

AN EXECUTIVE PERSPECTIVE ON SUPPORT OPERATIONS

AI-Native Support Operations

How enterprises are moving L1 to agents so engineers work on the novel — without giving up governance, auditability or control.

For CIOs, heads of service management and platform leaders accountable for incident response across a growing estate, who are being asked to adopt AI and need a use case that survives an audit, an outage and a board review.

EXECUTIVE SUMMARY

Most incident time is not diagnosis. It is assembly.

Ask an on-call engineer what they did during the last incident and they will describe diagnosis. Time the incident and you will see something else: most of the elapsed minutes were spent locating and reading information that already existed, in systems that were already running, about a service that was already instrumented.

That work is mechanical. **It has a correct answer that does not depend on who performs it, it is repeated identically on every occurrence of the same alert, and it is the single largest component of the number the business feels.** It is also, unlike most of the run budget, addressable without a migration, a reorganisation or a platform rewrite.

Many organisations are responding by adding AI assistants to the incident channel. The leading ones are discovering that the transformation is not about the assistant — it is about which part of the work a person should still be doing.

THE OLD QUESTION

"How do we resolve incidents faster?"

THE NEW QUESTION

"Why is a person assembling this at all?"

THE AI-NATIVE SUPPORT MODEL IN ONE SENTENCE

Agents handle L1 — triage, evidence and the routine resolutions — against runbooks held as data rather than prose, with every conclusion checked against the evidence that supports it before anything acts on it, and every action recorded in an audit trail that satisfies a reviewer rather than worrying one.

AGENTS

Execute the runbook in seconds, in parallel · before the engineer has opened a laptop

RUNBOOKS AS DATA

Typed, versioned, testable steps · not a wiki page that drifts one deploy at a time

VERIFIED CONCLUSIONS

Each claim names its evidence · an assumed link lowers the confidence and says so

The remainder of this paper sets out what is mechanical and what is not, the operating model that follows from the distinction, the reference architecture that implements it, and the sequence a CIO should use to adopt it.

1 • The Support Model Is Under Pressure

For thirty years enterprise support has been tiered, and for good reason. Expertise is scarce and expensive, so a broad, cheaper, always-available first tier filters the queue and only the incidents that genuinely need scarce expertise reach it. The economics are sound. Nobody has invented a better structure.

What has changed is the volume arriving at the front of that filter. Every new service, every migration, every acquisition adds alerting surface. **Estate growth is compounding; the tiering model responds to it linearly, by adding people.**

INDUSTRY DATA

Across published DevOps and service-management research the pattern is consistent: alert volume grows with estate size, and **a substantial share of pages resolve with no change made to anything**. Organisations that have measured their own queues typically find the benign share somewhere between a fifth and a third.

DORA State of DevOps · industry service-management benchmarks · and, more usefully, whatever your own ticket system says — the measurement is in section 7.

The tier boundary, not the tier

Time a real incident and the largest single block of elapsed time is usually not the investigation. It is the handoff — and the handoff carries two costs that per-tier metrics cannot see.

The evidence is gathered twice. L1 opens the dashboards, reads the logs, then writes three lines in the ticket. L2 receives three lines, cannot reconstruct what L1 saw from them, and opens the same tools again. The work is repeated because the first pass was never captured in a form the second person could use.

Context degrades at every hop. What L1 saw becomes a sentence; what L2 concluded becomes another. By the time it reaches an engineer the original signal has been through two rounds of lossy summarisation, and their first question is usually for something that was on L1's screen forty minutes earlier.

Every tier's handling time looks healthy. The handoff belongs to nobody, so it is measured by nobody — and it is the largest component of the number the business actually experiences.

There is a second, quieter effect. L1 escalates more than it should — not through lack of skill, but because the information needed to stop is not available at L1. The runbook says *check whether the error rate has recovered*; following that instruction requires access to the metrics tool, knowledge of which query represents "the error rate" for this service, and the judgement to read a bouncing number. When one of those is missing, escalating is the only safe action for the individual — whatever it costs the organisation.

2 · Anatomy of an Incident

Averages hide the shape. Below is one production incident, timed end to end: an error-rate page on a payments-adjacent service at 02:14, handled by a competent engineer following a runbook that existed and was roughly accurate. It closed in forty-one minutes, which most teams would call a good night.

