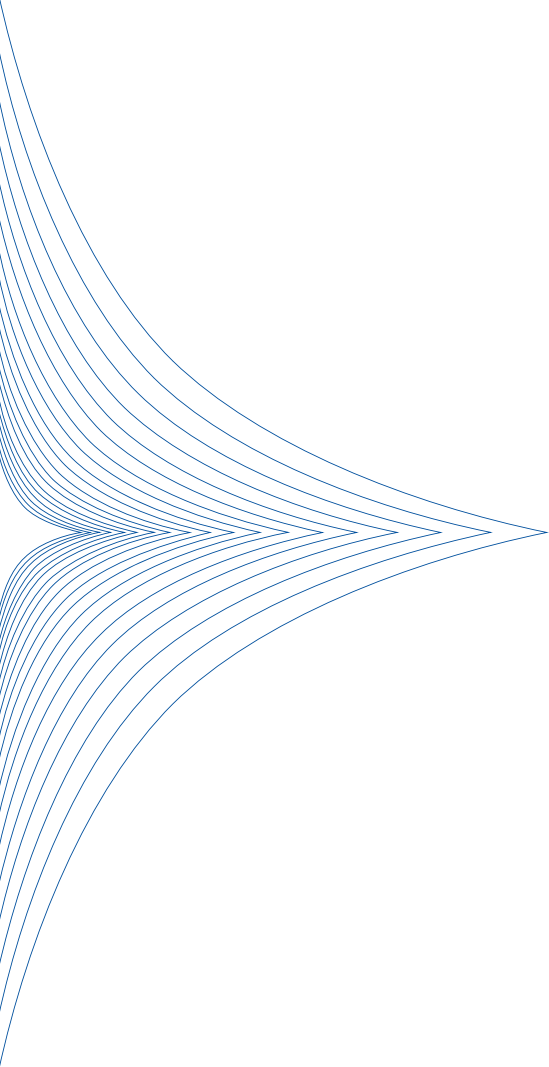




WHITE PAPER

The Bridge Pattern for AI Observability





Overview

Production agents fail in ways their own replies never disclose. We rebuilt the trace from outside six of them, across three runtimes that expose no telemetry of their own, and report the failures that became visible as a result

AUTHOR

Pratham Chintrate

Independent assessment practice

October 2026

Correspondence: the author

Abstract

BACKGROUND

An evaluation answers whether a named build met a stated bar before release. It cannot answer what the deployed system is doing now, on traffic nobody chose. The published failure distribution makes that gap concrete: across 91,747 production tool calls the dominant failures are exhausted context, missing resources, timeouts and stale state, and none of them is visible in the answer the agent returns.

PROBLEM

The agents we were asked to observe run inside vendor runtimes that expose no hook to instrument and emit no spans of their own. The platform therefore has nothing to ingest, and an empty dashboard reads as a healthy agent.

METHOD

We placed a proxy in front of the runtime rather than inside it, reconstructing an OpenInference span tree from what an external observer can see, exporting it over OTLP, and judging off the request path. The pattern was built against three unrelated runtimes and applied to six production agents.

RESULTS

Each trace exposed a failure the agent's own reply did not disclose: a submission scored full marks on some attempts and found no rubric on others; a generated query changed between runs of an unchanged question; a retrieval tool failed on every call while the reply still read as confident. We also report four occasions on which the trace export itself returned success and delivered nothing, a failure of the same kind as the ones it was built to catch.

LIMITS

Six agents, three runtimes, one assessor. A reconstructed trace is thinner than a native one, sampling makes coverage a statistic rather than a guarantee, and observation is not enforcement.

KEYWORDS

AI observability · agent tracing · OpenInference · OpenTelemetry · closed runtimes · silent degradation · model-as-judge · sampling economics

Contents

1	The gate and the watch	5
2	How agents actually fail in production	6
3	What a trace is, and which standard it follows	8
4	Runtimes that emit nothing	8
5	The bridge	10
6	The production stack	12
7	Six agents, and what their traces exposed	13
8	Connectors into the evaluation engine	16
9	Four silent successes	17
10	What an observability platform does not solve	18
11	The cost governor	20
12	Detection, and which way round to build it	21
13	Direction of travel	21
14	What this does not do	22
15	Conclusion	23
16	References	24
	About QKTech	25

Figures

Figure 1	5	Figure 6	14
Figure 2	6	Figure 7	17
Figure 3	8	Figure 8	18
Figure 4	10	Figure 9	19
Figure 5	12	Figure 10	21

1 The gate and the watch

An evaluation answers whether a named build met a stated bar before it was released. It cannot answer what the deployed system is doing now, on traffic nobody chose. These are different questions, and conflating them is the commonest mistake in this field.

A great deal of effort now goes into the gate, and rightly: a deployment decision needs evidence attached to deployed behaviour, produced independently of the team that shipped it, which stops being true when the build changes. However high one sets that bar, it is a bar about a single moment, the moment of release. What happens afterwards is a different instrument with different properties, and the rest of this paper is about that one.

The distinction matters because each failure mode is invisible to the other. A build that passed its gate can drift: a model update, a prompt edit or a changed tool schema moves behaviour with nothing in the codebase changing, and no test runs because no commit landed. Equally, a system under continuous observation can drift without anyone noticing, because nobody decided in advance what would constitute drift. Watching without criteria produces dashboards. Criteria without watching produce stale certificates. Figure 1 sets the two instruments against each other and marks the seam where one hands over to the other.

Enterprises have reached the same conclusion from the procurement side. Roughly three-quarters now require monitoring and failure alerting before an agent is allowed to deploy at all, and nearly two-thirds name insufficient observability as a barrier to wider adoption [3]. The requirement is no longer contested. What the instrument actually delivers still is.

