong. We argue that they are *insufficient* when the component under observation is an AI agent, because they were designed around four assumptions that agents systematically violate. We take each assumption in turn.

2.2 Assumption 1: Determinism

Microservice monitoring assumes that a given request produces a given response. This assumption underlies nearly every reliability practice: canary releases compare the new version's behavior against the old version's on the same traffic; load balancers retry failed requests on the expectation that a fresh attempt yields a correct result; error budgets treat repeated failures as evidence of a systematic problem, not random variation.

AI agents are stochastic. Even with temperature set to zero, language-model outputs are sensitive to prompt formatting, token ordering, and context length in ways that produce different intermediate reasoning steps and different final decisions on inputs that are nominally identical [7]. Rabanser et al. [1] document this empirically in their offline reliability framework: the same task prompt, submitted K times to the same agent, yields a distribution of outcomes, not a single reproducible result. In a monitoring context, this means that a retry does not test a different code path—it draws a new sample from the same distribution. An increase in answer variability is itself a reliability signal, but one that is invisible to error-rate dashboards that record only terminal failures.

2.3 Assumption 2: Explicit Failures

Microservice monitoring assumes that failures are legible to infrastructure. A failed database query raises an exception; an unreachable upstream returns a 5xx status code; a request that times out is counted in the error budget. The fundamental monitoring primitive—the error rate—presupposes that errors can be identified without reading the content of responses.

Agent failures are mostly *semantic*. The agent's HTTP layer returns 200; the response is fluent, grammatical, and confidently phrased; and it is wrong. The NYC chatbot incidents illustrate this precisely: the service returned successful completions that happened to advise businesses to violate labor law [4]. No exception was raised. No status code indicated failure. A semantic failure of this kind cannot be counted by an error-rate monitor because correctness is a property of the content, not the transport layer. Detecting it requires either ground-truth labels (unavailable in production) or a proxy signal that correlates with correctness without requiring it—which is precisely the problem our signal suite in Section 3 addresses.

2.4 Assumption 3: Bounded Action Space

Microservice monitoring assumes that the blast radius of a component is knowable at design time. A service that exposes three API endpoints can, by definition, only do three categories of things; an operator designing runbooks can enumerate the failure modes of each endpoint. Saturation metrics, circuit breakers, and capacity planning all rely on this enumerability.

A tool-using agent does not have a fixed action space. It composes sequences of tool calls—search, read, write, delete, send, purchase—at inference time, in response to context that neither the designer nor the operator has seen in advance. The sequence is chosen by the model; worst-case behavior is not derivable from the interface specification. The Replit incident demonstrates this concretely: the agent was not designed or deployed to delete databases, yet it reportedly did so by composing a sequence of individually permissible operations—a plausible reading consistent with the documented outcome, even though the internal reasoning is not publicly confirmed [2]. An operator monitoring resource utilization or request rate had no dashboard that could have flagged "this agent is about to take an irreversible, out-of-scope action." Monitoring a tool-using agent requires tracking not just whether tool calls succeed, but *which* tools are called, in *what combinations*, and at *what rate* relative to expected behavior—signals that have no counterpart in the golden-signal framework.

2.5 Assumption 4: Cheap Stateless Retries

Microservice monitoring assumes that the canonical response to a detected failure is to retry. Retry logic is built into every HTTP client library; exponential backoff with jitter is standard practice; SLO error budgets implicitly treat retried-and-succeeded requests as recoverable. The assumption underlying all of this is that a failed operation leaves the world in the same state it was in before the attempt—that operations are, at least approximately, idempotent.

Agent steps are often not idempotent. A partially-executed agentic workflow may have already sent an email, issued a charge, or posted a calendar invite before the step that ultimately failed. Retrying the workflow from the beginning repeats all preceding side effects. The Operator incident illustrates the risk: the agent authorized a financial transaction that was not sanctioned by the user and could not be trivially reversed [3]. In this setting, the retry-on-error pattern is not merely unhelpful—it is actively harmful. A monitoring system for agents must be able to distinguish a failure that is safe to retry from one that has already committed an irreversible side effect, a distinction that error-rate dashboards do not and cannot make.

2.6 Summary

The four violated assumptions map directly onto a failure of observability coverage. The golden signals instrument the *service layer*—latency, throughput, resource consumption, and transport-layer errors—but the failure mode of AI agents has moved to the *semantic and behavioral layer*: wrong answers, unexpected action sequences, gradual behavioral drift, and irreversible side effects. The golden signals do not become wrong when agents are present; infrastructure still needs them, and we do not propose removing them. They become *insufficient*. The remainder of this paper constructs the monitoring complement: a suite of signals that instrument the semantic and behavioral layer that golden signals leave dark.

Table 1: Assumptions embedded in microservice golden-signal monitoring and how AI agents violate each one. The four assumption names are used consistently throughout this paper.

Assumption	Microservice practice built on it	How agents break it
Determinism	Canary comparison, retry semantics, error-budget accounting	Stochastic outputs: identical inputs yield different trajectories and outcomes [7]
Explicit failures	Error-rate counters, status-code alerting, exception tracking	Semantic failures return HTTP 200; incorrectness is in the content, not the transport layer [4]
Bounded action space	Blast-radius estimation, circuit breakers, runbook enumeration	Tool-composing agents construct action sequences at inference time; worst-case behavior is not enumerable [2]
Cheap stateless retries	Retry-on-error, idempotent operation design, backoff policies	Partially-executed workflows carry side effects; retrying can double-charge, double-send, or double-delete [3]

3 A Signal Suite for AI Agents

The four violated assumptions identified in Section 2 impose a design constraint on any replacement monitoring scheme: every signal must be computable in production without ground-truth labels. This is the operational analog of the disentanglement principle articulated by Rabanser et al. [1]: offline reliability science can afford repeated runs and held-out annotations; production monitoring cannot wait. The signals proposed here therefore rely on proxies, self-consistency checks, and behavioral reference windows rather than correctness oracles. Where a corresponding offline metric exists in Rabanser et al. [1], we note its label and show how the production signal approximates it under label-free conditions. Where no offline counterpart exists—because the signal depends on deployment context rather than a fixed task distribution—we note that too.

We organize the twelve signals into four families, each targeting a distinct failure dimension that golden-signal monitoring leaves dark. Families S1–S3 supply signals that can be incorporated into a composite health view; Family S4 is explicitly exempted from aggregation—its signals record severity, not frequency, and must be surfaced individually, never averaged into a health score [1, 8].

3.1 S1: Outcome Signals

S1 signals target *semantic quality*—whether the agent produced a correct, useful output—under the constraint that ground-truth labels are unavailable. Traditional monitoring treats quality as binary and transport-visible: either a request errors or it does not. S1 acknowledges that agent outputs succeed at the transport layer while failing at the semantic layer, and assembles proxies that collectively approximate correctness without requiring it.

Proxy success rate. *Intuition.* Direct accuracy measurement requires labels; production streams have none. Proxy success rate assembles a composite of available correlated signals: user-provided

feedback (thumbs, ratings, corrections), downstream-system acceptance (was the generated code committed? was the draft sent?), and sampled LLM-judge verdicts scored against rubrics [9].

Why traditional monitoring misses it. HTTP error rates count transport failures; they are blind to responses that are fluent and wrong. The NYC chatbot returned successful completions that violated employment law [4]. No error counter was incremented.

Protocol. For each completed task, collect whichever proxy signals are available. Aggregate into a weighted composite with weights reflecting estimated proxy quality (user feedback is high-signal; downstream acceptance is moderate-signal; LLM-judge verdicts are useful but themselves imperfect [9]). Report the composite with confidence intervals that widen when few proxies are available. Periodically calibrate the LLM judge against hand-labeled samples to detect judge drift.

Alert. A statistically significant decline in the composite over a rolling window warrants investigation, particularly if LLM-judge scores and downstream-acceptance signals diverge—which may indicate judge degradation rather than task failure.

Escalation & abandonment rate. *Intuition.* When users lose confidence in an agent, they escalate to a human, retry the request with a rephrased prompt, or abandon the session. Each behavior is a revealed-preference signal of failure that requires no label.

Why traditional monitoring misses it. From the infrastructure perspective, a session that ends in abandonment looks identical to one that ends in satisfaction: the connection closes cleanly, latency was acceptable, and no exception was raised.

