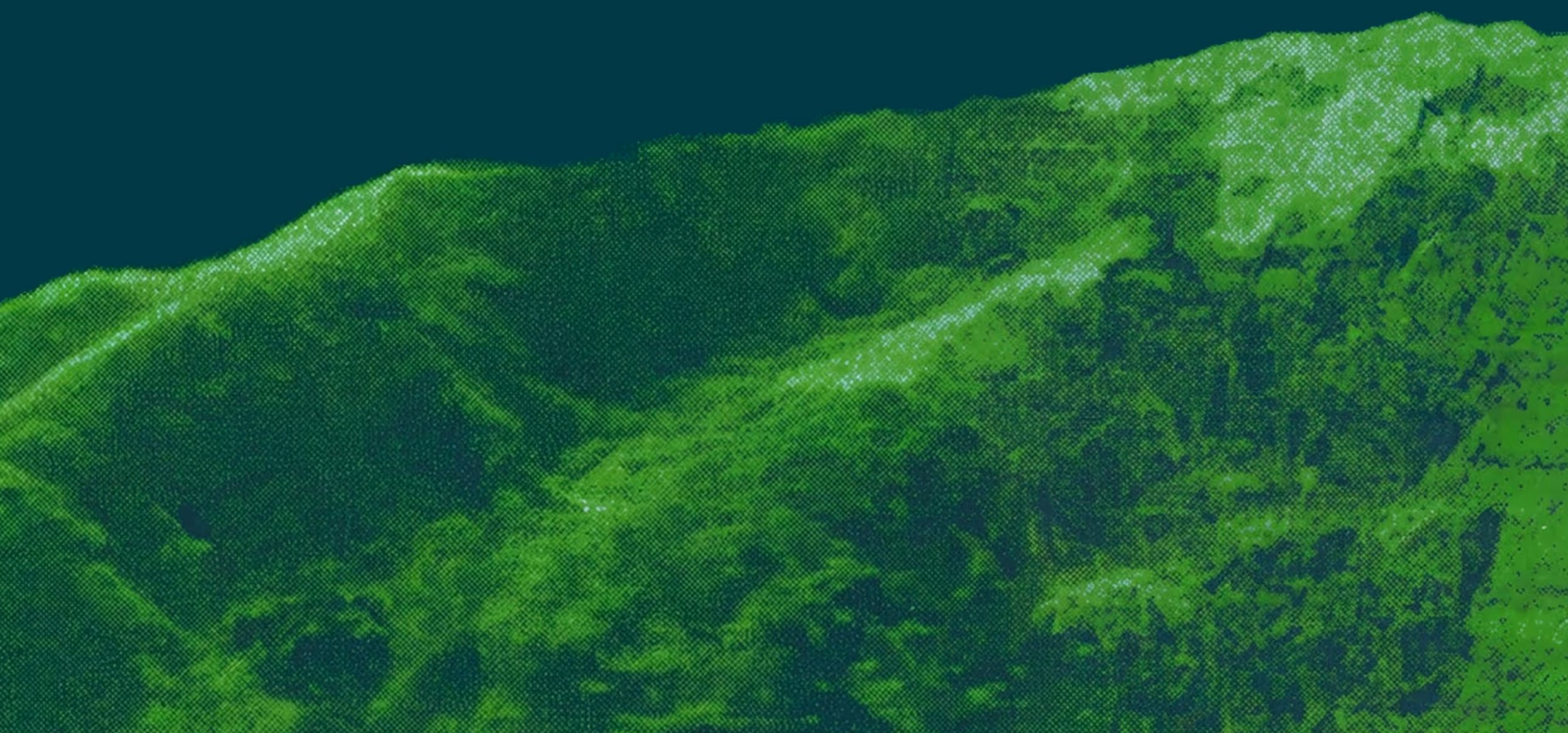
Traversal

The AI SRE Handbook

Operating Production Systems in the Age of AI

A TRAVERSAL HANDBOOK



Contents

Foreword 3

Preface 4

PART 01 THE GROWING CHALLENGE

A Brief History of Software Reliability 7

Complexity Outpacing Human Investigation Capacity 8

The Industry's Response: Instrument Everything. More Telemetry, Less Understanding 11

AI Is Accelerating the Curve 13

From RED to RADES 15

Limitation of AIOps: Correlation Is Not Causation 16

What Needs to Be Solved 18

PART 02 THE VISION FOR AI-NATIVE SOFTWARE RELIABILITY

The Self-Driving Maturity Ladder 20

What L5 Actually Looks Like 22

Closing the Outer Loop of the AI-Native SDLC 23

PART 03 WHAT IT TAKES TO GET THERE

How to Evaluate an AI SRE Platform 26

Why This Is a Hard Problem 28

Traversal's Approach to the 5 Evaluation Criteria 29

What the Architecture Unlocks 33

The Claude Code Moment for SRE 35

Reliability Is Still a Choice 37

Glossary 38

Index of Figures 39

Foreword

Software production operations have entered a new era. For decades, systems grew steadily more complex: each generation of computing added new layers of scale, abstraction, and operational distance between cause and effect. AI has accelerated that trajectory. The change is not merely that systems are larger or that cloud infrastructure is more dynamic. The deeper shift is that AI has changed both the nature and the pace of production operations: more code, assembled faster, with more integration points and more probabilistic components, arriving in production at a rate that outpaces the human workflows designed to operate it.

For more than two decades, site reliability engineering (SRE) gave the industry a rigorous way to manage this complexity. The discipline codified in the Google SRE Book introduced a practical language for balancing reliability and velocity: service level objectives, error budgets, toil reduction, automation, incident response, and postmortem learning. Those ideas remain foundational. They are among the most successful operating ideas the software industry has produced.

What has changed is the environment in which those methods now have to work, though "changed" undersells it. Systems have been growing more distributed, more interdependent, and more fluid for years. A single incident could already span application code, infrastructure, data pipelines, third-party services, feature flags, and deployment tooling a decade ago, and the evidence needed to explain it was already scattered somewhere across the system. Organizations have not been able to reason across all of it fast enough, consistently enough, or economically enough for a long time. This is not a new problem. It is one the industry has been slowly losing ground on for years, and AI has sharply steepened the slope: more code, more model behavior, more integration points, arriving faster than the workflows built to operate them.

AI is accelerating the problem. It is also, properly built, the way out of it. The same capability that lets systems generate and ship code faster than humans can track it can be turned on the problem it creates: reasoning across evidence at a speed and scale no human operator can match, holding a model of the production environment that never goes stale, and closing incidents before a war room ever needs to form. That is what this booklet explores: not how to cope with AI's added complexity, but how to put AI to work actually solving the reasoning problem the industry has been losing ground on for years.

That gap between the system's complexity and the human capacity to reason about it under pressure is what this booklet addresses. Its argument is that the next chapter of reliability engineering, in the age of AI, will be defined by how organizations put AI to work operating production systems once human troubleshooting is no longer the scaling factor.

Preface

This booklet is written for the people responsible for operating production systems at scale: engineering executives; infrastructure leaders; heads of platform, observability, and SRE; and the SREs, platform engineers, and on-call responders doing the work day to day. It stays loyal to the real language of operations: blast radius, paging load, Sev1 and Sev2 response, multi-hop failures, runbook drift, and the difference between symptoms and root cause.

Its subject is what it takes to operate production systems reliably in the age of AI, and how to tell whether a platform claiming to help has actually built what that requires.

The three parts are organized around that argument at different depths.

Part 1 lays out why the problem has gotten harder: too much telemetry, not enough reasoning, and AI accelerating both the complexity and the failure modes that existing tools weren't built to catch.

Part 2 describes what becomes possible once that gap is closed: a maturity progression from manual troubleshooting to self-driving production, and what each level actually requires.

Part 3 gets specific: how to evaluate an AI SRE platform, an architecture for building an AI SRE platform at enterprise scale, and what that architecture unlocks.

A reader evaluating whether to invest in this category will get the most from **Parts 1 and 2**. A reader responsible for implementing it will want to spend the most time in **Part 3**.

PART 01

The Growing Challenge

Why the problem has gotten harder, and why AI is accelerating a trend that predates it.

PART 02

The Vision for AI-Native Reliability

The progression from manual troubleshooting to self-driving production, and what each level requires.

PART 03

What It Takes to Get There

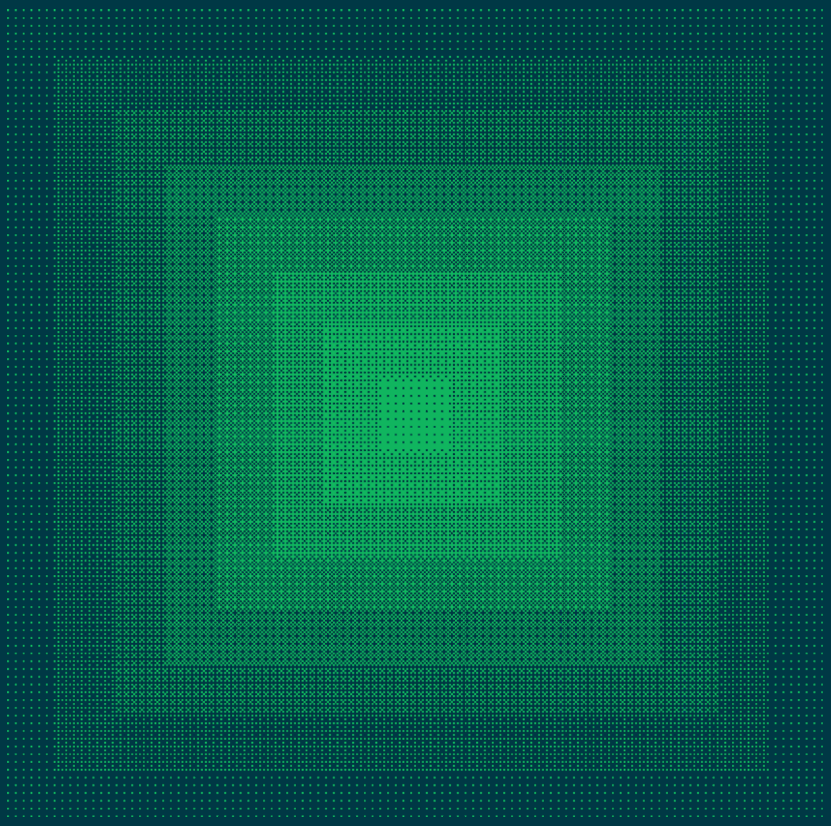
How to evaluate an AI SRE platform, the architecture behind one, and what that architecture unlocks.