THE GATE · BEFORE RELEASE

A named build, a fixed suite

Cases the assessor constructed. Criteria fixed before testing. Every criterion resolved separately. One verdict, bound to a fingerprint.

Answers: may this build ship?

THE WATCH · AFTER RELEASE

Live traffic, continuously

Inputs nobody chose. No coverage claim is possible. Scored as it arrives, within a budget. No verdict: a signal, and a trend.

Answers: is it still behaving?

release

Why the distinction is load-bearing

A build that passed its gate can drift: a model update, a prompt edit or a changed tool schema moves behaviour with nothing in the codebase changing. A system under observation can drift without anyone having decided what drift is.

Neither instrument substitutes for the other, and a platform that offers one while implying the other is the failure this paper is about.

Figure 1: The two instruments, and the seam between them. A gate produces a verdict about a build; a watch produces a signal about a deployment.

What this figure does not show: that observability is the weaker of the two. It is the only one that sees real traffic.

2 How agents actually fail in production

Before deciding what an instrument should watch, it is worth knowing what goes wrong. The published distribution is not what most evaluation design assumes.

The largest public study of the question analysed 91,747 production tool calls from coding agents and found a 4.12% hard failure rate, with the ten commonest modes accounting for 77% of all failures [4]. The top six are given in Figure 2. Context budget exceeded is the single largest at 17.2%, followed by a resource that could not be found at 15.6%, then timeouts, runtime errors, stale context and missing preconditions.

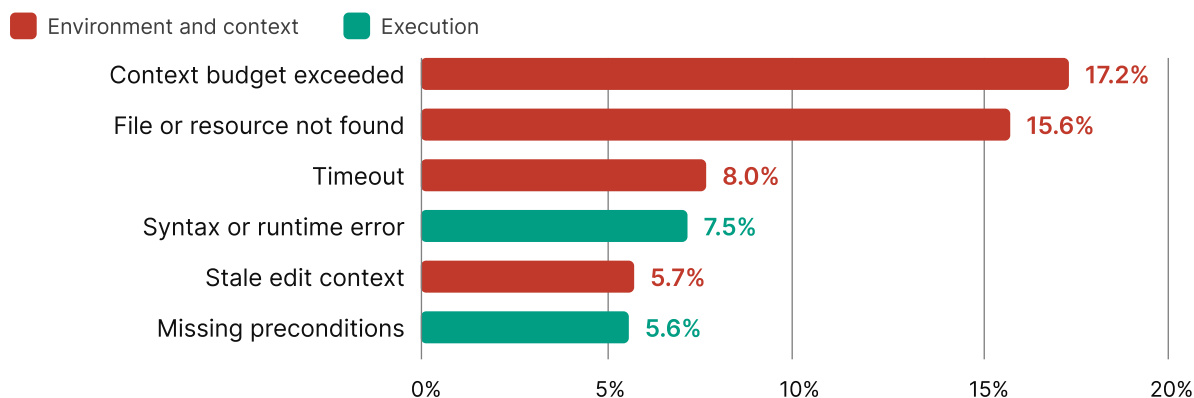
Read the labels rather than the numbers. Not one of the six is a failure of the model's reasoning. They are failures of environment, context and execution: the agent ran out of room, the file was not there, the call did not return, the state it assumed had moved. The same study found stateful tools failing roughly five times more often than stateless ones, which is the same finding seen from another side.

Independent reporting agrees on the shape. Tool calling fails somewhere between 3% and 15% of the time depending on model and task; task termination problems account for over a fifth of observed failures; and around 8% of cases show recursive tool or agent calls that continue until an iteration limit stops them [5].

This has a direct consequence for instrument design. A failure of reasoning might be visible in the answer. A context budget exhausted mid-run, a tool that returned nothing, a loop that ran eleven times, are not visible in the answer at all. They are visible only in the trace, and only if the trace records tool calls and their return values. An instrument that reads outputs is looking in the one place where this distribution does not show up.

WHERE PRODUCTION AGENTS ACTUALLY FAIL

Share of all hard failures, from 91,747 production tool calls (4.12% hard failure rate)



None of the six is a failure of the model's reasoning.

Figure 2: The six commonest hard-failure modes in 91,747 production tool calls. Environment and context dominate, and none of the six is a reasoning failure [4].

3 What a trace is, and which standard it follows

A trace is the structured record of one agent run: the question, the retrieval it performed, the tools it called with what arguments, the model turns, and what came back from each.

Two specifications describe that record and they are not the same one.

OpenTelemetry's generative-AI semantic conventions extend the general tracing standard with model-specific spans and metrics, giving a vendor-neutral schema [1]. OpenInference, which grew out of the observability tooling itself, defines fourteen agent-aware span kinds including AGENT, RETRIEVER, TOOL, LLM, GUARDRAIL and EVALUATOR [2], and is the richer of the two for agent work specifically.

Neither has won. As of early 2026 the OpenTelemetry generative-AI conventions remain in development status rather than stable, while OpenInference is in production use across several platforms. There is no announced convergence. A team instrumenting an agent today is choosing between a standard that is not finished and a standard that is not neutral, and a reader should treat any claim of portability with that in mind.

Everything described in this paper emits OpenInference span kinds and exports them over OTLP to Arize, with an optional second exporter to a local Phoenix collector in development. That choice is practical rather than principled: it is what the receiver ingests, and a span shape the receiver does not understand is not telemetry, it is noise.

4 Runtimes that emit nothing

The instrumentation literature assumes the agent runs in your process. A large and growing share of production agents do not.