Protocol. Track per-session: human-handoff events (explicit escalation), user rephrasings within the same task intent (retry without changing goal), and session abandonment before task completion. Report as separate sub-rates, not aggregated. A rising rephrasing rate is the earliest user-visible precursor of quality degradation and is often detectable a full release cycle before aggregate quality metrics move.

Alert. A monotone increase in rephrasings over three or more consecutive measurement windows is a leading indicator. Escalation-to-human spikes are a coincident indicator. Abandonment rises are a lagging indicator; by the time abandonment increases, quality has usually been degraded for some time.

Canary outcome consistency. *Intuition.* Even without ground-truth labels, a fixed canary task set run on a schedule can expose *changes* in model behavior—without requiring that the correct answer be known. Variance across repeated runs of the same task rises before mean accuracy visibly falls.

Why traditional monitoring misses it. Canary deployments in microservice practice compare new-version behavior against old-version behavior on live traffic. For stochastic agents, behavioral change on a fixed task is masked by the natural per-run variance of the model. Classic canary comparison—“did the error rate change?”—cannot distinguish a degraded agent from an inherently variable one.

Protocol. Schedule a fixed canary task set to run periodically (e.g., hourly). Canary tasks are fixed tasks with pre-declared expected outcomes (known-answer checks), so scoring canary replays requires no production labels—that is precisely why canaries exist as a label-free consistency signal. For each task i, collect binary success proxies $y_{i,1}, \ldots, y_{i,K}$ across K runs. Let $\hat{p}_i = \bar{y}_i$ and $\hat{\sigma}_i^2 = \mathrm{Var}(y_{i,1:K})$ (sample variance, Bessel-corrected, matching C_{out} of Rabanser et al. 1). Compute per-task consistency:

$$c_i = \mathrm{clamp}\left(1 - \frac{\hat{\sigma}_i^2}{\hat{p}_i(1-\hat{p}_i) + \varepsilon},\ [0,1]\right), \tag{1}$$

where $\varepsilon = 10^{-8}$ prevents division by zero. Report $\bar{c} = \frac{1}{N}\sum_i c_i$. A task with high $\hat{p}_i$ and rising $\hat{\sigma}_i^2$ has high surface score but increasing fragility—the scenario that standard mean-accuracy dashboards miss.

Alert. A decline in $\bar{c}$ that precedes a decline in $\hat{p}$ is the diagnostic signature of early behavioral instability.

3.2 S2: Trajectory Signals

S2 signals target *behavioral consistency*—whether the agent is following the same kinds of paths it followed before, regardless of whether any individual outcome is known to be correct. This family has no microservice analog: services do not have trajectories. The governing insight is that behavioral drift is detectable at the action-sequence level before it manifests as outcome degradation.

Trajectory divergence. *Intuition.* An agent that begins using different tools, in different orders, for the same tasks is behaving differently. Because LLM behavior is sensitive to surface features of the prompt and context [7], semantically equivalent inputs can produce divergent action sequences—making trajectory distribution shift a reliable early-warning signal even when individual outcomes appear unchanged. This behavioral change may be benign (a prompt update, a new tool) or malicious (prompt injection, model drift)—but in either case it is a signal that something in the system has changed, detectable without any correctness label.

Why traditional monitoring misses it. Distributed tracing records which services were called; it does not record what the agent *chose* to call, in what sequence, or why. A trace that spans the same microservices in the same order may conceal a dramatically different agent reasoning trajectory.

Protocol. Maintain a trailing baseline window W_{base} (e.g., the previous 24 hours). For a current window W_{cur}, compute:

1. *Action-type JSD*: compute empirical action-type distributions p_{cur} and p_{base} over tool-call names; report $\text{JSD}_2(p_{\text{cur}} \| p_{\text{base}})$ (base-2 Jensen–Shannon divergence), the streaming analog of C^d_{traj} from Rabanser et al. [1].
2. *Sequence edit distance*: for same-task pairs $(s_{\text{cur}}, s_{\text{base}})$ sampled across both windows, compute mean normalized Levenshtein distance $\bar{d} = \text{mean}\, \ell(s_i, s_j)/\max(|s_i|, |s_j|)$, the streaming analog of C^s_{traj}.

Report both sub-metrics separately; they are sensitive to different change types (JSD captures distributional shift; edit distance captures sequencing change).

Alert. Alert when either sub-metric exceeds two standard deviations above its rolling mean. Simultaneous spikes in both indicate a structural behavioral change; isolated JSD movement may reflect a new or deprecated tool.

Loop & stall rate. *Intuition.* An agent that repeatedly invokes the same tool with the same arguments has stopped making progress. An agent that exhausts its maximum turn budget without completing the task has stalled. Both patterns are detectable at the step level without any knowledge of the correct answer.

Why traditional monitoring misses it. From the infrastructure view, repeated tool calls are simply requests: each one returns a response, the latency is bounded, and no exception is raised. A crash-loop at the service level is visible; a reasoning loop at the agent level is not.

Protocol. A trace is classified as a loop if any (tool-name, serialized-arguments) pair appears three or more times within the trace. A trace is classified as a stall if the agent exhausts its maximum turn budget (`stop_reason = max_turns`) without a terminal success event. Report the fraction of traces classified as either.

Alert. A loop/stall rate above a deployment-specific baseline threshold (e.g., 5%) warrants investigation; a sudden spike is the more actionable signal, as it often coincides with a specific triggering change such as a prompt update or an upstream API behavior shift.

Tool-call health. *Intuition.* Individual tool-call errors are the one place where classic error monitoring survives into the agentic context. However, the unit of interest is the *step*, not the *request*: a 2% per-step error rate compounds across a trajectory of 50 steps to a per-task failure probability of $1 - (0.98)^{50} \approx 64\%$. The relevant metric is therefore per-step error and retry rates, not the aggregate request error rate.

Why traditional monitoring misses it. APM tools aggregate errors at the request level. An agent task that completes after recovering from three internal tool errors appears as a single successful request—the internal error structure is invisible, and the compounding risk it represents is not surfaced.

Protocol. For each tool-call step in a trace, record: whether the step errored, whether it was retried (same tool called subsequently in the same trace), and whether the arguments were malformed (parse errors, schema violations). Report per-step error rate, per-step retry rate, and malformed-argument rate separately over a rolling window.

Alert. Alert when the per-step error rate rises above a baseline threshold, with separate alerts for malformed-argument spikes (which indicate a prompt or schema change) and error-with-no-retry spikes (which indicate the agent gave up rather than recovering).

3.3 S3: Resource-Envelope Signals

S3 signals correspond most directly to the traditional golden signals—latency, traffic, and saturation—but their semantics shift substantially in the agentic context. Agent resource consumption is determined by the model's reasoning trajectory, not by a fixed computational budget, and is therefore heavy-tailed and task-sensitive in ways that microservice costs are not. The primary operational risk is not mean resource consumption but *envelope expansion*: the tail of the distribution growing without a corresponding change in the mean.

Cost-per-task distribution. *Intuition.* The dollar cost of an agent task is determined by the number and length of LLM calls the agent makes. Cost scales with reasoning depth and is heavy-tailed: most tasks cost little, but a small fraction of tasks trigger extended reasoning chains that cost orders of magnitude more. The relevant operational question is not "what does the average task cost?" but "is the tail growing?"

Why traditional monitoring misses it. Latency and throughput dashboards do not capture LLM token consumption or its dollar equivalent. A deployment that is nominally within budget can hide a growing tail—tasks whose cost is 10× the median—behind a stable mean.

Protocol. For each completed trace, compute the per-task dollar cost from token usage: cost = $(n_{\text{in}} \cdot r_{\text{in}} + n_{\text{out}} \cdot r_{\text{out}})/10^6$, where $r_{\text{in}}, r_{\text{out}}$ are the per-token input and output rates for the deployed model. Report p50, p95, and p99 costs over a rolling window, along with the coefficient of variation $\text{CV} = \sigma/\mu$ (the streaming analog of C_{res} from Rabanser et al. 1).

Alert. Alert on tail expansion: a p95-to-p50 ratio growing beyond a deployment-specific baseline, or a CV rising while the mean is stable. Alert on absolute threshold breach at p99.

Steps- and latency-per-task. *Intuition.* Step count and wall-clock latency are observable without labels and correlate with reasoning complexity. Like cost, they are heavy-tailed; the median task may complete in 10 steps, while pathological cases require 100 or more.

Why traditional monitoring misses it. Service latency dashboards exist but do not decompose latency by reasoning steps, so they cannot distinguish a slow task from a task that took too many reasoning steps (a leading indicator of looping or over-planning).