Two terms recur throughout and mean different things. **SRE in the age of AI** is the discipline: practicing reliability engineering in an environment AI has fundamentally changed. **AI SRE** is the platform category: the agentic systems built to do core SRE tasks at scale: causal investigation, autonomous remediation, continuous reasoning over large-scale production environments. The distinction matters because an organization can face the challenges of the first without yet having deployed the second, and a poorly built second can fail to actually solve the first.

This is not a product manual, and it is not a claim that classic SRE became obsolete the moment AI entered the software stack. Its purpose is narrower and more practical: to describe what has actually changed in production environments, where the existing tools and methods have reached their limits, and what a credible operating model empowered by an AI SRE platform looks like in response.

The transition is no longer hypothetical. Gartner's 2025 Market Guide for AI Site Reliability Engineering Tooling forecasts that 85% of enterprises will use an AI SRE platform by 2029, up from less than 5% in 2025: one of the steepest adoption curves Gartner has projected in the operational tooling category.

This booklet is published by Traversal, the first enterprise AI SRE validated at Fortune 100 scale. The arguments here reflect what we have learned from production deployments at Nubank, Harvey, PepsiCo, and many more, where AI SRE has not only changed how incidents are diagnosed, but has reshaped on-call rotations, escalation pyramids, and the way operational knowledge is held inside the organization. The framework is not Traversal-specific, but it is informed by what works in practice at this scale.



PART 01

The Growing Challenge

SECTION I

A Brief History of Software Reliability

For more than two decades, site reliability engineering (SRE) gave the industry a rigorous way to manage the complexity of production systems. The discipline, first codified in the Google SRE Book, introduced a practical language for balancing reliability and velocity: service level objectives, error budgets, toil reduction, automation, incident response, and postmortem learning. It's important to state here that traditional SRE did not fail. It solved a set of foundational problems so effectively that much of the industry now treats the solutions as common sense.

What the Google SRE Book introduced, perhaps most importantly, was a more transparent relationship between systems and risk: the recognition that 100% availability is the wrong target for almost everything, and that the gap between your target and 100% is a budget to be spent, not a failure. Large-scale systems do not become reliable through aspiration. They become reliable through explicit objectives, repeated learning from failure, and engineering work focused on the failure modes that matter most. Alerting discipline, incident command, postmortem rigor, and automation became first-class engineering investments rather than afterthoughts.

Observability arrived as the next major advance, and it deserves the same credit. Metrics, events, logs, and traces (MELT) gave teams a richer basis for understanding system behavior than they'd ever had before. They reduced guesswork, accelerated local debugging, and made it possible to correlate symptoms across distributed architectures. In many organizations, observability platforms were the first tools that let operators move from anecdote to evidence during an incident.

Those contributions remain indispensable. Even now, organizations still need clear service expectations, fast and well-run incidents, meaningful postmortems, capacity planning, change discipline, and a bias toward automation over repetitive human work. The point of this booklet is not that SRE has become obsolete. The point is that the load now placed on it has grown beyond what its original, human-centered workflows were designed to absorb, and that this has been true for longer than most organizations have realized or been willing to admit.

SECTION II

Complexity Outpacing Human Investigation Capacity

Every generation of production engineering inherits a different operating environment, and each one asks more of the people running it than the last. As the Foreword argues, this isn't a new phenomenon AI introduced from nothing; it's a slow erosion the industry has been losing ground on for years. What changed is how steep the slope has become.

The number of services, dependencies, and interactions in a modern environment outpaced what a human being can reason about, and, as a direct consequence, what a human being can troubleshoot. A single customer request can traverse application services, queues, service meshes, feature flags, identity systems, caches, databases, third-party APIs, and background jobs before it completes. Each of those may have separate ownership, separate telemetry, and separate operational vocabulary. Few of them can be understood in isolation during a live incident, and increasingly, none of them can.

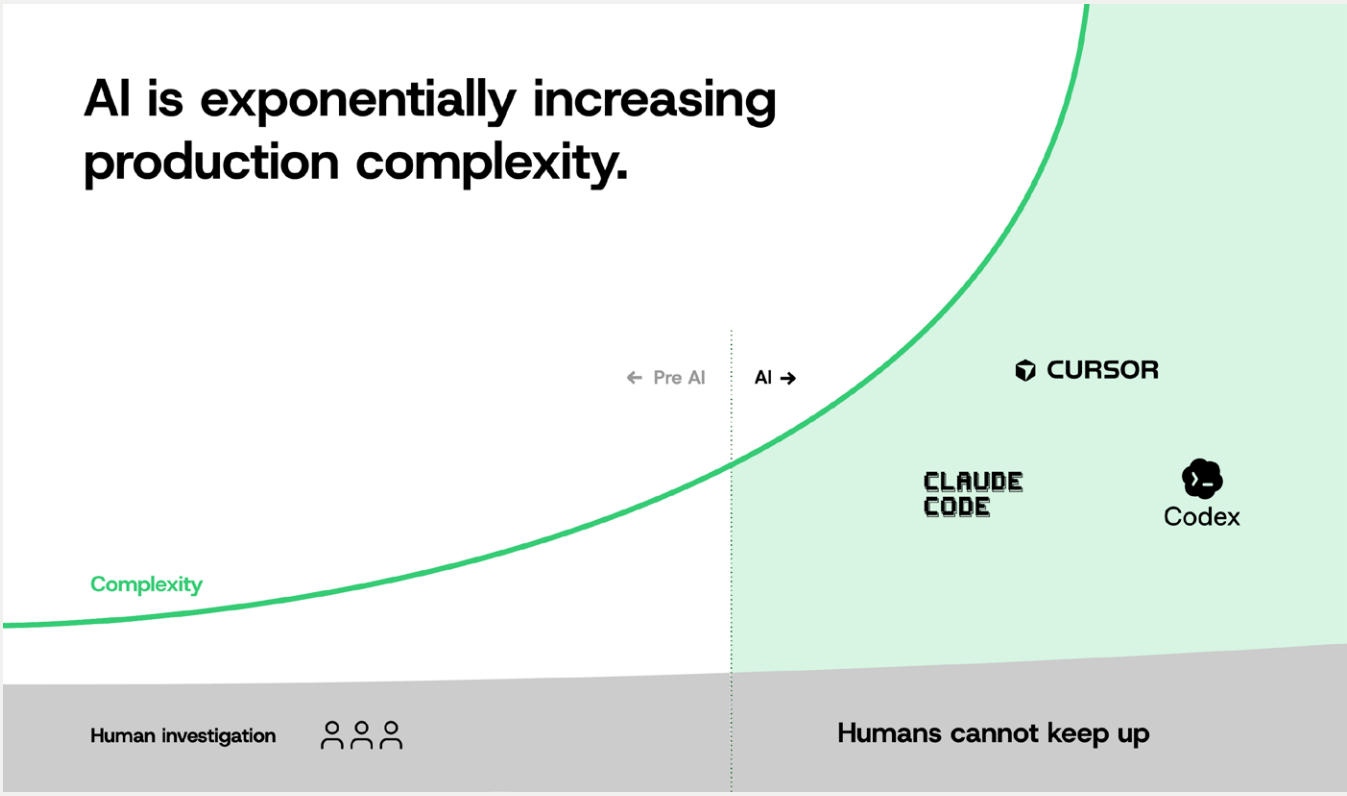


FIG 1.1 Production complexity has outpaced human investigation capacity. AI accelerates this divergence.

Three key forces are driving this, and they compound each other rather:

FORCE	WHAT IT MEANS OPERATIONALLY
Distributed depth	The system spans many services, runtimes, and teams, and a failure in one can surface as a symptom several boundaries away. No responder carries the full picture, and the path from symptom to cause is rarely local.
Change velocity	Code deploys continuously. Last quarter's system map may be directionally correct but operationally inadequate today.
Semantic variability	AI-enabled components can return a 200 status code on time while producing wrong, stale, or unsafe output. Surface health and user-visible correctness decouple. Standard monitoring has no signal for this.

Distributed depth means the system no longer lives in one place, one runtime, or one clearly bounded architecture. Ownership is fragmented across application, platform, infrastructure, data, ML, and vendor teams, and that fragmentation is what makes multi-hop incidents, where root cause sits several dependencies away from where the symptom appears, so expensive. The checkout API is slow, but the actual cause is a queue consumer dropping messages because a schema change deployed two days ago was only partially backward-compatible. The checkout team finds nothing wrong in their service. The queue team sees normal throughput. The data team that owns the schema wasn't paged because their service shows no errors. No single team has the full path in view, so the incident runs until someone does.

This is not a rare edge case. It's a predictable consequence of distributed architecture, and its frequency rises with system complexity. As services and recent changes multiply, plausible explanations multiply faster: a system with twenty interacting services can generate dozens of plausible causal paths for a single symptom. Human teams cannot evaluate all of them in the time available. Responders anchor on the first explanation that fits enough of the evidence, often correctly, sometimes not, always with real cost when they're wrong.