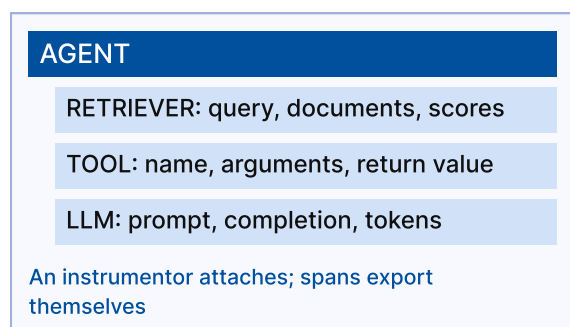
When an agent is built on an open framework in your own runtime, instrumentation is close to free: an instrumentor attaches to the framework, spans are emitted as the agent executes, and the platform receives a full tree without the developer writing any telemetry code. Almost every published integration guide describes this case, which is the left half of Figure 3.

The agents we were asked to observe did not work that way. The DevRev agents run server-side inside DevRev's own runtime, reached through a single synchronous call. Glean agents run inside Glean. ElevenLabs voice agents run inside ElevenLabs and answer over a telephone call. None of the three exposes a hook to attach an instrumentor to. The endpoint returns the final answer and nothing else: the retrieval step, the skill invocations and the model turns happen inside the box and are never surfaced. Where the vendor collects telemetry of its own, it goes to the vendor's endpoint, and there is no supported path to redirect that stream anywhere else.

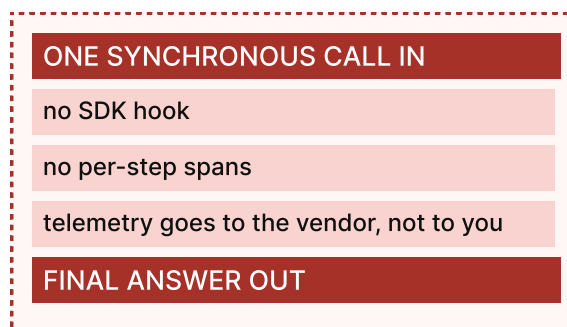
Set this against Section 2 and the problem is sharp. The failure modes that dominate production are exactly the ones that happen inside the box: the exhausted context, the tool that returned an error, the loop that retried six times. All of it leaves no record. The platform has nothing to ingest, so the dashboard is empty, and an empty dashboard reads as a healthy agent.

This is not a deficiency in the observability platform. It is a property of what the runtime is willing to say about itself, and no amount of configuration on the receiving end changes it.

OPEN FRAMEWORK · INSTRUMENTS ITSELF



CLOSED RUNTIME · EMITS NOTHING



The consequence

Every tool failure, every retrieval that returned nothing, every retry loop happens inside the closed box and leaves no record. The platform has nothing to ingest, so the dashboard is empty and the agent looks healthy.

This is not a gap in the observability platform. It is a gap in what the runtime is willing to say about itself.

Figure 3: The instrumentation asymmetry. Frameworks that run in your process can be instrumented by attaching to them; a runtime that executes the agent on someone else's infrastructure cannot.

5 The bridge

The answer we built, repeatedly, against three unrelated vendor runtimes, is the same shape: stop trying to instrument the agent, and instrument the call instead.

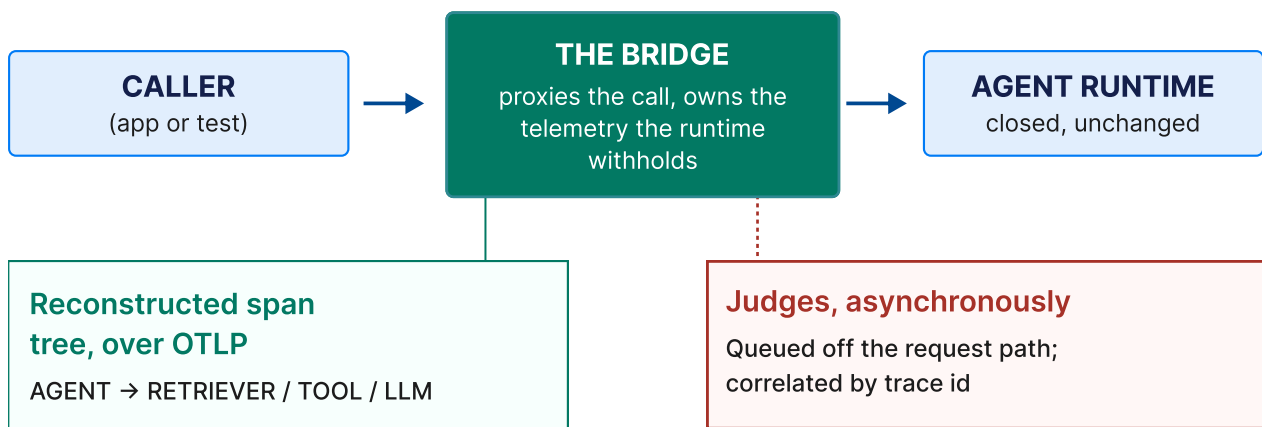
A thin service sits in front of the agent. It receives the request, forwards it unchanged, and returns the response in real time, so from the caller's side nothing has changed. Around that passthrough it owns the telemetry the runtime withholds: it times the call, captures the answer and whatever structured material comes back with it, and builds a span tree in the shape the platform expects, with the agent span as parent and retrieval, tool and model turns beneath it. Those spans are exported over standard OTLP into a tracing project of their own. Figure 4 gives the pattern; the two properties below are what make it usable rather than merely ingenious.

The agent is not modified. The bridge sits in front of a runtime it does not control, and that matters for a reason stronger than convenience: a system that has to be changed before it can be assessed is no longer the system that is deployed.

Judging happens off the request path. Scoring is queued to background workers and joined back to the trace by its identifier afterwards. A judge that adds latency to a production request is a judge that gets switched off within a week, and the honest way to avoid that is architectural rather than a matter of choosing a faster model.

What a bridge cannot do is also clear, and we state it on the figure rather than in a footnote. The span tree is reconstructed from outside, not captured from within. It is an accurate account of what an external observer could see, and correspondingly thinner than a native trace: the agent's internal reasoning steps are absent, because the runtime never emitted them and inventing them would be worse than their absence.