ELAPSED	WHAT HAPPENED	PRODUCED
00:00–09:30	Wakes, connects, acknowledges. Reads the alert, finds the dashboard on the third attempt, confirms the rate is still elevated.	Still happening
09:30–13:00	Opens the logs tool, constructs a query, sees connection refused against a dependency.	A symptom
13:00–17:00	Checks the dependency. It is not alerting and its dashboard looks healthy.	A contradiction
17:00–26:00	Opens the deploy pipeline; finds a release forty minutes before the page. The diff changes a connection-pool setting.	A hypothesis
26:00–39:00	Posts in Slack; no reply at 2:40am. Rolls back, and watches the error rate fall below threshold.	A fix
39:00–41:00	Resolves the page. Writes two sentences in the ticket.	A thin record

Sorted by what each minute was *for*, rather than when it happened, the shape is stark. Roughly half the incident — twenty-one and a half minutes — was locating and reading information that already existed. Seven minutes were genuine judgement.

ASSEMBLY — 21.5 MINUTES

Mechanical, and repeated identically every time. Confirm the rate. Check the pods. List recent releases. Pull the error lines. Check the dependency. Six queries with correct answers that do not depend on who asks.

JUDGEMENT — 7 MINUTES

Consequential, and requiring accountability. Deciding that the pool change explains the errors, and that rolling back is safer than fixing forward at 2:40am. This is why you employ senior engineers.

The two-sentence ticket that closed it records neither — not the error rate, not that the dependency looked healthy from outside, which was the genuinely confusing part. When the same alert fires in five weeks a different engineer spends their own twenty-one minutes. Not through carelessness: writing that down at 02:55 is not a reasonable thing to ask of someone who has to work tomorrow.

3 · What Should Be Automated, and What Should Not

There are two failure modes in AI-for-operations programmes and they are mirror images. Automating too little produces a chatbot that summarises what everyone can already read. Automating too much produces a system that restarts a database at 3am on a model's say-so and is wrong. Both come from skipping the same step: separating the mechanical from the consequential.

THE DEMAND	ITS CHARACTER	WHAT FOLLOWS
Gather <i>What is true right now?</i>	Deterministic. Same inputs, same answer.	Automate without reservation. This is where most of the elapsed time goes.
Interpret <i>What does that mean?</i>	Probabilistic. Needs a stated confidence.	Automate, but only if the output shows its evidence and the confidence means something.
Decide <i>What should change?</i>	Consequential. Needs accountability.	Keep with a person for anything with real blast radius. "The model suggested it" is not an answer in a review.

The failure mode that ends pilots

An assistant reads the logs and reports: *"Root cause: database connection pool exhaustion. Confidence 92%."* The engineer checks. The pool is at forty per cent utilisation. Nothing in the evidence supported ninety-two per cent; the number was generated, not measured.

The damage is not the wrong answer — engineers are used to monitoring tools being wrong. The damage is that the confidence was fabricated. Once an engineer catches a system doing that they stop reading its output permanently, and they tell their colleagues. **Trust in this category is spent once.** So the question that matters is not how often it is right, but *how it behaves when it is wrong* — and whether it knew.

WHAT A DEFENSIBLE CONCLUSION LOOKS LIKE

Not a score. Three properties, each of which you can test in a pilot before signing anything.

CLAIMS CITE EVIDENCE

"A deployment landed inside the window — established by recent_deploys", not a bare percentage

ASSUMPTION CAPS CONFIDENCE

Where the causal link is correlation rather than demonstration, the number comes down and the reader is told why

GAPS ARE STATED

"A rollback that clears the errors would settle it" — worth more to the engineer than any score

A corollary vendors rarely volunteer: for much of this work a rule beats a model. Thresholds need arithmetic, not inference; known signatures should be encoded, not guessed; and anything a reviewer will question is better served by a deterministic rule you can point at. **A system that cannot run without a model has put a probabilistic component on a critical path.**

4 · Runbooks Have To Become Data

Every organisation has written its incident knowledge down. It is in the wiki, and it is out of date. This is not a discipline problem, and exhorting the team to maintain it harder will not work. Prose cannot be executed; what cannot be executed cannot be tested; what cannot be tested drifts from reality one deploy at a time. **The decay is structural.**

TRADITIONAL

A wiki page nobody trusts. *Written after an incident by whoever handled it, accurate on the day. Six months later the dashboard has moved, the query has changed and the service was renamed — and nobody knows which parts are still true.*

AI-NATIVE

A typed, versioned, testable asset. Steps with action types, parameters, conditions and variables. Match rules that bind it to alerts. A dry run that walks it against a sample alert and reports what would break — without executing anything.