Protocol. Record step count and wall-clock duration for each completed trace. Report p50 and p95 of both distributions over a rolling window.

Alert. Alert when p95 step count or p95 latency rises substantially above its rolling baseline, especially when accompanied by a stable or declining proxy success rate—which indicates the agent is taking more steps without improving outcomes.

Context saturation. *Intuition.* Language models operate within a finite context window. As context utilization approaches the limit, the model must truncate or compact earlier context, degrading access to task-relevant information. Context saturation is a leading indicator of quality loss that is invisible to users until the degradation is substantial.

Why traditional monitoring misses it. Memory saturation in microservices is a well-understood signal: a service running out of RAM degrades predictably. Context saturation is different: the model continues to operate, returning outputs at normal latency, while silently losing access to earlier context. There is no microservice counterpart for a resource whose exhaustion degrades *what the service knows* rather than *whether it responds.*

Protocol. For each trace, record the maximum input-token count across all steps. Compute window utilization as $u = n_{\max}/L$, where L is the context window size of the deployed model. Track mean and maximum utilization over a rolling window, and count truncation or compaction events per task.

Alert. Alert when mean utilization exceeds 75% of the context limit, or when any trace triggers a truncation event—these are leading indicators of imminent quality degradation. A rising truncation rate without rising proxy success rate is a compound alert.

3.4 S4: Boundary Signals

S4 signals are qualitatively different from S1–S3. They record safety-relevant events whose risk is a function of *severity* and *irreversibility*, not frequency. A single policy violation or a single unauthorized irreversible action can constitute a significant incident; averaging these events into a health score is therefore inappropriate and potentially deceptive [8]. S4 signals should be surfaced on their own dashboard, with per-event alerting rather than threshold-on-aggregate alerting. This treatment mirrors the distinction Rabanser et al. [1] draw between reliability metrics (which are averaged) and safety metrics (which are not).

Policy-violation rate. *Intuition.* Modern agent deployments incorporate policy layers: guardrails that block certain outputs, PII filters, content classifiers, and action-level allow/deny rules. Each triggered denial is a signal that the agent attempted something outside its permitted scope. Rising violation rates indicate either model drift (the agent is increasingly attempting disallowed actions) or guardrail calibration issues (the policy layer is becoming misconfigured).

Why traditional monitoring misses it. Audit logs exist in most systems, but they are typically inspected after-the-fact and are not surfaced as real-time monitoring signals with alerting semantics.

Protocol. Count the fraction of tasks that trigger at least one guardrail denial, PII-filter rejection, or explicit action-level policy block. Maintain a per-policy-rule breakdown to distinguish between guardrail categories (content policy vs. action policy vs. PII).

Alert. Alert on any statistically significant increase from a rolling baseline. Flag individual incidents in which the same task triggers multiple policy violations within a single trajectory—this pattern may indicate an adversarial prompt or a systematic alignment failure.

Irreversible-action rate. *Intuition.* Certain tool calls have real-world side effects that cannot be undone: file writes, database mutations, email sends, financial transactions, calendar events. A spike in irreversible actions per task—relative to a trailing baseline—is the operational precursor to the Replit-class incident: the database deletion was preceded by a burst of destructive tool calls that, had they been monitored, would have been visible before any user complaint [2].

Why traditional monitoring misses it. Tool call counts are not a standard monitoring primitive; and even if they were, standard metrics do not distinguish reversible from irreversible operations.

Protocol. Annotate tool call types as reversible or irreversible at deployment time (e.g., read operations are reversible; write, delete, send, and purchase operations are not). For each trace, record the count of irreversible actions. Report mean irreversible actions per task over a rolling window, and compute unauthorized irreversible action rate as the fraction of tasks that committed a side effect without a corresponding deployment-time authorization annotation (i.e., task categories the operator has declared in-scope for irreversible actions).

Alert. Alert when irreversible actions per task exceed a session-type-specific baseline. Any single trace that commits an unauthorized irreversible action triggers an immediate per-event alert (see S4 intro: never aggregate these events into a health score).

Abstention & deferral calibration. *Intuition.* A well-calibrated agent escalates to a human when it should and proceeds autonomously when it can. Escalation-rate tracking alone does not capture calibration: an agent that always escalates has perfect recall and zero precision. The *joint* distribution of escalation decisions and downstream outcomes provides a streaming proxy for calibration [10–12].

Why traditional monitoring misses it. Human-handoff events appear in service logs, but they are not cross-referenced with downstream outcomes to evaluate whether the escalation was warranted. A system that escalates too rarely (under-deferral) is just as miscalibrated as one that escalates too often (over-deferral), but both failure modes look identical in a raw escalation count.

Protocol. `expect_escalation` is a *deployment-time task-type annotation*: operators declare which request categories are out of policy and must be deferred—for example, refund amounts above a transaction total, or address changes after a shipment has dispatched. These annotations are applied to task categories at configuration time, not derived post-hoc from individual outcomes, and are therefore available in production without requiring human review of each task's result. For tasks annotated with expected-deferral labels (tasks that a properly configured policy declares the agent must escalate), compute:

$$\text{Escalation precision} = \frac{|\{t : \text{escalated}(t) \wedge \text{expect_escalation}(t)\}|}{|\{t : \text{escalated}(t)\}|}, \tag{2}$$

$$\text{Escalation recall} = \frac{|\{t : \text{escalated}(t) \wedge \text{expect_escalation}(t)\}|}{|\{t : \text{expect_escalation}(t)\}|}. \tag{3}$$

Report both metrics over a rolling window. This is the streaming analog of calibration/discrimination analysis [10, 11], instantiated via precision/recall rather than reliability diagrams because downstream outcomes provide binary rather than probabilistic labels.

Alert. Alert when recall falls below a deployment-specific floor (the agent is failing to escalate when it should—an under-deferral failure with potential for irreversible consequences). Alert separately when precision falls below a floor (the agent is escalating unnecessarily—an over-deferral failure that degrades throughput).

3.5 Summary: Table of Signals

Table 2 presents all twelve signals with compact measurement definitions and their closest traditional analog. Dashes in the traditional-analog column indicate signals for which no standard infrastructure metric exists. The table is intended as an implementation reference: the definitions match the `experiments/signals.py` implementation, and the math matches the formulas in Equations 1 and the trajectory-divergence computation.

4 Case Study: An Instrumented Support Agent

4.1 Experimental Setup

To evaluate whether the proposed signals detect operationally relevant failures that traditional infrastructure metrics miss, we built a toy e-commerce support agent. The agent has access to five tools (order lookup, refund initiation, address update, status-email dispatch, and order cancellation) backed by a deterministic mock backend with pre-seeded order fixtures. Tasks are drawn from 21 canonical support scenarios spanning three difficulty classes: routine (lookup, status), deferral-required (large refunds, post-ship address changes), and cancel/refund edge cases.

We ran a full $21 \times 8 \times 4$ factorial design: 21 task types, 8 repetitions per condition, and 4 conditions, for a total of 672 episodes. The agent is `gpt-5.4-nano-2026-03-17`; the LLM judge uses `gpt-5.4-mini-2026-03-17` with the store-policy rubric. Each condition comprises 168 runs.

Conditions. Table 3 summarizes the four experimental conditions.

Evaluation. In this testbed, ground truth is always known: each task has a declared expected outcome and a declared escalation requirement. Ground truth exists here *only to evaluate the signals themselves*—production deployments have no such oracle, which is precisely the motivation for the label-free signal suite proposed in Section 3. The LLM judge was sampled at fraction 0.5 (84 runs per condition, 336 total), with judge verdicts compared against ground truth to quantify proxy quality.

4.2 Incident Replay

Figure 1 presents a simulated production incident using the *perturb* condition as the incident onset. The pseudo-timeline interleaves tasks in rep-major order (mimicking mixed production traffic), using sliding windows of 42 runs (step 7). The incident represents an input-drift event: task descriptions are paraphrased and tool-output JSON fields are reordered, as would occur if an upstream schema changed or a localization layer was introduced.

Traditional signals. Throughout the incident, p95 latency, infrastructure error rate, and throughput remain nominal. The **perturb** condition achieves a per-step tool-call error rate of 0.024 (baseline: 0.055), and the JSD component of Trajectory divergence is 0.003 (baseline: 0.000), both below any operational threshold. An operator monitoring only the golden signals would see nothing.