Sections 6 to 8 are what that pattern looks like once it is built: the production stack, the six span trees it produced, and the connectors that carry the same traffic into an evaluation engine.



Two properties that make this usable rather than clever

The agent is not modified. The bridge sits in front of a runtime it does not control, so the thing observed is the thing deployed.

Judging is off the request path. Scoring runs in background workers and is joined to the trace afterwards, so observation does not slow the thing it observes. A judge that adds latency to production is a judge that gets switched off.

The span tree is reconstructed, not captured.

It is an honest account of what the bridge could observe from outside, and it is correspondingly thinner than a native trace.

Figure 4: The bridge pattern in the abstract. Sections 6 to 8 are what it looks like built.

What this figure does not show: internal reasoning steps. A reconstruction cannot invent detail the runtime never emitted, and does not pretend to.

6 The production stack

A service in front of a DevRev agent, exporting OpenInference spans to Arize over OTLP and running its judges on a queue. It is the reference implementation of Section 5 and the one that has carried real traffic.

The request path, shown across the top of Figure 5, is deliberately dull. A caller posts a query; the proxy forwards it to DevRev's agent endpoint and returns the answer. Around that it adds the things a production edge needs and an agent runtime does not provide: a correlation identifier threaded through every log line, a filter that strips personal data before anything is written, a request-size cap, per-key rate limiting and security headers.

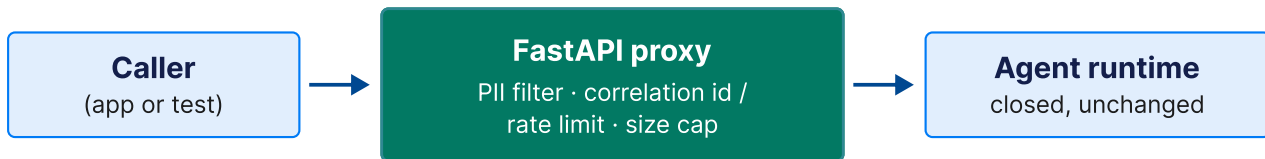
The tracing side is a tracer provider whose resource carries the service name, version, deployment environment and the Arize project name, and a batch span processor exporting to the Arize OTLP endpoint over HTTP with the API key and space passed as headers. In development a second processor exports the identical spans to a local Phoenix collector, so the same trace can be read without a cloud round trip. The span tree uses five of the fourteen OpenInference kinds: AGENT at the root, with RETRIEVER, TOOL, LLM and CHAIN beneath it.

Evaluation runs on background workers behind a queue, never on the request. Five judges execute concurrently against one trace: hallucination, groundedness, answer relevance, context relevance and a response-quality score, with a separate judge for tool-call quality. Their thresholds are configuration rather than opinion, which is what lets a reader argue with them: hallucination fails above 0.20, groundedness fails below 0.65, relevance warns below 0.60, and response quality passes at 7.0 out of 10.

One detail carries more weight than its size suggests. Results are written back with the prediction identifier set to the trace identifier, so an evaluation is attached to the trace that produced it rather than accumulating in a parallel table. A score that cannot be navigated back to the exact exchange that earned it is a number in a report, and the first time somebody disputes it there is nothing to show them.

THE REQUEST PATH · REAL TIME

REQUEST PATH



Return-path: the answer returns unchanged, at the same latency the caller had before

WHAT THE PROXY OWNS · OFF THE REQUEST PATH

1. Spans the runtime never emitted

AGENT over RETRIEVER / TOOL / LLM / CHAIN — five of the fourteen OpenInference span kinds. Batched out over OTLP/HTTP to a tracing project of its own.

2. Judges, queued

Five judges run concurrently on a worker: hallucination, groundedness, answer relevance, context relevance, response quality. Joined back by prediction id = trace id.

Thresholds are configuration, not opinion

Hallucination

Fails above 0.20

Groundedness

Fails below 0.65

Relevance

Warns below 0.60

Response quality

Passes at 7.0 of 10

A judge that adds latency to a production request is a judge that gets switched off within a week. Keeping it off the request path is an architectural choice, not a matter of picking a faster model.

Figure 5: The production stack: a proxy in front of a DevRev agent, OpenInference spans to Arize over OTLP, and five judges on a queue joined back to the trace by identifier [7].

7 Six agents, and what their traces exposed

The pattern was applied six times across three vendor runtimes. What is worth reporting is not the shape of each span tree but what each one made visible, because in every case it was something the agent's own answer did not disclose. Figure 6 sets the six side by side; the paragraphs below take them in turn.

An education-feedback agent returned fluent, well-formed feedback on every request. Giving each attempt a trace of its own showed that the identical submission scored full marks on roughly half the attempts and reported that it could not find its rubric on the rest. No single answer indicated which had happened, and an evaluation that averaged across attempts would have reported a mediocre agent rather than an unreliable one.

A business-intelligence agent that writes and executes SQL produced an answer and a table. Separating generation from execution showed the generated query changing between runs of a question that had not changed, which is a different defect from a wrong answer and is fixed in a different place.

A quality-audit agent is a router over five skills, and its reply never says which skill produced it. Carrying the routing decision as its own span is what makes a misrouted request visible at all; the answer to the wrong question can be perfectly well written.

The Glean agent is an enterprise retrieval agent whose retrieval is permission-aware. Carrying its citations as retrieval results rather than as prose lets a scorer see that a claim had no cited source behind it, and lets the same question be asked as several users to show where the answer moves with the caller's entitlements.

The ElevenLabs agent is not a text agent at all. Timing transcription, decision and speech as separate spans locates where in a call a required disclosure went missing, rather than establishing only that it did. In a deliberately degraded test agent this recovered five real compliance failures that the call audio alone would have required a human to find.