Change velocity means the system is evolving continuously beneath the people trying to operate it. Infrastructure is ephemeral. Every production engineering team has an architecture diagram somewhere, and it likely has a date in the filename. Look closely at the topology it describes and several of the dependencies shown no longer exist in that form; at least two services have been added that aren't shown at all; one of the boxes represents something that was split into three components during a refactor eight months ago. Nobody updated the diagram, not because they were careless, but because the system moved faster than the diagram could follow.

During steady state, this matters less than it should, because engineers who work closely with a service carry an accurate mental model of it, continuously updated by the act of building and operating it. Then an incident happens, and someone who doesn't carry that informal knowledge reaches for the representation. What they find is a map of a city that's been substantially rebuilt since the map was drawn; and worse, they don't know which parts of the map are wrong, so they can't safely discount any of it. That's a harder problem than having no map at all.

Semantic variability is the newest of the three forces, and the one existing tooling is least equipped to handle. It exists because a growing share of production systems no longer execute fixed, deterministic logic. They classify, retrieve, generate, and infer, and those operations don't fail the way a database query or an API call fails. A generative feature can produce a fluent, well-formed, on-time response that is simply wrong. A retrieval step can return context that's stale or subtly irrelevant while completing successfully by every infrastructure measure. There's no exception thrown, no timeout, no non-200 status code, because nothing crashed. The system did exactly what it was asked to do; it just did the "wrong" thing.

Monitoring built for deterministic systems checks whether a request executed; it was never built to check whether what it returned was correct.

The scale of the underlying problem is measurable even before any of these AI-specific failure modes are added on top of it. The average engineer loses seven hours a week to troubleshooting and incident response. And 41% of executives report that downtime is first detected not by their monitoring systems, but by their customers.[1] These are the baseline costs of operating systems whose complexity has already outpaced the workflows designed to manage them—before accounting for what AI adds on top.

1 Splunk & Oxford Economics, The Hidden Costs of Downtime, June 2024.

SECTION III

The Industry's Response: Instrument Everything. More Telemetry, Less Understanding

Faced with rising complexity, the industry did the logical thing: it instrumented everything.

Agents went into every host, every pod, every service. MELT started flowing at a volume measured in petabytes. The 2025 Grafana Observability Survey found that observability now consumes 17% of total compute infrastructure spend on average, with some organizations' monitoring bill outpacing their actual compute spend. For many enterprises, observability has quietly become one of the largest line items in the engineering budget, just behind the infrastructure it's watching.[2]

The pitch behind all of that spending was simple: enough data, and you can see everything. And that part worked. Teams today have more visibility into their production environments than at any point in the industry's history.

However, having all the data does not mean you can explain what happened.

> *"Monitoring and alerting enables a system to tell us when it's broken, or perhaps to tell us what's about to break. Every page response should require intelligence. If a page merely merits a robotic response, it shouldn't be a page."*
>
> GOOGLE SRE BOOK

Let's walk through a typical Sev2. An alert fires at 2am. The on-call engineer opens their laptop to find latency elevated about thirty percent above baseline. They check recent deployments: two went out that afternoon, both looking clean. They scan error rates, slightly elevated but not alarming. A handful of slow requests, traversed through the system, fan out across three downstream services before the trail goes cold. They post in Slack asking the team that owns one of those services whether anything changed, and while waiting for a reply, pivot to logs, filter by request path, and start reading.

Forty minutes in, they've eliminated the most obvious hypotheses and narrowed their focus to a single suspicious dependency. They might be right. They might be wrong. Either way, they've spent forty minutes gathering context that the system technically already had available the entire time. The evidence existed. No one had assembled it.

2 Grafana Labs, Observability Survey 2025, March 2025.

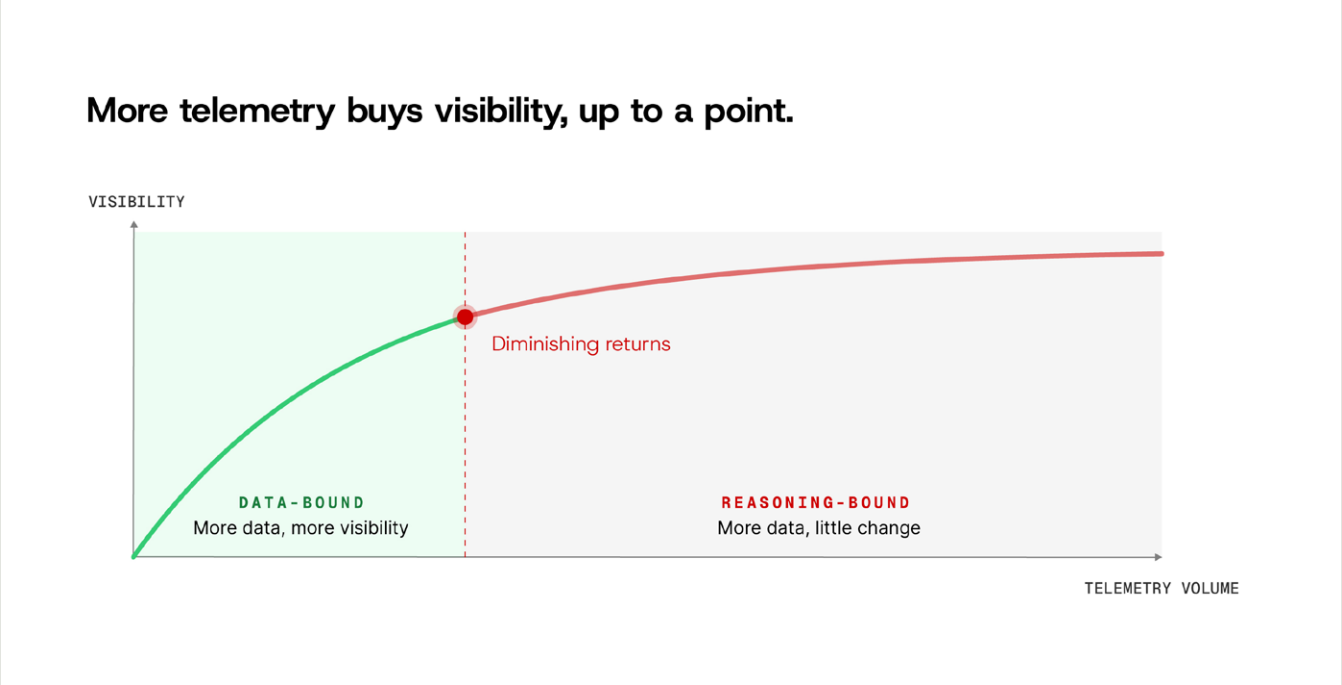


FIG 1.2 More telemetry increases visibility up to a point. Beyond it, the bottleneck is reasoning, not data.

The core issue is that visibility and diagnosis are not the same thing. A dashboard shows that latency is elevated; it does not and cannot explain why. A log line records an event without placing it in context: whether it's causal or coincidental, part of a pattern or a one-off. Converting visibility into explanation still falls entirely to the operator, running the same loop every time: receive an alert, form a hypothesis, check the evidence, rule it out or pursue it, generate the next hypothesis, repeat. An experienced engineer runs this loop competently. But the loop is slow, sequential, and bounded by the engineer's working memory and available attention. The ceiling appears exactly when system complexity exceeds the rate at which the loop can run, which, as Sections II and III described, is happening more often, not less.

The practical implication: investing in more observability addresses visibility, not the reasoning bottleneck behind it. A system with abundant telemetry will not generally recover faster from a complex multi-hop incident just because it gets more telemetry. What it needs is a faster path from evidence to explanation. That is a fundamentally different kind of investment than the one most organizations have been making for the last decade—and it's the investment this booklet is actually about.

SECTION IV

AI Is Accelerating the Curve

> *"We already learned this lesson once in infrastructure: you can automate yourself into a very resilient catastrophe machine. Systems can appear healthy by local metrics while globally becoming incomprehensible. Bug reports can go down while latent risk explodes. Test coverage can rise while semantic understanding falls. Changes happen so fast that nobody notices the underlying architecture decaying."*
>
> MITCHELL HASHIMOTO, FOUNDER OF HASHICORP

AI did not appear in an otherwise stable world and create complexity from nothing. Production systems had already been getting bigger, more layered, and more difficult to reconstruct under pressure for years. What AI changed was the slope of that curve, and it changed it in two directions at once.

The first direction is velocity. AI coding agents have made development teams dramatically faster. A team using AI coding tools may be shipping five times the code volume of two years ago. The inner loop—the local edit-build-test cycle a developer runs before pushing code—has accelerated. That sounds like a win, and in one sense it is. But velocity without a corresponding improvement in the outer loop—test, deploy, maintain—of the software development lifecycle (SDLC)—means code is shipping faster than it can be reliably operated. The outer loop didn't keep pace with the inner loop.

And the cost of the gap between how fast code is being written and how reliably it can be operated is measurable. Lightrun's 2026 State of AI-Powered Engineering report found that 70% of problematic pull requests stem from missing production knowledge: context that exists somewhere in the system but never made it back to the engineer writing the code. That missing context has a direct consequence: faster shipping means more incidents. Global 2000 companies now lose an estimated $600 billion annually to downtime, up 50% from roughly $400 billion just two years earlier.[3]

The second direction is a new class of failure. AI-powered systems don't fail the way a database query or an API call fails. The request returns. The endpoint stays green, and the dashboard looks normal. The result is simply wrong. A retrieval-augmented system answers confidently from information that changed an hour ago. A classifier routes requests into the wrong workflow because the traffic distribution it's seeing has quietly drifted from what it was tuned on. A generative feature produces a response that is fluent, plausible, and incorrect. None of these register as an HTTP error. None spike latency. Every infrastructure signal an SRE has trusted for two decades says the system is fine.