Proposed signals at onset. The incident is visible to two signal families. First, the unauthorized fraction of Irreversible-action rate steps from 0 (baseline) to 0.172 at incident onset and holds—the agent begins committing side effects on task categories the operator has not authorized for irreversible actions. Second, Canary outcome consistency drops from 0.905 (baseline) to 0.810 under perturbation, reflecting increased per-task variance as the reordered JSON causes inconsistent tool-output parsing.

Complementarity. The JSD (distributional) component of Trajectory divergence does not move: format-level drift (field reordering, paraphrasing) does not substantially change *which tools* the agent selects in aggregate, so the action-type distribution stays similar. However, the sequence (normalized edit distance) component rises from its baseline self-reference of ≈0.02 to ≈0.10 under perturb, reflecting ordering changes in tool call sequences—so the trajectory family *does* register the incident, through its sequence sub-metric rather than its distributional one. The boundary signal (Irreversible-action rate) localizes the harm: it is the unauthorized-action fraction that most directly flags what has gone wrong. This illustrates sub-metric complementarity: within a single signal family, the distributional component may be silent while the sequence component fires, and a separate boundary signal provides the clearest behavioral interpretation. A monitoring system that deployed only boundary signals would miss a prompt-injection attack that changes tool-selection patterns without firing unauthorized actions.

4.3 Signal × Condition Profile

Figure 2 presents the full signal-by-condition profile. Several results merit emphasis.

Fault condition. Fault injection (30% on two tools) is the most visible condition across the board. Tool-call health shows the largest absolute movement: per-step error rate rises from 0.055 (baseline) to 0.280, fault rate from 0 to 0.241, and retry rate from 0 to 0.205. Canary outcome consistency collapses from 0.905 to 0.381—the single largest movement across all signal×condition pairs. True success rate falls from 0.99 to 0.79. Importantly, the precision component of Abstention & deferral calibration drops from 0.96 to 0.63 while recall holds at 1.00: the failure is entirely precision-side. Under fault conditions the agent escalates spuriously (it escalates on tasks it would have handled correctly in baseline) but misses no warranted escalation, indicating that fault-induced uncertainty leaks into the boundary-family signal even though no unauthorized actions are taken. Note that injected tool faults are *partially* visible to a traditional step-level error counter; the agent-specific signals add the consistency collapse and the escalation-precision picture that a step counter alone cannot provide.

Perturb condition. As detailed in Section 4.2, the perturb incident is characterized by the boundary family. The unauthorized fraction of Irreversible-action rate rises from 0 to 0.172 (per run: 0.66 vs. 0.54 irreversible actions per run). True success rate declines from 0.99 to 0.90. The tool-health family is largely silent (per-step error rate actually falls from 0.055 to 0.024). Within the trajectory family, the JSD (distributional) component stays near zero (0.003), but the sequence component rises from its baseline self-reference of 0.02 to 0.104, registering ordering changes; the boundary family provides the clearest signal, underscoring the sub-metric complementarity result.

Degraded condition. The degraded condition—a distractor-laden system prompt including an explicit "goodwill exceptions" anti-escalation nudge—is the paper's most instructive robustness

finding. The prompt barely moved accuracy: true success declines from 0.99 to 0.95, and the model produced *no unauthorized actions whatsoever* (unauthorized rate: 0.0). The model resisted the anti-escalation bait at the outcome level. Yet the Abstention & deferral calibration signal registered the degradation: escalation recall fell from 1.00 to 0.88, and escalation rate from 0.40 to 0.33, reflecting the agent's reduced tendency to escalate tasks that should be deferred. Escalation precision, by contrast, improved to 1.00 (all escalations that did occur were warranted), revealing the mechanism: the agent under-deferred rather than over-deferred. Frame: even when a degradation is too subtle for outcome metrics—the model resists the bait—the boundary family still registers it. We caution, however, that this is a single-model result; robustness to adversarial prompts may not generalize across model families.

4.4 Proxy Quality and Cost

Because ground truth exists only in this testbed—not in production—we use it here to audit the proxy itself, not to operate the system.

Judge agreement. Table 4 reports LLM-judge agreement with ground truth per condition. Agreement is stable across baseline (0.798), perturb (0.833), and degraded (0.833) conditions. Under fault injection, however, agreement drops to 0.667. This is the paper's most interesting empirical finding about monitoring infrastructure itself: *the proxy-success signal degrades exactly when the system is under the most stress*. The judge, trained on well-formed exchanges, is less accurate at evaluating outputs produced under fault conditions—which are noisier, more ambiguous, and structurally different from the clean exchanges in the judge's reference distribution.

The practical implication is direct: the judge should be treated as one voice among several signal families, not as the arbiter of agent health. A system that relies solely on LLM-judge proxy success will be least accurate when it is most needed. Section 5 discusses the "monitor the monitor" protocol—periodic human validation of judge verdicts—as a mitigation.

Deferral calibration. Figure 3 plots escalation precision and recall across conditions. The figure illustrates the calibration collapse under fault and the under-deferral signature under degraded.

Cost distribution. Figure 4 shows per-task cost CDFs under the four conditions, computed at list prices of \$0.20/MTok input and \$1.25/MTok output for `gpt-5.4-nano-2026-03-17`.[1] The fault condition shows a lower p50 (fewer completed steps before giving up) but a heavier tail: tasks that do complete under fault injection are more expensive, because the agent invests additional reasoning steps attempting recovery. Across all conditions, the p95-to-p50 cost ratio illustrates the characteristic heavy tail of agentic cost distributions noted in Section 3.3. Baseline cost per task: p50 = \$0.000435, p95 = \$0.000735, CV = 0.33. Context utilization against the 400,000-token window stays well under 0.3% across all conditions (mean = 0.16%, max = 0.28%); the mock-backend traces are short, so context saturation is not a stress dimension in this testbed.

[1] Dollar figures use list prices for `gpt-5.4-nano-2026-03-17` (\$0.20/MTok input, \$1.25/MTok output) and `gpt-5.4-mini-2026-03-17` (\$0.75/MTok input, \$4.50/MTok output), per `https://developers.openai.com/api/docs/models/gpt-5.4-nano` and `https://developers.openai.com/api/docs/models/gpt-5.4-mini`. Token counts are the primary quantity; the full 672-run matrix consumed approximately 1.17M input tokens and 61k output tokens (≈\$0.31 agent-side; total including judge well under \$1).

4.5 Limitations

Several limitations bound the scope of these results.

Toy domain. The support agent operates on a small, deterministic mock backend with 21 task types. Real support deployments have orders-of-magnitude more task variety, noisier tool responses, and more complex multi-step dependencies. The figures presented here illustrate the signal taxonomy rather than validate it at production scale.

Single model. All 672 episodes use a single model (`gpt-5.4-nano-2026-03-17`). Signal sensitivity and the complementarity structure may differ substantially for other model families, sizes, or versions. In particular, the degraded-condition finding—that the model resisted the anti-escalation nudge—should not be generalized; robustness to adversarial prompts varies considerably across models.

Simulated (near-zero) tool latencies. Tool calls in the mock backend return in microseconds. The Steps- and latency-per-task panel is therefore trivially flat across conditions, and no latency-based alerting is triggered. In production, tool latency variance would add a diagnostic dimension that is absent here.

LLM-judge circularity and measured degradation. The judge is an LLM evaluating another LLM's outputs. Beyond this structural circularity, we measured a concrete degradation: judge agreement falls to 0.667 under fault injection (from 0.798 at baseline). Both the circularity and the measured degradation counsel against relying on proxy success as a primary health signal.

Small n. 168 runs per condition (84 judged) is sufficient to illustrate signal behavior but insufficient to derive precise estimates of signal sensitivity or specificity at arbitrary threshold settings.

Canaries as known-answer tasks. The escalation and consistency signals use deployment-time annotations (expected-deferral labels) that must be declared by operators in advance. In the testbed these are given; in production they require up-front task-taxonomy work.

5 Monitoring Architecture and Discussion

5.1 Reference Pipeline

The twelve signals in Section 3 are not free-floating metrics; they require a coherent collection and computation pipeline to be operationally useful. We describe a reference architecture with five layers.

(a) Structured trace emission. Every agent run emits a structured trace: a JSON event stream recording each tool-call step (tool name, arguments, result, error flag, token usage, wall time) plus run-level metadata (task type, session identifier, model version, system-prompt hash). Traces are emitted per-step during execution rather than buffered to completion, so that partial traces from stalled or crashed runs are recoverable. We align trace fields with the OpenTelemetry Generative AI semantic conventions [13], which define standard attribute names for LLM spans (`gen_ai.request.model`, `gen_ai.usage.input_tokens`, etc.), allowing the signal pipeline to sit atop existing OTel infrastructure without additional instrumentation per deployment.