The sixth is the live proxy of Section 6, and it found the plainest failure of the set: a retrieval tool returning an error on every call while the assistant answered every question confidently and signalled nothing. That is the distribution of Section 2 showing up exactly where it was predicted to, in the one place an output-only check cannot look.

Across all of them one rule holds. A span the runtime did not emit carries a reconstructed attribute, so the distinction survives into the platform rather than living in a README, and where a routing decision was inferred by our scorer rather than reported by the agent the span says so. The instrument records what it actually observed and marks the rest, which is the standard this paper asks of the agents it watches.

SIX AGENTS, THREE RUNTIMES, AND WHAT THE TRACE EXPOSED

the right-hand column is the point: each of these is invisible in the answer the agent returned, and visible in the trace it never emitted

Agent and runtime	What the runtime emits, in full	What the bridge recovers	What that made visible
Education feedback DevRev	The final feedback text	One span per skill, and each attempt in a trace of its own	The same submission scored full marks on some attempts and found no rubric on others
BI, text-to-SQL DevRev	An answer and a table	Generation separated from execution and interpretation	The generated query changed between runs of a question that never changed
Quality audit router DevRev	One reply, from whichever skill ran	The routing decision carried as a span of its own	Which of five skills answered, which the reply itself never revealed
Enterprise search Glean	An answer with citations inline	Citations as retrieval results, one span per source	A claim with no cited source behind it, and answers that move with the caller's rights
Voice collections ElevenLabs	An audio recording	Transcript, decision and speech as separately timed spans	Where in a call a required disclosure went missing, not merely that it did
Support, live traffic DevRev , via the proxy	Nothing beyond the reply	Live tool calls with their arguments and return values	A retrieval tool failing on every call while the reply still read as confident

The second column is the complete native telemetry in each case: the reply, and nothing beside it. No client is named and no runtime was modified; a span the runtime did not emit carries a reconstructed attribute, so a reader can always tell an observation from an inference.

Figure 6: Six production agents across the DevRev, Glean and ElevenLabs runtimes: what the runtime emitted, what the bridge recovered, and the failure that became visible as a result [7, 8].

What this figure does not show: any of these failures being detectable in the answer the agent returned. That is why the column exists.

8 Connectors into the evaluation engine

The bridge produces traces for a monitoring platform. The same three runtimes also had to be reachable from the evaluation engine, behind one adapter seam that is transport and nothing else.

DevRev. The endpoint is internal and undocumented, and its wire contract had to be established empirically, because every wrong assumption failed as a bare 400 that looked exactly like a bad credential. Three findings were load-bearing. The Accept header must ask for JSON even though the body is a stream of server-sent events, because the endpoint refuses to negotiate as the thing it actually returns. The session object sits at the top level of the request; placed inside the event or the message it is rejected as invalid and simultaneously reported as missing. And the message is a plain string with no type discriminator. The session identifier is also the conversation key, which means reusing one string across calls is what gives the agent multi-turn memory, so an evaluator that wants independent attempts has to vary it deliberately. Independence between attempts is a precondition for measuring consistency at all, and here the whole requirement reduces to one line of a payload.

Glean. The useful property of a Glean agent is that retrieval is permission-aware, and an evaluation that ignores that tests a system nobody uses. The connector runs with a global token plus a header naming a specific user, so the agent surfaces only documents that user is entitled to see, and the same question can be asked as several personas. Citations are mapped into tool calls so the groundedness and traceability scorers see sources the same way they do for a DevRev skill call. The instance name is validated against a strict pattern and the resolved address is pinned with redirects disabled, so a crafted instance cannot send the bearer token somewhere else.

ElevenLabs. Voice is not text, and the honest seam here is a refusal. A spoken reply scored by a text evaluator produces a plausible number about the wrong thing, so the puller routes voice sources to the voice engine instead and stores the resulting observation already scored, with its own evaluator version recorded, so the text loop never picks it up. With no live credentials the connector refuses to run rather than falling back to bundled sample recordings, because a complete-looking report about an agent that was never called is the worst output this system can produce.

That refusal is the general rule rather than a special case. Every connector is transport only and fails loud: an unreachable agent yields an unreachable observation, never a fabricated healthy reply, and never the expected answer standing in for the real one.

Traffic also moves the other way. For teams already emitting OpenTelemetry, the engine ingests their spans directly, mapping the OpenInference convention onto the same observation row a pushed trace produces, so the two paths are indistinguishable downstream. Tenant and agent identity are taken from the authenticated token and never from the span, so a span cannot claim to belong to someone else.

9 Four silent successes

The integration work produced one finding worth more than the rest. Four separate times the export reported success and delivered nothing, and in each case the only visible signal was a green one. Figure 7 lists them with the probe that exposed each.

The first was a name collision. Traces were accepted, the endpoint returned 200, and no tracing project ever appeared, because the name had already been claimed by a prediction model in the same workspace and a generative trace project cannot share it. Nothing in the response said so.

The second was a confusion between two ingestion paths that look alike from the caller's side. One writes a prediction row, a set of features and a label. The other sends OpenInference spans and is the only one that renders an agent graph. Early work used the first and produced a dashboard showing a model with no trace tree anywhere, which reads as an integration that is working.

The third was transport. The gRPC path failed with deadline-exceeded in one network environment while HTTP worked, and the default batching then hid whatever failures remained, because a batch processor swallows an export error by design. Exporting synchronously during bring-up is what made the remaining failures visible at all.

The fourth was the verification method itself. The 200 is written by a gateway before ingestion completes, and the console lags by minutes in both directions, so neither is evidence. Only querying the workspace through its API establishes that a project exists and holds spans.