3 Splunk (a Cisco company) & Oxford Economics, The Hidden Costs of Downtime, May 19, 2026.

This is what it means to say AI-powered systems fail semantically before they fail technically: the degradation shows up eventually, but not where anyone's watching. Not in an alert, but in a support ticket, a churn metric, a customer escalation, or a postmortem struggling to explain why a system that was never technically down still caused a real business problem, usually hours or days after the degradation actually began, once enough of those tickets have piled up for someone to notice the pattern. Each of these passes every check a dashboard can feasibly give, which is exactly why they require a structurally different kind of measurement.

That mechanism plays out in a handful of concrete patterns:

FAILURE MODE	DESCRIPTION	WHY IT'S HARD TO DETECT
Model behavior regression	Model update, provider change, temperature or system prompt change alters output quality	Invisible in infrastructure metrics; requires semantic evaluation
Retrieval failure	Relevant context missing, stale, poorly ranked, or swamped by matches that are semantically similar yet not correct	Generation layer works correctly; the context layer is wrong
Prompt/orchestration regression	Small change to instructions, tool order, or agent workflow shifts behavior across edge cases	High-impact, deceptively small; hard to catch without canary evals
Evaluation mismatch	Offline benchmarks show acceptable quality; production behavior drifts	Synthetic evals capture one error class while missing what users actually experience
Policy/safety regression	System remains responsive but becomes less aligned with compliance, brand, or trust requirements	May not appear in any conventional SLI until a human reviews output
Non-deterministic instability	Same inputs produce inconsistent outputs across runs or replicas	Debugging is difficult; reproducibility breaks
Data and context drift	The world the system reasons about changes; retrieval, ranking, or reference knowledge hasn't kept pace	System answers confidently about a state of the world that no longer exists

This table is the concrete answer to a question that's easy to state abstractly and easy to underestimate in practice: AI systems don't just fail more often. They fail differently, in ways that pass every conventional health check on their way to becoming a business problem.

SECTION V

From RED to RADES

Classic SRE workflows were built around failures that manifested as visible degradation in conventional operational metrics: if a service was slow, unavailable, or erroring, it signaled distress in a recognizable way. The failure modes above don't announce themselves that way. Operating AI-powered systems requires a way to detect when observable health and user-visible correctness have separated, which means the metrics themselves have to change, not just the tooling built on top of them.

RED was built for a world that no longer exists. In 2015, Tom Wilkie at Weaveworks introduced RED metrics—Rate, Errors, Duration—as a practical shorthand for monitoring request-driven microservices. It became a standard, cited in the Google SRE canon and taught to a generation of operators. RED isn't wrong. It was designed for a world where a service either returned a correct response or an error. AI systems introduce a third outcome RED has no signal for: a response that's technically successful, arriving on time, with no error code, within latency bounds — but semantically wrong. A system can post perfect RED metrics while failing its users on every single request.

The framework that extends RED for this environment is **RADES:** Rate, Accuracy, Duration, Errors, Saturation.

SIGNAL	WHAT TO MEASURE	QUESTION IT ANSWERS
Rate	Requests per second by type	Is the system receiving expected traffic?
Accuracy	The ceiling on how reliably the system can be diagnosed, set by the quality of its telemetry, documentation, and system knowledge	How observable is the system, and therefore how well can it be diagnosed at all?
Duration	Latency decomposed: time-to-first-token, retrieval, reranker, tool execution, end-to-end generation	Is the system fast enough at each stage, not just overall?
Errors	Hard errors (5xx, timeouts) plus soft errors (fallbacks, safety triggers, empty retrievals, refusals)	Is the system failing visibly, or failing silently?
Saturation	GPU memory, context-window utilization, KV cache pressure, token budget, vector-index query latency	Is the system approaching its AI-specific capacity limits?

Accuracy is the signal RED never had, and it measures something different in kind from the other four. Where Rate, Duration, Errors, and Saturation describe the health of the production system, Accuracy describes how well that system can be diagnosed in the first place — a proxy for how well-instrumented the environment actually is. Errors gain a second tier RED never needed: soft errors like fallback invocations and empty retrievals return a clean 200 and look healthy from the outside, which is exactly the failure mode a system can hide in if nobody's watching for it separately from hard errors.

SECTION VI

Limitation of AIOps: Correlation Is Not Causation

> *"Correlation is not causation: some correlated events share common causes. Even worse, as systems grow in size and complexity and as more metrics are monitored, it's inevitable that there will be events that happen to correlate well with other events, purely by coincidence."*
>
> GOOGLE SRE BOOK

The industry's response to the reasoning bottleneck was a new generation of tooling built to find correlation. AIOps platforms cluster alerts that fire together. Anomaly detection flags metrics deviating from baseline at the same moment. Event-correlation engines group signals by temporal and statistical proximity, collapsing a storm of three hundred alerts into a handful a human can actually read.

That approach has a valid, albeit narrow, use: in a small, tightly-scoped system, where the causal path from symptom to cause was short and the service that alerted was usually the service with the problem, knowing which signals moved together was often sufficient to know where to look. Correlation was a reasonable proxy for causation because the distance between them was small. Outside that band, the picture is less favorable. Even at mid-sized organizations, correlation tooling tends to tell you one thing: surfacing that something is wrong, and roughly where. It's rarely good at telling you why something is broken, and the engineering effort required to stand up and tune a correlation pipeline, instrument the right signals, build the clustering logic, keep it current as the topology changes, is often out of proportion to the diagnostic speed it actually buys back. Organizations frequently end up with a system that's very good at generating a shorter alert list and only marginally better at helping anyone find the root cause of an issue.

As distributed depth increases, root cause increasingly sits several dependencies away from where the symptom appears, and correlation-based tooling has no mechanism for crossing that distance. It can tell you that latency in checkout, error rates in the payment orchestrator, a queue-depth spike, and a deployment four services away all moved within the same ninety seconds. It cannot tell you which of them is the cause, which are intermediate effects, and which are simply coincidental. It surfaces the cluster and leaves the actual reasoning to the human.

The problem is combinatorial, and it gets worse exactly as systems get harder to operate. The Google SRE Book named this two decades ago: as you monitor more metrics, spurious correlation becomes not a risk but a statistical certainty. A multi-hop failure in a system of twenty interacting services can produce dozens of correlated signals for a single symptom, and most of them are coincidental: services that happen to be under load, deployments that happened to land in the same window, metrics that move together because they share an upstream dependency that has nothing

to do with the failure. More telemetry does not thin this thicket. It thickens it. This is why organizations with the most observability data are often no faster at complex incidents: they have more to correlate and no better way to tell signal from coincidence.

It's tempting to assume a sufficiently capable large language model (LLM) dissolves this problem. It does not. A frontier model that has read ten thousand incident postmortems is an extraordinary correlation engine: it can generate fluent, specific, plausible-sounding hypotheses faster than any human. But fluency is not verification. The model is pattern-matching the current symptom against everything that resembles it in its training distribution, which is correlation operating over a much larger corpus, not a different kind of operation. It can tell you what tends to cause symptoms like this one. It cannot tell you what caused *this* one, in *your* environment, right now, because it has no access to the cause-and-effect structure of your specific system, and no way to test its own hypotheses against evidence. And because the output is articulate and confident, a wrong answer is harder to catch than a human's tentative guess would have been.

SECTION VII

What Needs to Be Solved

Twenty years of observability investment significantly closed one problem: the distance between systems and the telemetry describing them. Teams today have more visibility into their production environments than at any point in the industry's history.

The following are the two problems that fundamentally need to be addressed for us to close the outer loop of the new SDLC.

The first is that production knowledge doesn't flow back into development. The signals that would tell an engineer their code is about to introduce a fragile dependency, touch a historically risky service, or repeat a failure pattern from three months ago exist somewhere in the system, but they never make it back to where the decision is being made. With coding agents, the inner loop closed. The outer loop, everything from deployment through production feedback, is not. The same failures get shipped repeatedly, and AI makes it worse: more production knowledge generated, faster, but a smaller fraction of it ever reaches the engineers and tools making the next decision.

The second is that production knowledge can't be assembled into a causal explanation fast enough when something breaks. The telemetry is there—scattered across services, teams, and tools, but present. What's missing is the ability to reason across it causally, under pressure and time constraints. More observability and better correlation don't solve this. They produce more signals to reason across, which makes the problem harder, not easier.

The first requires production knowledge to flow back into development before code ships. The second requires a system that can reason across the environment causally when something breaks. Neither is solved by more telemetry.

Part 2 maps what becomes possible when both problems are addressed: the progression from manual troubleshooting to self-driving production. Part 3 is about what it actually takes to build a system capable of getting there.



PART 02

The Vision for AI-Native Software Reliability

SECTION I

The Self-Driving Maturity Ladder

Part 1 described two problems that compound each other: production knowledge not flowing back into development before code ships, and no system capable of reasoning across the environment causally when something breaks. Closing both follows a progression. Organizations don't get there in one step: they move through levels of capability, each one requiring something the level below it didn't have.

The levels below map that progression. The self-driving car industry offers the clearest parallel; not because the technology is similar, but because the gating condition is the same: in both domains, each level of autonomy depends on having a more complete and causally connected world model of the environment than the level below it. What that world model and surrounding architecture requires in practice is what Part 3 gets into. For now, the pattern holds: without that foundation, every level above basic automation collapses back into rules-based pattern matching. With it, the higher levels become possible.