The difference is not tooling preference. It is that the second form can be *tested*, and therefore relied upon. A runbook held as data can tell you, before you activate it, that step four references a variable nothing sets, that step six depends on a system this deployment has never been given credentials for, and that step two would page someone.

The knowledge-capture argument

This is where the operating-model case is stronger than the cost case, and it is the part CIOs tend to recognise immediately. The engineer who knows that this alert on that service is almost always the payment gateway is a single point of failure your org chart does not show. When they are on holiday the same incident takes four times as long; when they resign, the organisation loses years of tacit knowledge and discovers it only during the next outage.

The transformation, stated plainly. Executable runbooks move institutional knowledge out of individuals and into an asset the organisation owns. Every incident the agent handles adds to that asset rather than to somebody's memory — and the record it leaves is structured, so the second occurrence of a known incident is genuinely cheaper than the first. That is the saving most organisations assume they are already getting and almost none actually are.

5 • Not Every Alert Needs an Action

The most common wrong answer in automated remediation is doing something. A meaningful share of pages — consumer lag, queue depth, replication delay, batch backlog — need nobody to act. They need somebody to watch a number come down and confirm that it did.

Humans are poor at this and resent it. It occupies an engineer who cannot do anything else while occupied, it happens disproportionately at night, and the correct outcome is indistinguishable from having ignored it.

Patience, however, is mechanical.

TRADITIONAL

Page a person, who waits. The engineer refreshes a panel every few minutes for half an hour, then closes the incident with "recovered". No record of the readings, and the same alert does the same thing next week.

AI-NATIVE

The runbook watches, and parks. The execution samples the value, judges direction from the series' peak, estimates when it clears, and suspends itself between looks. If it clears, the incident closes with every reading attached. If it stops recovering, *that* is when a person is paged.



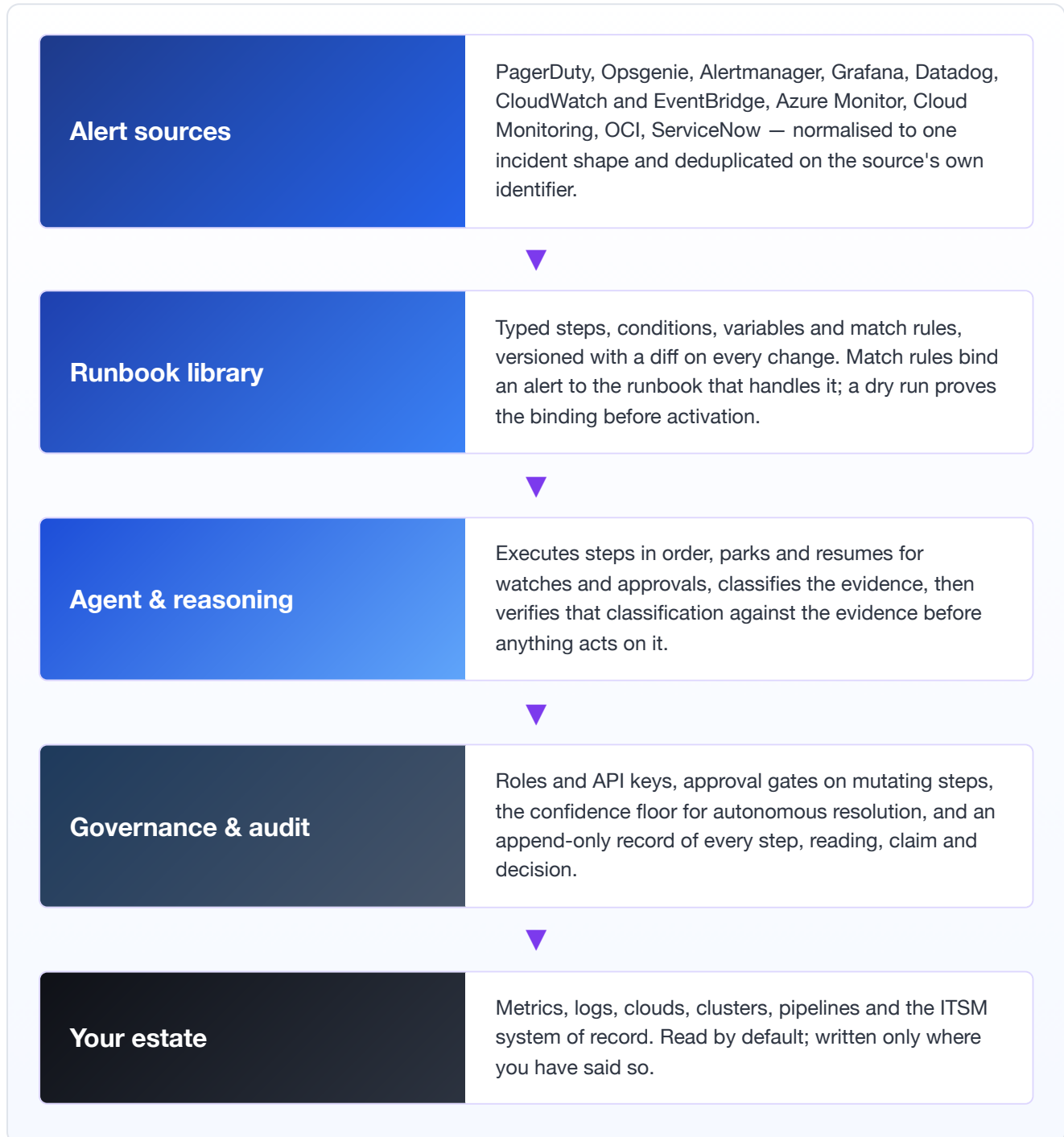
An incident the agent watched rather than acted on: three readings, the direction, the threshold it was waiting for, and the conclusion — "19855 is < 25000 — the condition has cleared on its own after 3 checks." No action was taken, and the readings stay on the record.