We report these because the pattern is the one this paper exists to argue against, met in our own tooling rather than a customer's. A pipeline that cannot distinguish between delivered and accepted will eventually report health it has not measured. The same class of bug turned up in our own scorer: an exception handler returned a passing result, so a crash in our code reported a customer's agent as having passed its safety gates. The fix was to make the outcome tri-state, so a gate that could not be evaluated is neither a pass nor a failure, is surfaced as its own anomaly, and is excluded from the denominator rather than quietly improving the rate.

FOUR WAYS THE EXPORT REPORTED SUCCESS AND DELIVERED NOTHING

What we saw	What was actually true	What exposed it
A tracing project that never appeared	The name was already claimed by a prediction model in the same workspace, and a generative trace project cannot share it.	Listing the workspace API
A dashboard showing a model, not an agent	Two ingestion paths look alike. One writes a prediction row; only OpenInference spans render an agent graph.	The absence of any span tree
Exports that timed out in one environment	gRPC over HTTP/2 was blocked on that network. HTTP worked, and batching then hid the export failures that remained.	HTTP transport, batching off
A 200 that meant nothing had landed	The 200 is a gateway trailer, written before ingestion, and the console lags behind by minutes in either direction.	Query the API, never the console

None of the four raised an error. Each is the failure mode this paper argues against, met in our own tooling: a green signal standing in for a result nobody checked.

Figure 7: Four trace exports that reported success and delivered nothing, and what exposed each one [9].

10 What an observability platform does not solve

We ran a capability evaluation: eleven problems raised by the engineering and applied-AI teams actually building agents, mapped one by one against what the observability platform does. Figure 8 gives the result, under both of the two readings it admits.

Four were fully supported out of the box, and these are real: a scoring framework with model-as-judge and semantic similarity, an evaluator hub that replaces ad-hoc scripts, and versioned experiments. That is genuine capability and it displaced a meaningful amount of bespoke tooling.

Four more were supported only through evaluators the team would have to write themselves: date-window validity, context-aware output checking, schema and aggregation rules, business-rule violations. One required infrastructure outside the platform entirely, checking a generated query by re-running it against a database, which an observability platform cannot do because it cannot call a database. Two were not addressed at all, being execution-layer concerns (mocking and deterministic test harnesses) that the platform does not own and does not claim to.

There is a second reading of the same table, and it is the more instructive one. Our mapping counted four problems as fully supported. In review, the engineering team counted two. The difference is not arithmetic. They scored a capability as supported only when it required no evaluator of their own writing, and by that standard most of the work remained theirs. Their conclusion was that the platform is enabling infrastructure rather than a complete solution.

We report their number alongside ours because the disagreement is the finding. A platform evaluated by the people selling the integration and a platform evaluated by the people who will maintain the evaluators produce different scores on identical evidence, and the second score is the one that predicts whether the thing gets used.

One further limit is architectural rather than a matter of coverage, and it recurs across every platform in this category: an observability platform can score a business-rule violation after it has happened, but it cannot block the violation at runtime. Observation is not enforcement. A control that only reports is a control that arrives after the customer did.

ELEVEN PROBLEMS RAISED BY THE TEAMS BUILDING THE AGENTS

Each block is one problem; width is proportional to count

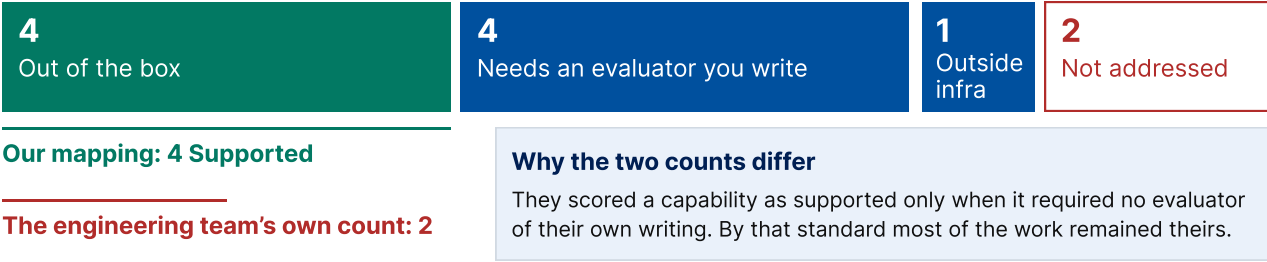


Figure 8: A capability evaluation of an observability platform against eleven problems raised by the teams building agents on it, with both readings of the same table [10].

11 The cost governor

The decision that determines whether continuous scoring is a product or a bonfire is not storage. It is how much of the traffic gets judged by a model.

The arithmetic is unforgiving and Figure 9 is simply that arithmetic drawn. A hundred thousand agent calls a day against the engine's eight prebuilt evaluators is 800,000 model calls a day, before a tenant adds one of its own. The line is straight because the cost is, and that is the whole problem: nothing about scoring everything gets cheaper with scale. Any design that judges every trace with a model has either not done this multiplication or is not expecting the traffic.

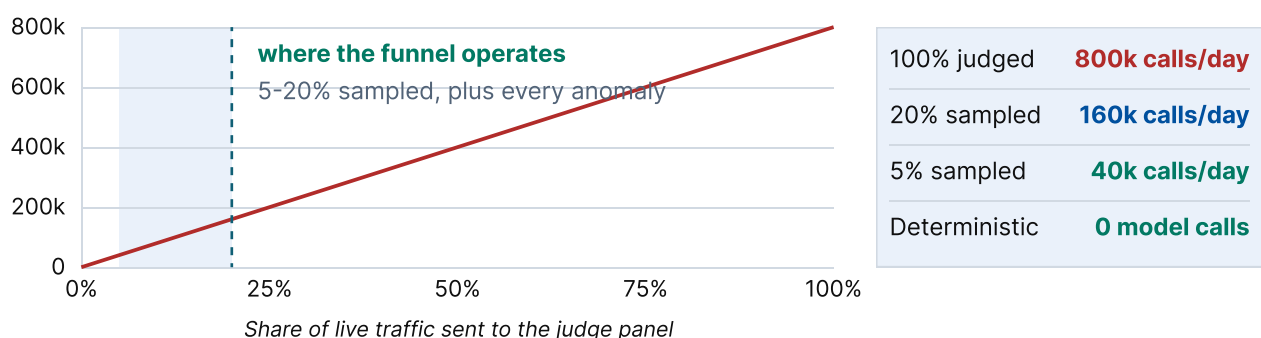
The structure that survives contact with real volume is a funnel, and the order of the layers is the design. Deterministic checks run on everything, cost effectively nothing, and catch a large class of failure on their own: banned phrases, brevity, language, latency, errors, tool calls that returned an error, replies that were empty. Section 2 is the argument for why that layer is worth more than it looks, since most of the dominant failure modes are detectable without a model at all.