LEVEL	NAME	WHAT IT MEANS	WHERE MOST ORGS ARE
L0	Manual troubleshooting	War rooms, dashboards, whoever's awake. Investigation is entirely human.	Most organizations
L1	Rules-based automation	Known failures trigger known workflows. Static runbooks. Breaks on novel failure modes.	Common baseline
L2	AI summarization	AI surfaces context and summarizes incidents. Humans still drive all diagnosis.	Current leading edge
L3	Single-domain investigation	Agents investigate within one layer: a codebase, an observability surface. Cross-service root causes are missed.	Emerging
L4	Cross-environment investigation	Agents can causally reason across all of production, over multiple hops of telemetry, code, changes, and dependencies. Humans gate high-risk actions.	Forward edge
L5	Self-driving production	Agents can take action and autonomously fix reliability issues. Further, Agents monitor changes across the entire SDLC, and proactively prevent incidents at the earliest possible time.	Aspirational

Most organizations, in practice, sit at L2 or below: tools that surface context, cluster alerts, and generate a readable incident summary, while a human still does every step of the actual diagnosis. A smaller set of more advanced organizations have pushed into L3, single-domain agentic investigation, where an agent can genuinely investigate within one bounded surface, a codebase, a single observability platform,

but still misses root causes that cross into a different layer or a different team's territory, because nothing causally connects what it knows to what the rest of the environment is doing.

The jump from L3 to L4 is not incremental. Single-domain investigation can work with a partial view because the questions it answers stay inside a small enough boundary, and because that boundary usually maps to a team. One agent, one observability surface, one service owner. Cross-environment investigation cannot work that way: reasoning across many hops of telemetry, code, changes, and dependencies requires full production context, not a slice of it, and a reasoning layer capable of deciding what evidence to pull, which hypotheses to pursue, and when to rule one out. Without both, an agent doesn't investigate more broadly at L4; it just applies single-domain reasoning to a wider surface, and gets things wrong at a larger scale.

The organizational implication is just as significant as the technical one. At L3, the agent respects team boundaries because it can't see past them. At L4, investigation crosses those boundaries by design: which means the escalation model, the ownership model, and the on-call structure all have to evolve alongside the technology. That's a real gap between where most organizations are today and where this ladder is heading next. It's not just a matter of better tooling. It's a different operating model.

The value of this ladder isn't to suggest every organization should reach for L5 immediately. It is to separate capabilities that get blurred together in most conversations about AI in operations. An organization that knows which level it's actually operating at—and what the next one concretely requires—is better positioned than one operating on a vague sense that it needs to "use more AI."

L5, self-driving production, is the north star the rest of this part describes.

SECTION II

What L5 Actually Looks Like

“Self-driving production” is easy to nod along with in the abstract and easy to misjudge in practice. The gap between L4 and L5 is not a matter of degree; it’s a different relationship between the system and the humans operating it.

At L4, the system runs the entire investigation autonomously across the production environment, following a symptom across telemetry, code, changes, and dependencies without a human directing any step of it. A ranked, evidenced hypothesis is ready before the bridge opens. But action—the rollback, the config change, the reroute—still waits on a person to approve it. That’s a notable leap over L3, where investigation stayed bounded to a single domain, and it’s where most organizations serious about this transition should expect to land first.

At L5, the system closes the loop itself, without waiting for a human to be in the room. It operates in two modes simultaneously. On the reactive side, the system senses a deviation through a configured trigger, whether it be an alert or an incident ticket. It then reasons about the cause, takes the remediation, and confirms the fix: the same loop a skilled on-call engineer runs manually, except continuous, and without needing a person awake to trigger it. On the preventative side, it also senses a deviation, but earlier and without waiting for a production trigger. It acts upstream in the development process—at code review or deployment—where it surfaces production context before a regression ships, or continuously against raw production telemetry, where it detects early warning signals before they surface as an incident. Investigation, remediation, and prevention all happen without a human in the loop, for the classes of action a policy has explicitly authorized.

The role of the human at L5 shifts accordingly. It’s not to drive individual incidents, but to set the policy that governs which classes of action the system is trusted to take on its own, and to review what it did afterwards. It’s also the work the system can’t do for itself: setting SLO budgets, defining risk tolerance, and spending the time that used to go to firefighting on longer-horizon resilience engineering: the architectural and process changes that reduce how often incidents occur in the first place.

That’s the qualitative shift. At L4, the system’s ceiling is how fast a human can read and act on what it hands them. At L5, the system’s ceiling is how much autonomy an organization is willing to authorize. Every incident that fits inside that policy remediates without ever generating a page.

SECTION III

Closing the Outer Loop of the AI-Native SDLC

L5's preventative capability—agents monitoring changes across the entire SDLC and intervening before incidents reach production—depends on one specific thing: production knowledge reaching development decisions at the moment they're being made. That's the outer loop. Closing it is what separates a system that responds to failures from one that prevents them before they reach production.

AI coding agents operate in the inner loop, and that loop has accelerated dramatically. What hasn't kept pace is everything past it: CI/CD, deployment, production behavior, and the feedback that should flow back from production into future development decisions. An agent that generates a technically correct function has no way of knowing that the service it's modifying has historically been fragile under exactly the kind of load that function will introduce. Neither the agent nor the engineer writing with it can see production. The knowledge exists somewhere in the system, but it never reaches the people or tools making the decision.

Closing the outer loop of the AI-Native SDLC enables self-driving production.

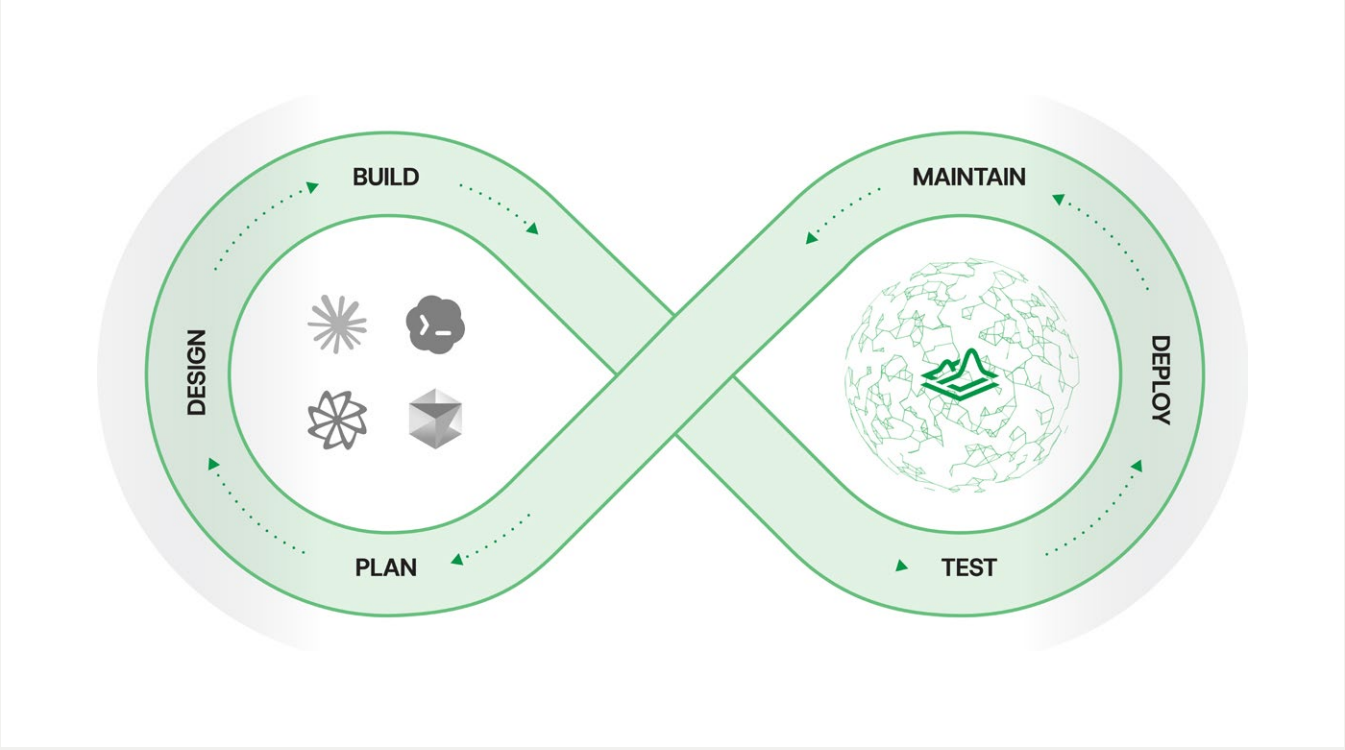
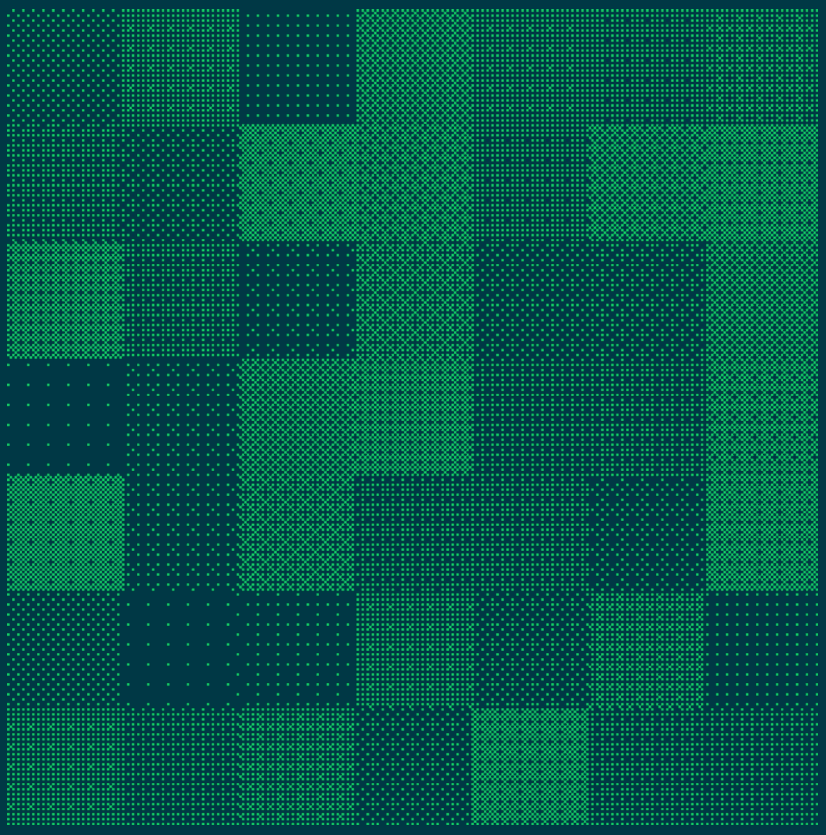


FIG 2.1 Traversal closes the outer loop of the AI-native SDLC, keeping software running in production.