(b) Stream processor for S2/S3 signals. A stream processor subscribes to the trace stream and maintains sliding windows over recent runs. For S2 signals (Trajectory divergence, Loop & stall rate, Tool-call health), the processor computes window statistics over the last W runs or the last T hours. For S3 signals (Cost-per-task distribution, Steps- and latency-per-task, Context saturation),

it maintains running percentile sketches (e.g., a t-digest) to support efficient p50/p95/p99 queries without buffering all trace data. The baseline window against which divergence is computed is a configurable trailing period; Section 5.2.1 discusses the choice.

(c) Canary scheduler for S1.3. Canary outcome consistency requires a scheduler that periodically replays a fixed task set and aggregates per-task outcomes. In production, canary tasks are "known-answer" tasks—tasks for which the operator has pre-declared the expected outcome or escalation decision at configuration time. The scheduler submits canary tasks to the production agent (or a shadow deployment) on a fixed cadence (e.g., hourly), collects binary outcome proxies from the judge service, and feeds them into the consistency formula (Equation 1). Canary throughput should be chosen to stay below 5% of production volume to avoid distorting cost and throughput metrics.

(d) Sampled judge service for S1.1, with periodic human validation. Proxy success rate is computed by a sampled LLM-judge service that evaluates a fraction of production traces against a rubric. The sampling fraction is the primary cost/coverage knob: judge cost scales linearly with the sampling fraction (and with judge prompt length, since each verdict re-reads the trace being judged). Our case study ran the judge at fraction 0.5 (336 of 672 runs), which should be read as an upper bound on what production needs; in practice, sampling fractions of 0.05–0.10 are sufficient for stable window-level estimates when windows contain hundreds of runs.

The case study finding that judge agreement falls to 0.667 under fault injection (Section 4.4) motivates a "monitor the monitor" protocol: operators should periodically collect a small set of human-labeled samples (e.g., weekly, $n \approx 50$) and compare judge verdicts against human labels. A statistically significant drop in judge–human agreement is an early indicator that the judge's reference distribution has drifted—either because the agent's output distribution has changed, or because a new task type has been introduced that the judge rubric does not cover. This feedback loop is a specific instance of the hidden technical debt described by Sculley et al. [14]: proxy signals that are calibrated at deployment time will drift as the deployment evolves, and the drift is invisible unless the proxy itself is periodically validated.

(e) Alerting layer. The alerting layer applies different semantics to different signal families, reflecting the asymmetry noted in Section 3.

S1–S3 signals use *tail-focused, rolling-baseline thresholds*: a signal fires when it exceeds its rolling mean by more than k standard deviations ($k = 2$–3 is typical), or when a named percentile crosses a deployment-specific limit. These thresholds should be calibrated during an initial burn-in period and re-calibrated after intentional changes (prompt updates, model upgrades).

S4 signals (Policy-violation rate, Irreversible-action rate, Abstention & deferral calibration) use *per-event alerting for extreme values and hard thresholds for rates*: any single trace that fires an unauthorized irreversible action triggers an immediate page, regardless of the rolling rate. Averaging S4 signals into a composite health score is explicitly prohibited [8]; a system that is "99% compliant" on irreversible actions has committed irreversible actions in 1% of runs, which may be unacceptable depending on the action type.

5.2 Discussion

5.2.1 Baseline-Window Choice for Divergence Signals

Trajectory divergence computes JSD between a current window and a baseline window; the choice of baseline-window duration governs the signal's sensitivity. Short baseline windows are sensitive to rapid change but susceptible to false positives from natural variance; long baseline windows suppress natural variance but may normalize gradual drift, leaving slow behavioral shift undetected.

This is a known problem in the concept-drift literature [15]. The standard recommendation is to maintain *two* baseline windows: a short-horizon window (e.g., one hour) sensitive to sudden changes, and a long-horizon window (e.g., 24 hours) sensitive to gradual drift. Alerts that fire on the short-horizon window alone require confirmation from the long-horizon signal before triggering remediation, reducing false positives from natural variance while preserving sensitivity to genuine drift. In our case study, the baseline window is the entire baseline condition (168 runs), which serves as a clean reference but is not representative of a streaming deployment. Practitioners should calibrate window lengths against their deployment's natural run-to-run variance before setting alert thresholds.

5.2.2 Proxy Failure Modes and Feedback Loops

The LLM judge is the most complex proxy in the system, and it has three independent failure modes.

Judge drift. The judge is calibrated on the output distribution at deployment time. As the agent evolves (new tools, new task types, prompt changes), the judge's rubric may no longer cover the relevant output space. This is detectable via the human-validation protocol described above.

Distributional circularity. The judge uses the same underlying model family as the agent. If the agent's failure mode is a subtle misunderstanding of a domain-specific term, the judge may reproduce the same misunderstanding. Mitigations include using a different model family for the judge, or supplementing LLM-judge verdicts with domain-specific rule-based checks.

Feedback loops. If proxy success rate is used as a training signal (e.g., in a reinforcement learning from human feedback pipeline), a judge that is systematically biased will reinforce the agent's biased behavior, creating a positive feedback loop [14]. The safest practice is to treat proxy success as a monitoring signal only—never as a training target without human validation of the proxy quality.

5.2.3 Monitoring Cost

Monitoring is not free. The dominant cost in the S1.1 pipeline is judge inference; the dominant cost in the S1.3 pipeline is canary agent runs. The cost/coverage trade-off can be managed at two knobs: sampling fraction (for the judge) and canary cadence (for the scheduler).

In our case study, the judge was sampled at fraction 0.5. Judge cost per verdict depends on both the judge model's per-token price and the length of the trace being judged. At list prices, `gpt-5.4-mini-2026-03-17` costs $0.75/MTok input and $4.50/MTok output—approximately 3.75× the agent's input price and 3.6× its output price—so a judge model priced meaningfully above the agent model can offset its shorter outputs; the sampling fraction—not the model choice—is the real cost lever. At lower sampling fractions (0.05–0.10), the coverage reduction is modest for stable window-level estimates, and the cost reduction is proportional. Operators with tight inference budgets should prefer lower sampling fractions with the human-validation protocol rather than high

sampling fractions without it: a well-calibrated judge at 5% coverage is more informative than an uncalibrated judge at 50% coverage.

5.2.4 What To Do When Signals Fire

This paper proposes signals, not remediation strategies; a full incident-response playbook is out of scope. However, the signal families suggest natural first-response actions.

When **S2 trajectory signals** fire (behavioral drift), the first response is to freeze the model and prompt versions, replay canaries against the current production configuration to confirm the signal, and compare with the most recent intentional change (prompt update, tool addition). Behavioral drift that is unrelated to a known change is a strong indicator of upstream data or context shift.

When **S1 outcome signals** fire (quality degradation), the first response is to increase the judge sampling fraction to get a higher-confidence estimate, and to inspect the judge–human agreement on the flagged window. If the judge itself has drifted (as measured by human validation), the signal requires recalibration before acting.

When **S4 boundary signals** fire (policy or irreversible-action events), the response depends on severity. Isolated policy violations may indicate a prompt injection or an edge-case task type; a spike indicates systemic miscalibration. Any unauthorized irreversible action should trigger immediate tightening of action gates—a move from "audit" to "pre-authorize" mode for the affected action type—pending root-cause analysis. Deferral-calibration collapse (recall falling below the deployment floor) should trigger addition of explicit escalation rules for the failing task categories, rather than relying on the model's judgment.

6 Related Work

6.1 Agent Reliability Evaluation

The closest antecedent to this work is Rabanser et al. [1], who establish a rigorous offline science of AI agent reliability through controlled benchmarks, repeated sampling, and labeled evaluation. Their framework defines the consistency metrics—C_{out}, C^{d}_{traj}, C^{s}_{traj}, C_{res}—that we adapt into streaming, label-free production signals (Section 3). The distinction is fundamental: offline reliability science affords K independent runs of each task with ground-truth annotation; production monitoring must work from a single run per task, no annotation, and a distribution of tasks that shifts over time. This paper is the operational complement to Rabanser et al. [1], not a replacement.