Model judgement is then spent in two places only: on a configurable sample of ordinary traffic, and on every trace that failed a deterministic check. Anomalies are always judged in full, because that is where the tokens are worth spending.

Two further properties keep it honest. The continuous layer uses a cheaper judge than the deep on-demand runs, a defensible trade when its job is to raise a signal rather than issue a verdict. And the whole loop runs under a hard per-window token budget that stops and raises an alert when exhausted, rather than silently dropping the traffic it could not afford to score [11]. Silent sampling degradation would make the dashboard quietly stop representing the system, which is the specific failure this paper exists to argue against.

WHAT IT COSTS TO JUDGE THE STREAM

Model-judge calls per day at 100,000 agent calls across 8 evaluators



The line is linear because the cost is. That is the whole problem: nothing about scoring everything gets cheaper with scale.

Figure 9: The cost of judging the stream, which is the multiplication drawn. The funnel operates in the shaded band, plus every anomaly.

12 Detection, and which way round to build it

Issue detection in this category is usually built model-first: a model reads the traces and proposes what is going wrong. We built it the other way round, deliberately.

Transparent rules run first and carry the detections they can justify. A tool that returned an error on every call in a session, a span that ran far longer than its neighbours, the same call repeated with identical arguments, an empty response. Each of these is a fact about the trace, computable without a model, reproducible by anyone with the same data, and incapable of being hallucinated. A model is then used on top, to cluster and describe what the rules found, not to decide whether something happened [11].

The reason is accountability rather than cost. A detection a customer cannot reproduce is a detection they will eventually stop believing, and the first time a model-proposed issue turns out not to exist, every other issue on the board loses its standing too. Rules first means every item on the board can be traced to the specific span that produced it.

The failure distribution in Section 2 is what makes this practical rather than merely principled. Context exhaustion, missing resources, timeouts and loops are all deterministic to detect. The rules layer is not a cheap approximation of the model layer; on the modes that dominate production it is the more reliable of the two.

13 Direction of travel

Two projections, both analyst-published and both directional, establish that this category is being funded rather than debated. Both are plotted in Figure 10.

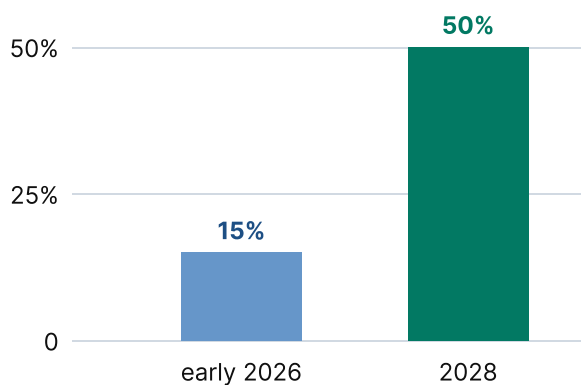
The clearer of the two concerns adoption rather than revenue: observability investment is projected to feature in half of all generative-AI deployments by 2028, against roughly 15% in early 2026. Whatever one thinks of the precision, a jump of that shape within two years describes a capability moving from optional to assumed.

The market projections point the same way, from a small base. The agent-observability segment specifically is put near 0.9 billion dollars in 2026, growing at 35.6% compound to around 14 billion by 2035; the broader platform market is larger and growing at a similar rate.

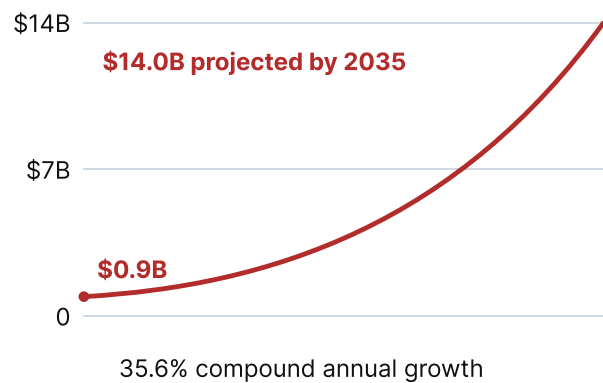
These numbers are reported here for one purpose and should be read for that purpose only. They establish that buyers are allocating budget, which is a claim about the category. They do not establish that the instruments being bought answer the questions in Sections 4 to 10, which is a claim about the products, and nothing in a market projection speaks to it.

THE DIRECTION OF TRAVEL

Generative-AI deployments with observability investment



agent-observability market, 2026 to 2035



Both figures are analyst projections and are directional. They are reported here to establish that the category is being funded, not to size it.

Figure 10: Adoption and market projections for the category. Both are analyst-published and directional [6].

14 What this does not do

Stated plainly, because a result of this kind is worth only as much as the limits published beside it.

A reconstructed trace is thinner than a native one. Everything in Sections 5 to 7 applies only to what an external observer can see. Where a runtime exposes its internal steps, a native integration will always carry more, and a reader comparing a bridge trace to a framework trace is not comparing like with like.