Closing the outer loop means that production knowledge flows continuously back into development: surfacing which services are fragile, which change patterns have historically caused failures, and what the blast radius looks like before a line of code ships.

This is what distinguishes an AI SRE platform from a coding agent. Coding agents close the inner loop.

An AI SRE platform closes the outer one, and in doing so, makes the proactive half of L5, self-driving production, possible. Without production knowledge flowing back into development, the system cannot respond to what production throws at it. With it, the system can prevent failures before they ship. Part 3 of this booklet is about what building that actually requires at a fundamental level.



PART 03

What It Takes to Get There

Part 2 described the destination: a system that can perceive the production environment completely, reason over it causally, and eventually prevent failures before they surface.

This part is about what it actually takes to build that, beginning with the evaluation criteria, then diving into the architecture, because understanding what's required to solve the problem is what makes the architecture legible. It then explains why this is a difficult engineering feat even for the most well-resourced teams. Finally, we walk through one way to build an AI SRE and the opportunities that are unlocked once the architectural foundation is successfully built.

The evaluation criteria and benchmarking results in Sections I and II apply to any team claiming to solve this problem. In Sections III and IV, we give a specific example of the topics covered via Traversal's own implementation, showcasing the full range of possibilities of an AI SRE platform.

SECTION I

How to Evaluate an AI SRE Platform

The right questions to ask about any platform in this category aren't about features. They're about infrastructure, and specifically, whether the platform has built the layers required to go beyond correlation at enterprise scale. A platform that correlates but can't reason causally will fail when root cause sits several hops from the symptom. One that queries on demand will hit rate limits under load. One that requires human-maintained runbooks will degrade the moment those runbooks fall out of date. These aren't edge cases; rather, they're the predictable failure modes of platforms that stopped short of what the problem actually requires.

Those failure modes map directly to five questions. Every meaningful dimension of AI SRE evaluation collapses into two buckets:

Effort and Time to Value: how hard is it to deploy, and how long before it's doing something useful?

Accuracy and Latency: when it does something, is it accurate, and is it fast enough to make a difference?

Five questions, organized across those two buckets, apply to any vendor in this category. A system that fails any one of them is missing something structural.

Effort and Time to Value

- Does it see all your production data agentless and read-only, without gaps or rate limits? API-based, push-and-pull capture is the baseline. A system that relies on querying your observability APIs on demand—mid-incident, under load—will hit the same rate limits those APIs were never designed to absorb at scale, and end up rationing which hypotheses are even worth testing. Push-and-pull capture, agentless and read-only, is the baseline.
- Can it reason at scale without blowing up costs or slowing down? Ask specifically what the vendor does to avoid losing signal at enterprise telemetry volume. Context-stuffing and naive sampling both fail, on economics or on accuracy, the moment production volume is a hundred times what the demo ran on.
- Does it get smarter on its own, or does it require a team of FDEs and markdown files to maintain? If the knowledge layer is a set of human-maintained runbooks someone has to keep current, the organization is now operating two systems in parallel: the platform, and the documentation pipeline feeding it.

Accuracy and Latency

- Can it causally map all of production—millions of entities—in real time? A system with no live world model starts every investigation cold, reconstructing what services exist and what depends on what. That's a slower, more expensive version of what an experienced engineer could already do at an observability console.
- Can it find a root cause far from the initial symptom, in minutes, without being prompted? The incidents that cost the most are never local. Isolating root cause across ten, fifteen, or twenty hops in minutes isn't a feature, it's the core capability this whole category exists to provide.

The five questions above aren't independent: they collectively need to be true and a platform that fails any one of them is missing something that every level above it depends on.

Whether the answer comes from a vendor or an internal platform team, the standard is the same: all five questions answered concretely, with evidence from production deployments at enterprise scale. That's the bar for having built what this problem actually requires.

SECTION II

Why This Is a Hard Problem

> *"We can't trust LLMs for incident response. The problem is its inability to step back and start discerning between causation and correlation. For us humans, it is hard as well."*
>
> ALEX PALCUIE, ANTHROPIC SRE LEAD

A language model alone can't perceive the production environment completely or reason over it causally, and the reason is architectural rather than a matter of scale or better prompting. A frontier model, dropped into an incident cold, has none of what the work actually requires: no persistent memory of the specific production environment, no causal model of how its systems relate, no access to telemetry until something retrieves it, no awareness of what changed recently. These aren't gaps a better model closes, but a fundamental architectural gap.

This is also why the instinct to wire a frontier model, with even a sophisticated agentic harness, directly to your observability stack—the approach most engineering teams consider first—fails under real incident pressure. The model has to solve everything at once, mid-incident: learn your topology live, decide which signals are critical, generate competing hypotheses, retrieve evidence across a dozen systems, and avoid the plausible-but-non-causal explanations that look identical to the real cause from a correlation standpoint. There is no accumulated context from prior investigations or a causal map of how your specific services relate. Every investigation starts from zero.

The result is a system that performs well in demos, where the environment is small, the incident is known, and the answer is implicitly available in the training distribution. However, it degrades badly in production, where the environment is large, the incident is novel, and the evidence is scattered across systems the model has never seen. The gap isn't a prompting problem or a model capability problem. It's a structural problem: the model is being asked to do work that requires infrastructure it doesn't have. Closing that gap requires building the infrastructure, not finding a better model to put on top of the same missing foundation.

The data backs this up more starkly than most vendors in this space would like. **ORCA-BENCH**, published by Traversal's founders and collaborators at Cornell and Columbia, tested frontier agents against realistic, time-delayed, ambiguous incident reports with the full evidence stack an engineer actually reaches for: telemetry, raw signals, and source code together. Run against that full stack, the strongest frontier agents reached only about **28% exact root-cause accuracy** on realistic-input tasks, and under **20%** on the hardest ones, where the report is vague and several causes are active at once. Hallucination rates across the five models tested ranged from roughly **10% to 45%**. Remove source-code access and every metric degrades further.[4]

Every way the benchmark simplifies reality makes it easier than production: a fixed, modest-scale test system, investigated without accumulated context, with no action loop at the end. That means the gap it reports is a floor, not a ceiling. The gap between a model that sounds like an SRE and a system that can be one isn't closed by a better model. It's closed by the infrastructure built around it, which is what the next section describes and what the evaluation questions in Section II are designed to surface.

4 traversal.com/blog/orca-bench-how-ready-are-language-model-agents-for-oncall

SECTION III

Traversal's Approach to the 5 Evaluation Criteria

Fulfilling the evaluation criteria for an AI SRE demands an architecture built specifically for the problem: infrastructure that captures production data continuously without gaps, distills it into a form a reasoning system can actually use at enterprise scale, accumulates operational knowledge across every investigation, maintains a live causal map of the entire environment, and searches that map systematically when something goes wrong.

Each of those is a distinct engineering problem and requires its own layer. What follows is how Traversal solves each of those problems.

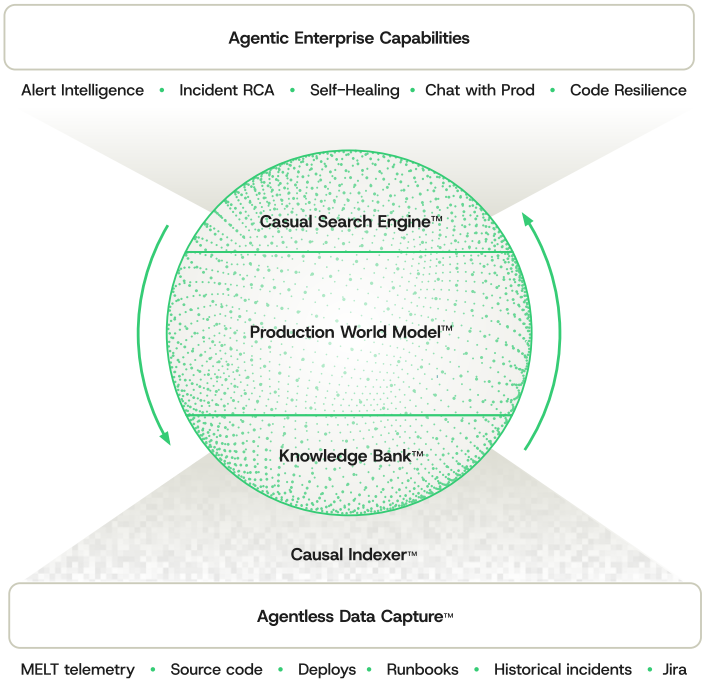


FIG 3.0 The five layers of Traversal's architecture. From the bottom up: Agentless Data Capture™, Causal Indexer™, Knowledge Bank™, Production World Model™, and Causal Search Engine™

Agentless Data Capture™

Does it see all your production data agentless and read-only, without gaps or rate limits?

The first problem is access. A platform that can't see the full production environment can't reason about it, and most tools in this category see less than they claim. Some require new agents on hosts or sidecars in the data plane, which means months of instrumentation work before the platform does anything useful. Others rely entirely on pull: querying observability APIs on demand, mid-incident, and hitting the same rate limits those APIs were never designed to absorb at scale. The result is a platform that rations which hypotheses are even worth testing at exactly the moment volume spikes.