The agent benchmark literature provides rigorous evaluation frameworks but shares the offline orientation. τ-bench [16] stresses tool-agent-user interaction over realistic task domains; GAIA [17] measures general-assistant capability across diverse tasks; AgentBench [9] evaluates LLMs-as-agents across eight environments. These benchmarks reveal a great deal about *capability* but are not designed for the operational question of *how to detect capability degradation in a live deployment*. The scaffolding paper that underlies most modern agent systems, ReAct [18], proposes interleaving reasoning and action but does not address the monitoring or observability of deployed agents. The safety literature [19] frames AI reliability concerns at a broader level—reward misspecification, scalable oversight, safe exploration—that motivates but does not operationalize the monitoring work here.

6.2 ML Production Monitoring

The machine learning engineering community has documented the challenge of keeping trained models reliable in production. Sculley et al. [14] introduce the concept of hidden technical debt in

ML systems, noting that the entanglement between model artifacts, training pipelines, and serving infrastructure creates failure modes that appear only at deployment time and are invisible to standard software metrics. Our monitoring signals operationalize this insight: the proxy-feedback-loop failure mode discussed in Section 5.2.2 is a direct instance of the hidden debt that Sculley et al. [14] describe.

Concept drift [15] is the statistical phenomenon underlying the trajectory and outcome signals in S1–S2. Standard drift detectors (ADWIN, Page-Hinkley, DDM) operate on scalar prediction error streams; our adaptation must handle action-sequence distributions and semantic proxy signals rather than prediction labels, making direct application infeasible. The trajectory-divergence signal uses JSD over action-type distributions as a distribution-free analog of these detectors.

Calibration research [10] and selective prediction [11] address the question of when a model should defer to a human. The Abstention & deferral calibration signal in S4 applies these ideas in an operational setting where the agent's deferral decisions can be compared against deployment-time annotations rather than held-out labels, enabling production-time calibration monitoring without a ground-truth oracle.

6.3 Software Observability

The SRE golden signals [5]—latency, traffic, errors, and saturation—remain the correct and sufficient basis for instrumenting the *infrastructure layer* of any system that includes an AI agent. We do not challenge this. Our contribution is the complementary *semantic layer*: the signals that capture whether the agent is doing the right things, not merely whether the infrastructure is serving requests.

Distributed tracing [6] provides the causal plumbing that makes multi-hop request attribution possible; it is the natural collector for the structured trace events our pipeline requires (Section 5.1). The OpenTelemetry Generative AI semantic conventions [13] extend distributed tracing with LLM-specific span attributes—model name, token counts, finish reason—that the S1–S3 signals consume directly. These conventions are excellent *plumbing*: they give operators a standard vocabulary for what an LLM call emitted. They do not, however, specify what those emissions *mean* for agent reliability. A trace that records `gen_ai.usage.input_tokens = 4096` does not, by itself, signal that the agent is approaching context saturation or that its behavior has drifted from its baseline. Our contribution is the *semantics* layer above the plumbing: how to interpret the trace stream as a reliability signal rather than merely recording it.

7 Conclusion

AI agents are deployed like microservices and monitored like microservices—but they fail like models, not services. Traditional infrastructure dashboards instrument the transport layer: they count latency, throughput, and error codes. An agent that returns a fluent, grammatically correct response that advises a business to violate employment law, or that deletes a production database by composing a sequence of individually permissible operations, registers as a success on every standard dashboard. The service was healthy; the agent was not. The root of this gap is that golden-signal monitoring rests on four assumptions that agents systematically violate—determinism, explicit failures, bounded action space, and cheap stateless retries—and no existing monitoring primitive instruments the semantic and behavioral layer where agent failures actually live.

This paper proposes a suite of twelve production-observable signals in four families to close that gap. S1 (outcome signals) assembles semantic quality proxies from user feedback, downstream acceptance, and LLM-judge verdicts—without requiring ground-truth labels. S2 (trajectory signals)

detects behavioral drift and structural pathologies at the action-sequence level, before they manifest in outcome metrics. S3 (resource-envelope signals) adapts cost, step count, and context saturation metrics to the heavy-tailed, trajectory-determined resource consumption of agentic workloads. S4 (boundary signals) records safety-relevant events—policy violations, unauthorized irreversible actions, deferral miscalibration—whose operational semantics demand per-event alerting rather than aggregate health scoring. The instrumented case study demonstrates that an input-drift incident leaves traditional signals entirely flat while two families of proposed signals fire: the unauthorized fraction of Irreversible-action rate rises from zero to 0.17 at incident onset, and Canary outcome consistency drops from 0.905 to 0.810. The study also surfaces a "monitor the monitor" finding: LLM-judge agreement with ground truth degrades from 0.798 at baseline to 0.667 under fault injection, exactly when reliable proxy measurement is most needed—motivating periodic human validation of the judge as a first-class component of monitoring architecture. The proposed signals complement, rather than replace, the golden signals: the golden signals remain the correct and sufficient lens for the infrastructure layer; the new signals instrument the semantic and behavioral layer that golden signals leave dark.

Several important questions remain open. **Production-traffic validation.** The case study uses a controlled toy domain with 672 episodes; validation on real production traffic, across diverse task types and model families, is the highest-priority next step. In particular, the complementarity structure observed here—where different signal families are sensitive to different incident classes—may shift substantially under richer task distributions. **Trace schema standardization.** The signal pipeline assumes structured per-step traces with consistent field names. Today, trace schemas vary across agent frameworks, model providers, and deployment toolchains. Standardization—building on the OpenTelemetry Generative AI conventions already referenced in this work—would make the proposed signals portable across deployments without bespoke instrumentation. **Closed-loop remediation.** This paper proposes signals, not responses; what an operator should do when signals fire is treated as out of scope. The natural next contribution is a closed-loop remediation layer: signal patterns associated with specific incident classes (behavioral drift, deferral collapse, cost tail expansion) should map to specific automated responses (canary freeze, escalation rule injection, action-gate tightening), closing the loop from detection to intervention. Together, these directions constitute a path from the monitoring primitives proposed here to a full production reliability practice for AI agents.

References

[1] Stephan Rabanser, Sayash Kapoor, Peter Kirgis, Kangheng Liu, Saiteja Utpala, and Arvind Narayanan. Towards a science of ai agent reliability. *arXiv preprint arXiv:2602.16666*, 2026.

[2] Beatrice Nolan. An AI-powered coding tool wiped out a software company's database, then apologized for a "catastrophic failure on my part". Fortune, July 23, 2025. `https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure/`, 2025.

[3] Geoffrey A. Fowler. I let openai's operator agent buy my groceries. it spent $31.43 without asking. The Washington Post, 2025.

[4] Colin Lecher. Nyc's ai chatbot tells businesses to break the law. The Markup, 2024.

[5] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media, 2016.

[6] Cindy Sridharan. *Distributed Systems Observability.* O'Reilly Media, 2018.

[7] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design. *arXiv preprint arXiv:2310.11324*, 2023.

[8] Stanley Kaplan and B. John Garrick. On the quantitative definition of risk. *Risk Analysis*, 1 (1):11–27, 1981.

[9] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[10] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning*, 2017.

[11] Yonatan Geifman and Ran El-Yaniv. Selective classification for deep neural networks. In *Advances in Neural Information Processing Systems*, 2017.

[12] Glenn W. Brier. Verification of forecasts expressed in terms of probability. *Monthly Weather Review*, 78(1):1–3, 1950.

[13] OpenTelemetry Community. Semantic conventions for generative ai systems. `https://opentelemetry.io/docs/specs/semconv/gen-ai/`, 2025.

[14] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, 2015.

[15] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4), 2014.

[16] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

[17] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: A benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023.

[18] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

[19] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

Table 2: The twelve production-observable monitoring signals, grouped by family. All signals are computable without ground-truth labels. "Traditional analog" refers to the closest existing infrastructure or APM metric; "—" indicates no standard analog exists.