Sampling means coverage is a statistic, not a guarantee. Outside the deterministic layer, which does run on everything, any claim about continuously scored behaviour is a claim about a sample, and should be reported with its rate attached.

Observation is not enforcement. Nothing described here blocks anything. The instrument raises a signal; whether that signal stops traffic is a decision made elsewhere, by someone with the authority to make it.

The failure distribution is from coding agents. The study in Section 2 covers one agent class. The shape of the finding, that environment and context dominate over reasoning, is consistent with the broader reporting, but the specific percentages should not be transferred to a customer-support or retrieval agent without checking.

The corpus is small. Six agents across three runtimes, and a capability evaluation covering eleven problems from one engineering organisation. That is not a population. It is reported because the detail is specific and checkable, not because the sample supports generalisation, and the four silent successes in Section 9 are findings about one vendor's ingestion path rather than about the category.

15 Conclusion

Evaluation and observability are frequently sold as one thing and are not. The gate answers whether a build may ship; the watch answers whether the deployment still behaves.

A programme that has only the first issues certificates that outlive their subject. A programme that has only the second accumulates traffic nobody has a standard to judge. Both failures are common, and neither is visible from inside the other.

What the work reported here suggests is that the hard part of the second is not the plumbing. Span ingestion, storage and analytics are solved, standardised and in several cases free. The hard part sits on either side of the platform: upstream, in runtimes that emit nothing and have no obligation to change, which is why Sections 5 to 8 exist at all; and downstream, in the evaluation content that decides what a trace means, which by the engineering team's own count is where most of the work still remains.

The failure distribution points at the same conclusion from a third direction. If the dominant production failures are exhausted context, missing resources, timeouts and loops, then the instrument that matters most is the one that records what the agent did, cheaply and on everything, and the model is for interpreting what the record already shows.

Section 9 is the warning attached to all of it. An observability pipeline is itself a system that can report success without delivering it, and it deserves the same suspicion this paper asks everyone to apply to the agents it watches.

About this work

Author. Pratham Chintrate designed and built the trace bridges and connectors, ran the capability evaluation and wrote this paper.

Contributors. Abilash Narahari and Megha Melgiri supported and reviewed the work reported here.

Naming. Agent runtimes and the observability platform are named, because an integration finding that does not say which system it concerns can be neither checked nor reused. No client is named and no customer data appears.

Scope. This paper reports engineering work, one capability evaluation and published third-party data. It is not a controlled study, the corpus is six agents assessed by one practice, and it makes no statistical claims of its own.

16 References

1. OpenTelemetry. Semantic Conventions for Generative AI Systems. Development status as of March 2026.
2. Arize AI. OpenInference: semantic conventions for LLM and agent tracing. Fourteen span kinds including AGENT, RETRIEVER, TOOL, LLM, GUARDRAIL and EVALUATOR.
3. Survey of enterprise agent deployment practice, 2026: 72.7% require monitoring and failure alerts before deploying an agent; 63.4% cite insufficient observability as a barrier to wider adoption. Vendor-published; directional.
4. Failure analysis of 91,747 production coding-agent tool calls, 2026: 4.12% hard failure rate; top ten modes account for 77% of failures; stateful tools fail roughly five times more often than stateless.
5. Practitioner reporting on production agent failure modes, 2026: tool calling fails 3% to 15% depending on model and task; task termination accounts for over 21% of failures; about 8% of cases show recursive calls to an iteration limit. Directional.
6. Analyst projections, 2026: observability investment in 50% of generative-AI deployments by 2028, from about 15% in early 2026; agent-observability market near 0.9bn USD in 2026 at 35.6% compound growth. Vendor and analyst published; directional.
7. Internal implementation, 2026: trace proxy in front of a DevRev agent, OpenInference spans exported to Arize over OTLP/HTTP, five concurrent judges on a background queue, evaluations joined to traces by identifier.
8. Internal implementation, 2026: agent connectors for the DevRev, Glean and ElevenLabs runtimes behind one transport-only adapter seam, with the OpenInference ingest path in the reverse direction.
9. Internal engineering record, 2026: four trace exports that returned success while delivering nothing, and the workspace API queries that exposed each one.
10. Internal capability evaluation, 2026: eleven agent-development problems mapped against an observability platform's response, with the engineering team's own re-reading recorded alongside.
11. Internal design note, 2026: observability layer architecture, cost governor and detection model.

About QKTech

QKTech, formerly QualityKiosk Technologies, helps businesses win in the AI era by engineering reliability. Backed by experience in delivering reliable software, products, and platforms for leading global enterprises, QKTech combines deep domain knowledge with strong technology skills to help customers deliver great experiences to their own customers in an AI-first world.

Founded in 2000, QKTech offers reliability, cloud, AI, and product engineering services that help organizations build and adopt AI effectively, spanning product development, quality and reliability engineering, digital assurance, CloudOps and automation. At the core of this is the AI Reliability OS, an integrated portfolio of services covering AI for Reliability, Reliability of AI, Shift Right testing, Reliability of Frontier Systems, and Agentic Engineering. It brings together context engineering, harness engineering, loop engineering, observability, and platform-based delivery.

Operating in more than 25 countries with offices in Princeton and San Jose in the US, as well as Dubai, Singapore, Manila, Hong Kong and Jakarta, QKTech helps leading brands in banking, capital markets, insurance, automotive & manufacturing, technology & digital natives, telecom, airlines, consumer goods, and healthcare to transform their digital capabilities. The company is headquartered in Mumbai, India, with engineering centers across Bengaluru, Hyderabad and Pune.

The company's expertise and achievements have garnered recognition from esteemed global analyst firms such as Gartner, Forrester, and The Everest Group for innovative, IP-led solutions and the significant value of those solutions delivered to clients. QKTech has partnered with the most innovative and disruptive technology companies to force multiply outcomes that can be delivered to the customers.

For more information, visit www.qktech.com