Agentless Data Capture™ solves both problems. It connects over read-only APIs and captures what you already run—your telemetry, your code, your infrastructure, your deploy history—in whatever format each source already speaks, natively, with nothing to map or normalize before it can be used. No sidecars, no agents on hosts, no new infrastructure for a security team to approve. To a security team, Traversal is simply a read-only consumer of telemetry it already trusts, operating inside limits the team already set. That posture also sets the safety boundary for everything built on top of it: whatever reasons over this data can observe production, but it cannot mutate it.

Read-only access alone isn't sufficient. Agentless Data Capture™ also pushes: a continuous, event-driven stream of telemetry arriving as it's generated, long before an incident starts. A deploy to the checkout service lands in the Production World Model™ within seconds of shipping. When an incident fires, the context behind most questions is already there. What's left for live query are the handful of targeted lookups an investigation actually needs, not the thousands it would otherwise take to reconstruct the environment from scratch.



Causal Indexer™

Can it reason at scale without blowing up costs or slowing down?

The second problem is economics. Raw telemetry at enterprise scale—petabytes of MELT data across thousands of services—blows past what any model's context window can hold, and cost scales linearly. Most platforms respond to this with sampling: discard enough data to fit the budget. That's the wrong tradeoff; sampling means discarding evidence before anyone could determine what data should be kept, which directly breaks the causal reasoning requirement. You cannot isolate root cause across ten hops if the data from hop six was discarded due to cost.

The Causal Indexer™ solves this without sampling. It continuously turns raw MELT data into precise causal dependencies—which services depend on which, and how strongly—distilling it down to as much as 1,000x smaller with minimal information loss. A petabyte of raw telemetry becomes a terabyte that still carries every causal relationship between entities. The intelligence goes into what to keep, not what to discard.



Knowledge Bank™

Does it get smarter on its own, or does it require a team of FDEs and markdown files to maintain?

The third problem is operational memory. A platform that investigates every incident as if it's the first one it's ever seen will keep making the same mistakes: failure patterns recur, dependency fragilities are known, and team practices and ownership context exist somewhere, usually in the heads of senior engineers and in documentation that decays the moment it's written. Most platforms require someone to maintain that knowledge manually: runbooks, topology configs, markdown files kept current by a team of forward-deployed engineers. That's a second system to operate alongside the platform itself.

Knowledge Bank™ solves this by building and maintaining that understanding autonomously. From the moment Traversal connects to your systems, it continuously refreshes its model of your services, infrastructure, recurring failure modes, and team practices. Engineers can inspect what Traversal has already understood, add context that only exists in the team's heads, and watch that understanding sharpen as every investigation adds to it. No markdown files required. The platform gets smarter on its own, and the knowledge it accumulates makes every subsequent investigation faster and more accurate



Production World Model™

Can it causally map all of production—millions of entities—in real time?

The fourth problem is perception. A system that starts every investigation cold by querying your observability stack to reconstruct what services exist, what depends on what, and what changed recently, is slower, more expensive, and more error-prone than one that already knows. Root causes that sit several hops from the initial symptom will be missed entirely if the system's view of the environment is incomplete or stale.

Production World Model™ is what the indexed, knowledge-enriched data becomes: a live, AI-readable map of over 100 million entities across your entire production environment—apps, services, and infrastructure—connected across more than 10 billion causal links. This is perception. Think of it the way you'd think about a self-driving car's sensor fusion: it's what lets the system see the whole road, not just the six feet directly in front of it. A self-driving car that can't see the truck in its blind spot will crash. A system that can't see across the full environment will miss root causes that cross the boundary of what it can observe.



Causal Search Engine™

Can it find root cause far from the initial symptom, in minutes, without being prompted?

The fifth problem is reasoning. A platform that can see the full environment but can only correlate what it sees—such as grouping signals by temporal proximity and ranking hypotheses by how often similar patterns appeared in the past—will produce plausible-sounding answers that are completely wrong. The incidents that cost the most are the ones where root cause sits far from the symptom, where the causal chain crosses ownership boundaries, and where the most correlated signal is just a coincidence.

Causal Search Engine™ is the agentic reasoning layer at the top, and the one most visible during an incident. It searches over 1,000 hypotheses in parallel to isolate root cause across ten or more hops and 100 million+ entities, following a failure from the application layer down through services, infrastructure, and networking, eliminating everything inconsistent with the system's actual topology, timing, and behavior. What survives isn't a ranked list of suspects. It's a single, causally consistent diagnosis, with a false positive rate under 1%.

Evaluating a platform means not just examining all five layers, but understanding how each independent layer makes the next one possible.

SECTION IV

What the Architecture Unlocks

Most existing AI benchmarks don't properly reflect production realities: they hand the agent a clean problem, a controlled environment, and a recognizable incident. **ORCA-BENCH,** published by Traversal's founders and collaborators at Cornell and Columbia, was the first attempt to close that gap: testing frontier agents against realistic, time-delayed, ambiguous incident reports with the full evidence stack an engineer actually reaches for. The results—28% accuracy on realistic tasks, hallucination rates up to 45%—showed how far the category had to go. **Traversal-Bench** was designed to go further, testing not just whether models could reason about incidents, but whether a purpose-built platform could outperform the DIY approach under conditions that actually resemble production at large enterprises.

Traversal-Bench covered 25 real production incidents at severity 1 through 3 at a large financial-services enterprise: full customer-facing outages, cross-team war rooms, failures that crossed ten or more dependency hops. Each agent received exactly what a human on-call gets at t=0: the incident timestamp, the original ticket description, and access to the customer's observability systems. Results were judged blind through pairwise comparison: each pair of answers was evaluated by a judge who saw the incident, the raw ground-truth record, and two responses in randomized order, then selected the more accurate root cause. Five votes per pair, majority wins.

The contenders were Traversal against frontier models wired directly to the same observability stack: the floor of the DIY approach any engineering team would consider first. Traversal was judged the better root cause analysis 83% of the time against the strongest direct-access setup, completed investigations in 9.5 minutes versus 19.7 minutes for the next best baseline, and gained 153 Elo over three months of platform upgrades compared to 41 Elo from swapping to a stronger frontier model alone. The gap between a model pointed at your observability APIs and a purpose-built system isn't a model release. On hard production incidents, it's roughly an era of frontier model progress.

Traversal is 2–3x faster than the frontier models. Every minute counts.

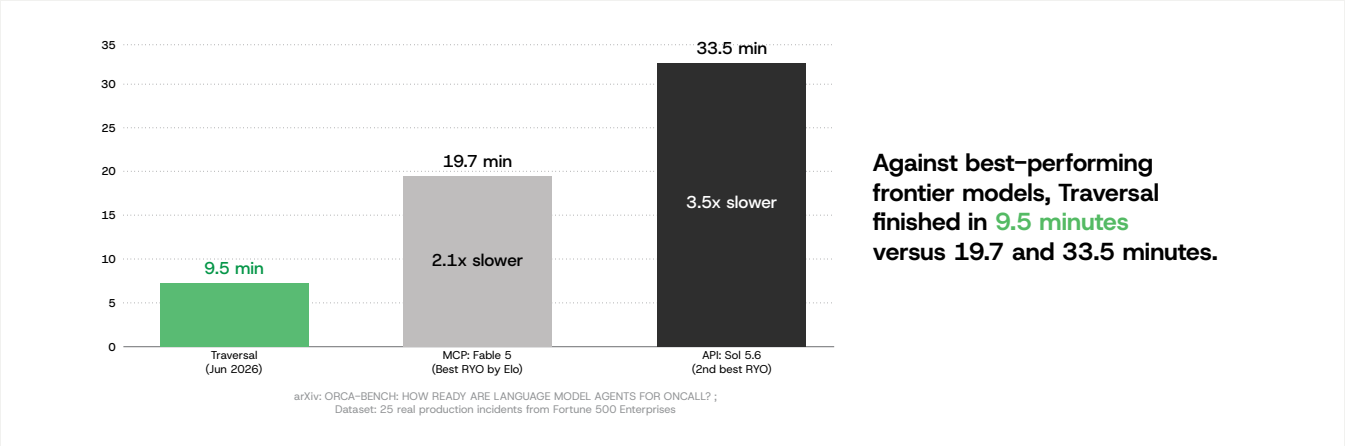


FIG 3.1 Traversal is 2–3x faster than the two highest-scoring direct-access roll-your-own setups.