Blast radius belongs to you

Reading is safe; writing is not. Where the boundary sits is a decision for your change-management policy, not for a vendor's default. Three controls make it explicit: steps that change something outside the platform are marked as such and surfaced before activation; anything consequential can be gated behind a human approval that suspends the run until a named person answers; and the dry run rehearses a whole runbook without executing a single step, **so a rehearsal can never page anyone.**

6 • Reference Architecture

The shifts above come together in five layers. The top is where alerts arrive from whatever is already installed; the bottom is the estate that must not be put at risk. Every layer in between exists to translate the speed of the top into the safety of the bottom.



Two properties matter more than the boxes. **It runs inside your environment** — your cloud, your credentials, your model endpoint if you have one — so incident data does not leave. And **it degrades honestly**: a step whose system is not configured reports that and is skipped, rather than guessing something plausible to fill the gap.

7 · What Leaders Should Do Next

This is a programme of work, not a procurement decision, and it earns trust before it asks for it: read-only first, writing later, autonomy last.

HORIZON 1 First 90 Days	HORIZON 2 Next 180 Days	HORIZON 3 Next 12 Months
• Measure the three numbers below • Point one alert source at it, read-only • Convert the three most-fired runbooks to data • Compare what it concluded with what on-call did • Agree the confidence floor, and who owns it	• Let it close the clearly benign categories • Enable watch-and-wait for lag and backlog alerts • Add approval-gated actions for the safe ones • Bring the change calendar in for causation • Report coverage and escalation rate monthly	• Extend across clouds and acquired estates • Runbook authoring becomes a team habit • Make the evidence packet the handoff standard • Use pattern analysis to retire noisy alerts • Absorb estate growth without growing the rota

Measure your own starting point first

Three questions, answerable in a fortnight from systems you already have. Do not buy anything before you can answer them.

1 · What share of incidents reaching L2 were escalated without any change made at L1? This is the escalation the information gap caused, and it is usually the largest surprise in the exercise.

2 · What is the median gap between escalation and first action at the receiving tier? Measured separately from handling time. It belongs to no team, so nobody reports it.

3 · What share of incidents close with no change made to anything? If it is a third, you are paying full incident cost for alerts that only needed somebody to wait.