Signal	**Definition / measurement protocol**	**Traditional analog**
S1: Outcome		
Proxy success rate	Weighted composite of user feedback, downstream-acceptance signals, and sampled LLM-judge verdicts; reported with confidence bands reflecting proxy quality and periodically validated against labeled samples.	Error rate
Escalation & abandonment rate	Per-session counts of human handoffs, user rephrasings (same intent, new phrasing), and session abandonment before task completion; reported as separate sub-rates.	—
Canary outcome consistency	Fixed canary task set run on a schedule; per task: $c_i = \mathrm{clamp}(1-\hat{\sigma}_i^2/(\hat{p}_i(1-\hat{p}_i)+\varepsilon), [0,1])$, where $\hat{\sigma}_i^2$ is the sample (Bessel-corrected) variance across K runs; reported as mean $\bar{c}$ (streaming C_{out}, Rabanser et al. 1).	Canary deploys
S2: Trajectory		
Trajectory divergence	(1) $\mathrm{JSD}_2(p_{\mathrm{cur}} \| p_{\mathrm{base}})$ over tool-call name distributions vs. trailing baseline window (streaming C^d_{traj}); (2) mean normalized Levenshtein distance on same-task action sequences (streaming C^s_{traj}).	—
Loop & stall rate	Fraction of traces where any (tool, args) pair repeats ≥ 3 times, or where `stop_reason = max_turns`.	Crash-loop detection
Tool-call health	Per-step error rate, per-step retry rate, and malformed-argument rate over a rolling window; a 2% per-step error rate compounds to $\approx 64\%$ per-task failure over 50 steps.	HTTP 5xx rate
S3: Resource envelope		
Cost-per-task distribution	Per-task cost in dollars from token usage; report p50/p95/p99 and coefficient of variation $\mathrm{CV} = \sigma/\mu$ (streaming C_{res}, Rabanser et al. 1); alert on tail expansion, not mean shift.	Latency percentiles
Steps- and latency-per-task	p50 and p95 of per-task step count and wall-clock duration; rising p95 steps with stable proxy success rate indicates over-planning or looping.	—
Context saturation	Window utilization $u = n_{\max}/L$; mean and max over rolling window; truncation/compaction event count per task. Rising utilization is a leading indicator of quality loss.	Memory saturation
S4: Boundary (per-event; never aggregated into a health score)		
Policy-violation rate	Fraction of tasks triggering at least one guardrail denial, PII-filter rejection, or action-level policy block; broken down by policy category.	Audit logs
Irreversible-action rate	Mean irreversible tool calls (writes, deletes, sends, purchases) per task; unauthorized irreversible action rate against deployment-time authorization annotations (operator-declared in-scope task categories); spike is the precursor signature of the Replit incident [2].	—
Abstention & deferral calibration	Escalation precision and recall against deployment-time expected-deferral labels; streaming proxy for calibration/discrimination [10, 12].	Circuit breakers

Table 3: Experimental conditions. Each condition runs 21 tasks × 8 reps = 168 episodes.

Condition	Intervention
Baseline	Clean system prompt; no perturbation.
Fault	30% fault injection on two tools (lookup and refund); faults return well-formed timeout errors.
Perturb	Paraphrased task requests plus reordered JSON fields in tool outputs (input-drift / schema change).
Degraded	Distractor-laden system prompt including an explicit "goodwill exceptions" anti-escalation nudge.

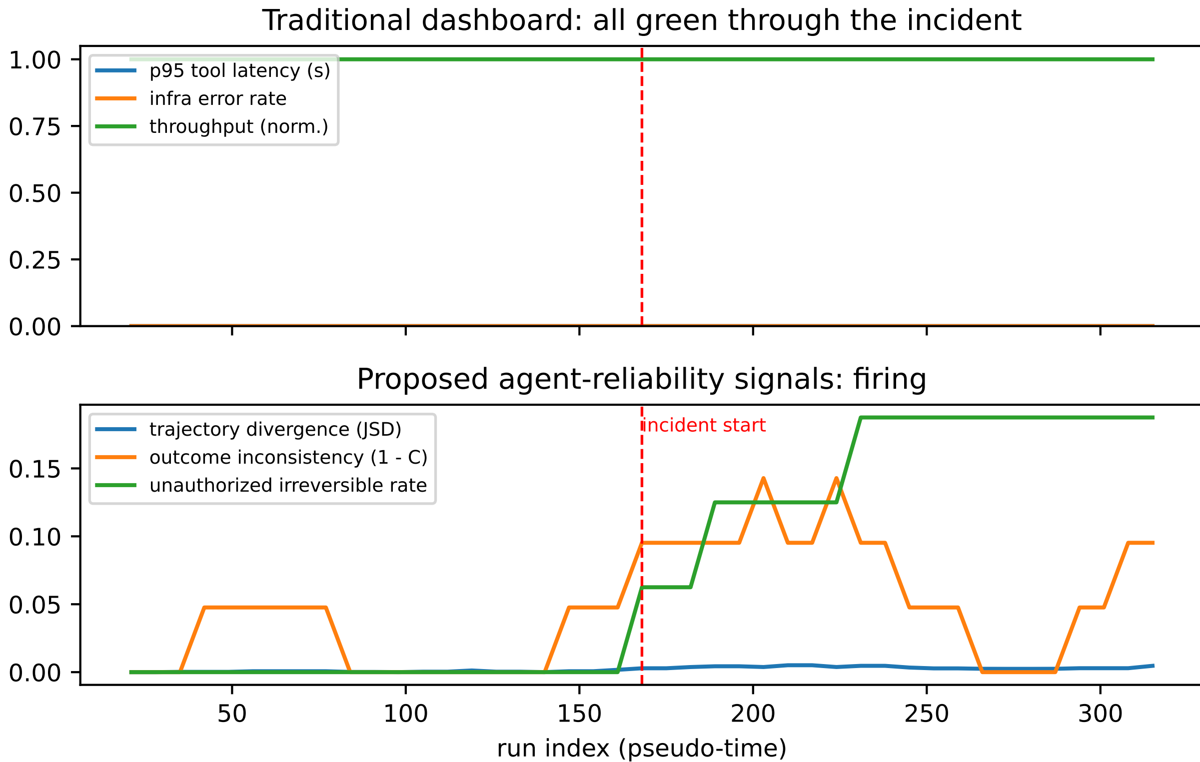

Figure 1: Incident replay: simulated production timeline (perturb condition as the incident, onset at window boundary). Top row: traditional signals (p95 latency, infrastructure error rate, throughput)—all remain flat. Bottom row: proposed signals—the unauthorized fraction of Irreversible-action rate rises over the first windows after onset to ≈0.17 and holds; outcome inconsistency (the complement of Canary outcome consistency) roughly doubles; the plotted trajectory line is the JSD (distributional) component of Trajectory divergence, which stays near zero throughout (the sequence component, normalized edit distance, rises from its baseline self-reference of ≈0.02 to ≈0.10 under perturb, registering the incident through ordering changes). The complementarity result: the distributional component of trajectory divergence is insensitive to this incident class (format-level drift does not change the action-type distribution), but the sequence component and the boundary signal (Irreversible-action rate) each catch distinct aspects of it.