Then time three incidents by hand — an observation, not a survey. The assembly share from three real incidents will be the most useful number in your business case, and it will be yours rather than a vendor's.

Final Thought

The advantage here will **not** come from access to AI models. Every organisation has that.

It will come from an organisation's ability to turn what its best engineers know into **assets it owns** — executable, testable, versioned — and to let agents run them under governance the organisation sets.

The teams that manage this will absorb years of estate growth without growing the rota, and their engineers will spend their nights on problems that are actually new. The measure of success is not that AI resolved the incident. It is that **nobody had to assemble it first**.

L1 handled by agents. Engineers focus on the novel.

REPRESENTATIVE EXAMPLE · INSIGHTRESOLVE

Support teams spend most hours on triage and routine tickets. InsightResolve agents take L1: normalise every incoming alert, run the matched runbook, resolve the routine ones autonomously, and hand the rest over with the evidence, a classification and a review of that classification already attached.

TRADITIONAL

Engineers do the investigation by hand.

L1 triages each page manually; every one needs dashboards, log search, deploy history and a judgement about whether the number has recovered. Routine pages consume the majority of L1 and L2 hours; novel cases get the leftover attention.

AI-NATIVE

Agents do L1. Engineers handle the novel.

Every alert normalised, matched and investigated within seconds. Routine cases resolved autonomously with a full audit trail. L2 receives the evidence, the classification and what that classification rests on — not a service name and a threshold.

ALL THE AVAILABLE OPTIONS — AND WHERE INSIGHTRESOLVE FITS

ServiceNow AI Agents

PagerDuty AIOps

Datadog Bits AI

Atlassian Intelligence

Microsoft Copilot for Service

Moogsoft / BigPanda

Custom RAG agents

InsightResolve

Vendor add-ons work well inside their own ecosystem: if your estate is genuinely one platform, use theirs. Correlation platforms are strong at reducing alert noise but stop short of executing the investigation. Custom RAG agents start quickly and then become their own maintenance project. **InsightResolve is the pick when an enterprise wants self-hosted L1 automation that spans several monitoring stacks and clouds, uses its own credentials, runs on its own infrastructure, and exposes every action in an open audit log — with no vendor lock-in.**

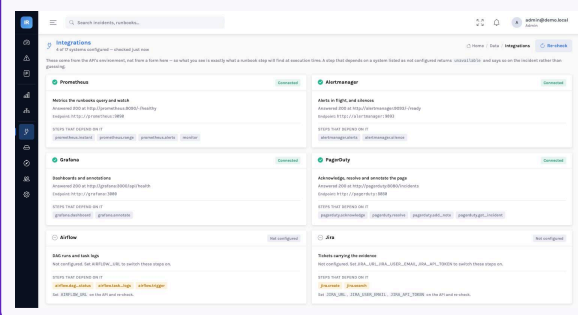
EXAMPLE IMPLEMENTATION

InsightResolve

L1 agents · Triage · Runbook execution · Verified root cause

- **Twelve alert sources** — paging, monitoring, four clouds and ITSM, normalised to one incident
- **Runbooks as data** — typed steps, conditions, variables, versions, and a dry run
- **Executing agent** — runs the steps, parks for watches and approvals, resumes by itself
- **Verified conclusions** — each claim cites its evidence; assumed links cap the confidence
- **Governance** — approval gates, a confidence floor you set, and a full audit record
- **Self-hosted** — your cloud, your credentials, your model endpoint or none at all

EVERY SYSTEM, AND WHETHER IT ANSWERS



CUSTOMER PATTERN · FINANCIAL SERVICES · MULTI-CLOUD ESTATE

A growing estate across two clouds, with L1 contracted out. Alert volume rose with each new service while the L1 contract was priced per ticket and per escalation. Normalising every source into one incident shape, and attaching the evidence before the ticket reached a human, moved both of the variables the contract was priced on — without renegotiating the contract or changing the tiering model.

A consumer group falls thirty thousand messages behind at 02:40.

REPRESENTATIVE EXAMPLE · INSIGHTRESOLVE · KAFKA CONSUMER LAG

Consumer lag is the archetypal alert that should not be acted on. The backlog is real, the service is healthy, and the correct response is almost always to watch it drain. Here is the runbook, and what the agent did with it.