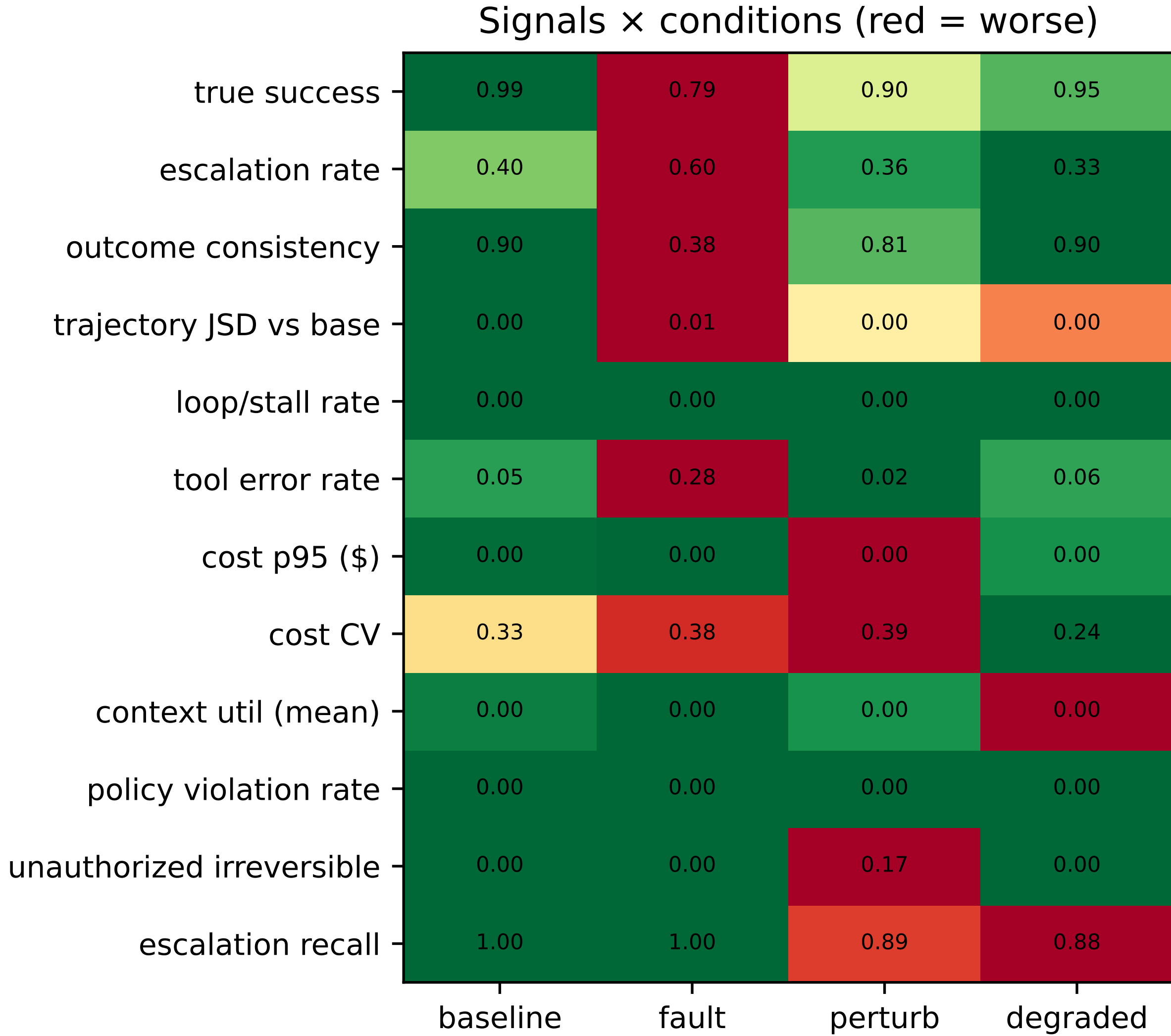

Figure 2: Signal heatmap: raw signal values across all four conditions, colored by per-row min–max normalization (red = worse for that signal; green = better). Each column is one condition; each row is one signal. The boundary family (S4) fires on perturb; the outcome and tool-health families fire on fault; the boundary-calibration signal fires on degraded.

Table 4: LLM-judge agreement with ground truth (84 runs per condition; sampling fraction 0.5). Proxy success rate is the judge's verdict rate; agreement is fraction matching the ground-truth label.

Condition	Proxy success rate	Judge agreement
Baseline	0.810	0.798
Degraded	0.810	0.833
Perturb	0.786	0.833
Fault	0.702	0.667

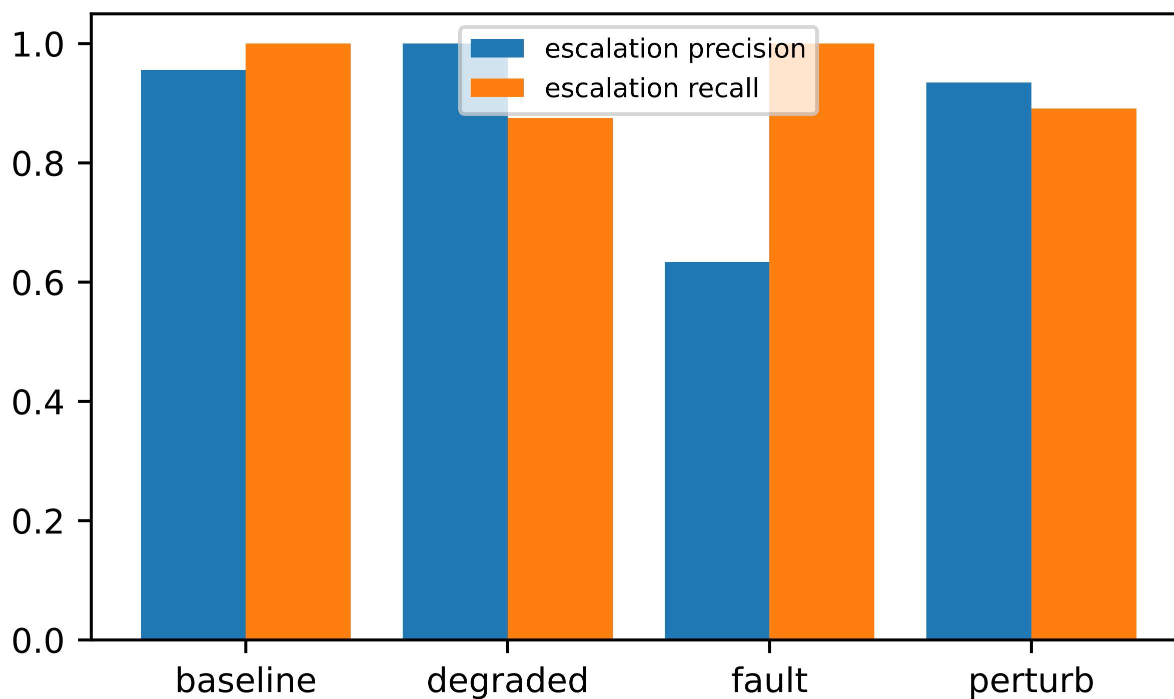

Figure 3: Abstention & deferral calibration: escalation precision and recall across four conditions. Fault injection collapses precision (0.96→0.63); the degraded prompt collapses recall (1.00→0.88) while preserving precision.

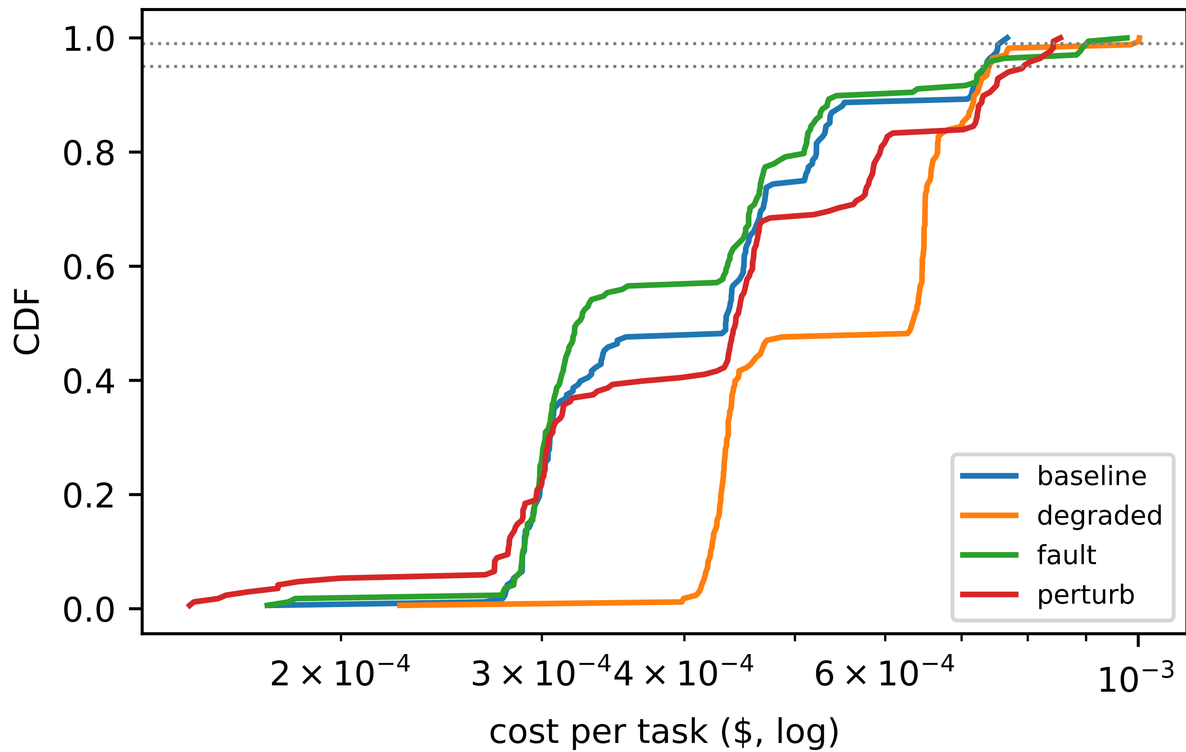

Figure 4: Per-task cost CDF under four conditions (dollars at list prices; see footnote). The fault condition exhibits a lower median but a heavier tail; the degraded condition exhibits the highest median (longer prompts from the distractor-laden system prompt).