STEP	ACTION TYPE	WHAT IT IS FOR
1 · Current lag	<code>prometheus.instant</code>	Establish the number the alert is about.
2 · Are the consumers alive?	<code>prometheus.instant</code>	A backlog with nobody consuming is a different problem from one that is draining.
3 · Watch the backlog drain	<code>monitor</code>	Sample the value, judge direction from the peak, park between looks, and decide.
4 · Write up what happened	<code>ai.summarize</code>	Leave a record the next engineer can use.

Step three is the one that matters. It does not block a worker while it waits: the execution is suspended with a resume time and picked up again by itself. Three outcomes are possible, and all three were verified against a live stack before this paper was written.

IT CLEARS

Resolved, with no action taken.

"19855 is < 25000 — the condition has cleared on its own after 3 checks. No action was needed." The readings stay on the incident, so the next occurrence starts from evidence rather than from nothing.

IT DOES NOT

Escalated, with the readings attached.

"Watched for 3 minutes over 3 checks and 53188 is still not < 10000; the trend is rising. This needs a person." The escalation carries the series that justified it.

There is a third case, and it is the one that exposes whether a system is honest: the source resolves the alert while the agent is still parked. The execution is closed out with the last reading recorded and an explanation that nothing further was needed — rather than being left waiting for ever on an incident that has already ended.

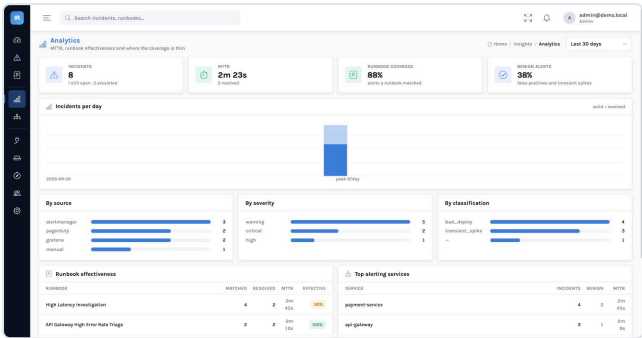
WHY THIS CASE IS THE BENCHMARK

Ask any vendor to show you this one. An agent that resolves incidents is easy to demonstrate. An agent that correctly decides to do nothing, records why, and hands over the moment the situation stops improving is a far better test of whether the system understands its own limits.

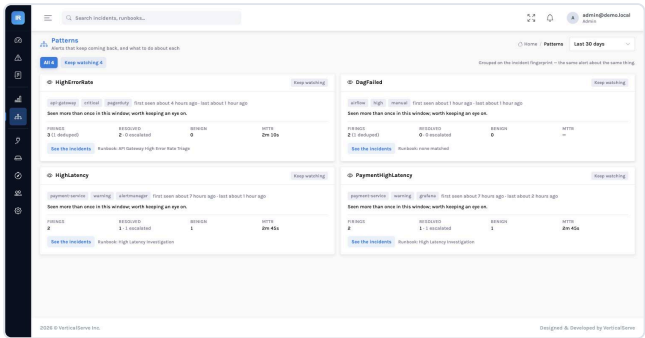
APPENDIX C · WHAT THE RECORD LOOKS LIKE

The artefact an auditor reads, and the numbers a board asks for.

REPRESENTATIVE EXAMPLE · INSIGHTRESOLVE CONSOLE



Every figure is a query over the incidents themselves, so the numbers cannot drift from the records behind them.



Recurring alerts grouped on the incident fingerprint, each with a suggested action and the rule that produced it.

Three properties of this record make the use case defensible to a risk function. It is **complete** — every step, parameter, reading and decision, not a chat transcript. It is **attributable** — each conclusion names the evidence establishing it and flags what is merely assumed. And it is **reversible** — the actions an agent may take on its own are bounded by a policy you set, and anything beyond that boundary waited for a named person.

The test to apply. Take five of your own resolved incidents and give the system only what was known at minute zero. Did it gather what your engineer gathered? Did it reach the same conclusion? *On the ones it got wrong, did its confidence reflect that?* And what did it do when a system was unreachable? The third question is the whole evaluation: a system that is right eighty per cent of the time and confident about the rest is unusable, while one that is right sixty per cent of the time and says so is something an on-call engineer will read.