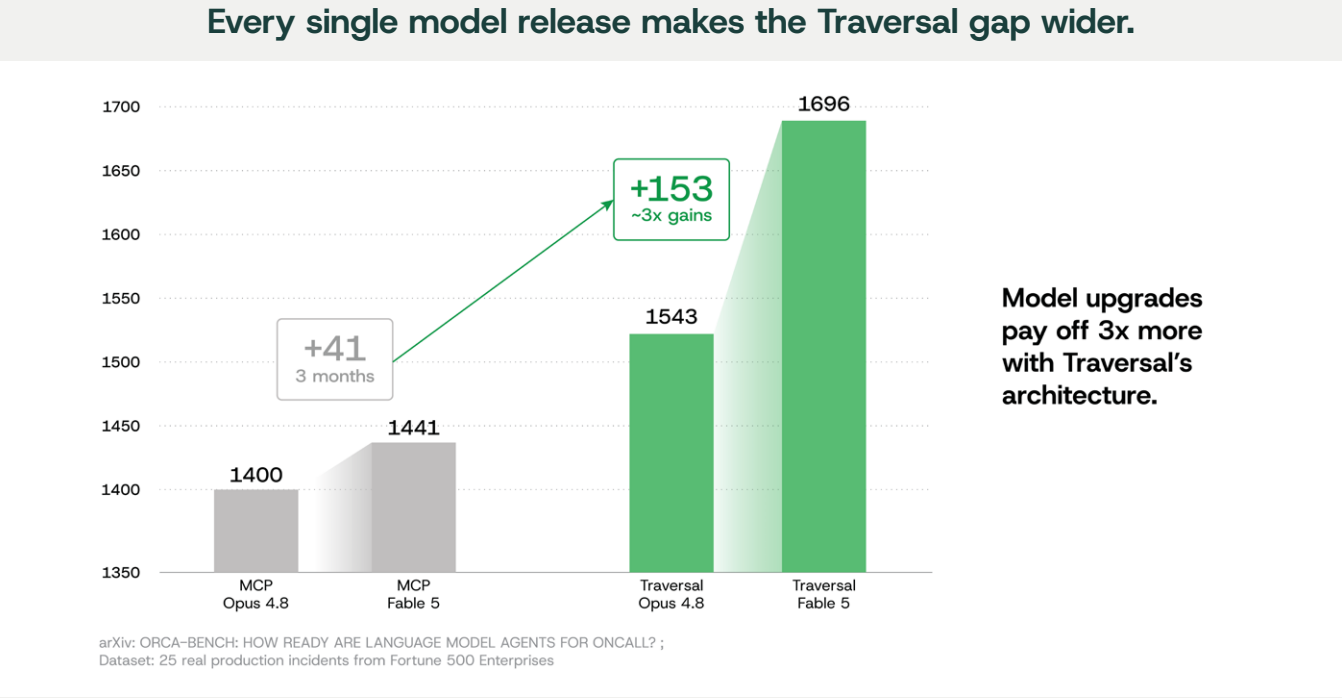


FIG 3.2 Every single model release makes the Traversal gap wider.

Perhaps the most surprising finding from the benchmark is that the performance gap is widening, rather than narrowing, with each new model release. The fundamental reason is that Traversal's Production World Model™ and Causal Search Engine™—the data processing layer and the agentic harness co-designed around it—are purpose-built to translate improvements in model intelligence into better SRE outcomes.

A more capable model given the wrong context will still reach the wrong conclusions. But a more capable model operating within a purpose-built architecture can conduct deeper investigations and produce more accurate answers. As frontier models improve, Traversal is uniquely positioned to compound those gains, advancing even faster on SRE tasks than the underlying models themselves.

SECTION V

The Claude Code Moment for SRE

A recurring pattern in agentic AI is that systems trained to solve the hardest problems in a domain, particularly those with verifiable outcomes, often generalize naturally to a broad range of adjacent, simpler tasks. This is one reason models trained on complex math and science problems develop reasoning capabilities that transfer well beyond those specific domains.

In SRE, the equivalent hard problem with a verifiable answer is incident root cause analysis (RCA). It is arguably the most demanding reasoning task in production engineering, requiring an agent to trace a chain of causality from an initial symptom to its underlying root cause, often across many steps. Doing so requires navigating large volumes of heterogeneous data—including code, logs, metrics, technical documentation, and Slack conversations—each with different structures, levels of quality, and scales. Critically, significant incidents typically produce postmortems that establish a ground truth against which we can evaluate the accuracy of an AI SRE system's conclusions.

The combination of this problem's difficulty and the ability to verify its answers has produced an interesting emergent property of the Traversal platform: a broad range of reliability use cases we never explicitly trained for or designed around have become natural extensions of the core system.

The same causal model that identifies root causes during incidents can reason across the entire SRE lifecycle—from alert triage and enrichment before escalation, to change and release risk assessment before deployment, to failure diagnostics and postmortem generation after resolution, to ongoing capacity and performance analysis. Engineers can use Traversal to map dependencies, assess blast radius, retrieve observability data across disparate systems without knowing where it lives, and understand customer journey health without building custom dashboards.

In effect, by training Traversal to solve the hardest verifiable problem in SRE, we have built a foundation that generalizes to much of the broader reliability engineering function.

Our breakthroughs have led to an explosion of use cases.

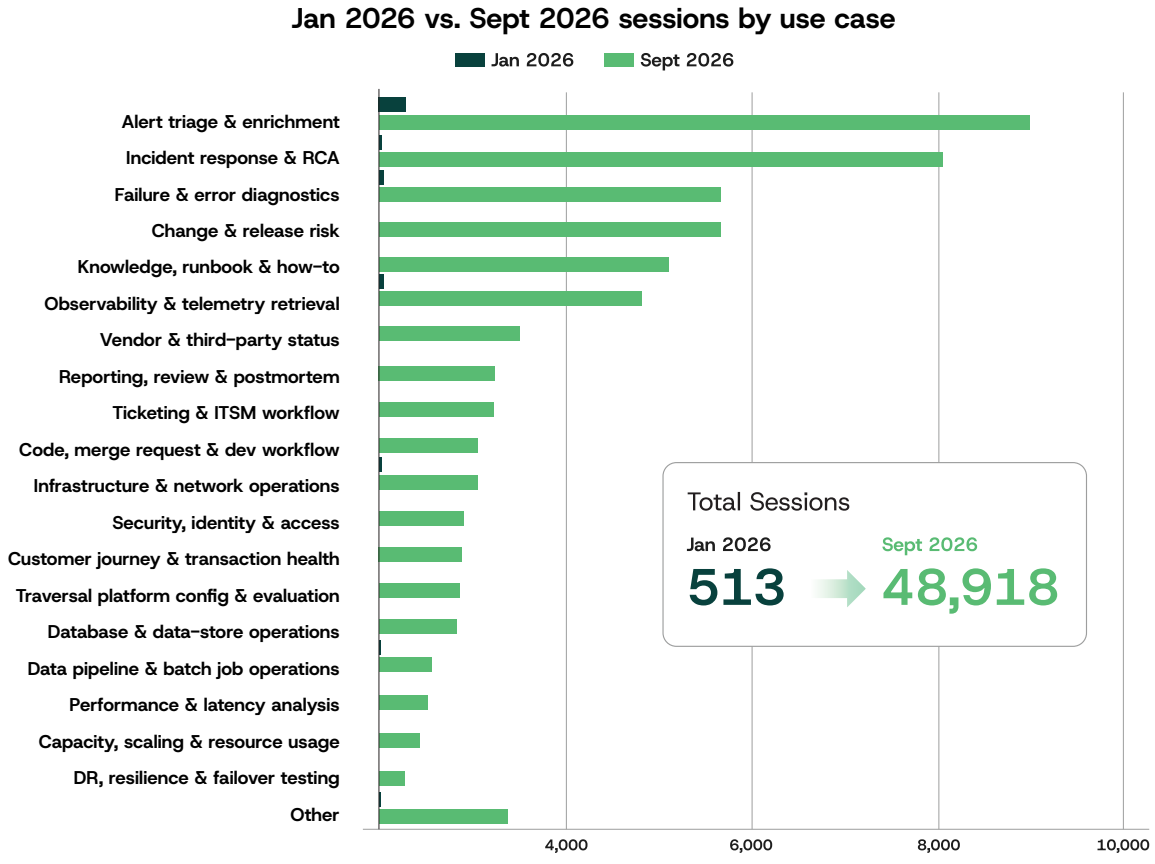


FIG 3.3 Our breakthroughs have led to an explosion of use cases. Sessions by use case, January 2026 vs. September 2026.

One practitioner described this breadth of applicability as the "Claude Code moment for SRE." The comparison captures something fundamental about modern agentic platforms: their value lies not just in the use cases they are explicitly designed to solve, but in the flexibility they give users to discover entirely new ones.

The extent to which a platform enables this kind of bottom-up discovery and adoption should itself be a key measure of its value.

CONCLUSION

Reliability Is Still a Choice

Classic SRE didn't fail. The environment outran it: eroding for years, and now sharply steepened by AI.

For two decades, the challenge was visibility: closing the distance between systems and the telemetry describing them. Organizations closed that gap. A different one opened in its place, the distance between the evidence that already exists and an organization's ability to reason across it fast enough to matter. When an incident spans thousands of services, tens of teams, two clouds, and components that fail semantically without failing technically, no single operator carries the model needed to diagnose it in time.

SRE in the age of AI is the discipline under that pressure. AI SRE is what relieves it, at the scale the problem now demands.

The organizations that navigate this well have already absorbed what follows from that. Incident response at modern scale is an AI problem, not an observability & staffing problem. Operational knowledge has to live in systems that persist, not in whoever happens to be holding it in their head this week. The escalation pyramid has to be rebuilt around authority limits rather than knowledge limits, because the reasoning that used to require a room full of tenure can now arrive with the page. And evaluation has to run continuously, not once before a system ships—including evaluation of the system doing the evaluating, since a platform that reasoned well on day one can still drift on day two hundred.

None of this asks anything new of the discipline itself. Explicit objectives, honest risk accounting, continuous learning from failure: that's the same SRE it's always been. What's different is the environment that discipline now has to operate in, and one genuinely new fact about it: closing the gap that environment opened is, for the first time, actually possible. Not aspirational. Not a roadmap item. Possible now, for an organization willing to deploy the architecture this booklet describes and hold it to the standard this booklet argues for.

Reliability has always been a choice. What's changed is that for the first time, the tools to make the right one actually exist. The gap between production complexity and human reasoning capacity—the one that has been widening for years and that AI has sharply steepened—is now closable. Not through more dashboards, more alerts, or more engineers on rotation. Through a system that perceives the production environment completely, reasons over it causally, and gets better at both with every incident it sees.

The organizations that recognize this early will not just respond to incidents faster. They will prevent failures that would otherwise have shipped, catch degradation before users notice it, and free their engineers to work on the systems that make reliability possible in the first place rather than the incidents that prove it isn't. That is what self-driving production actually delivers, not automation for its own sake, but a fundamentally different relationship between the people responsible for production and the systems they operate.

The standard is clear; the architecture, evaluation criteria, and capability exists, already deployed at some of the world's largest enterprises. What remains is the decision to hold a platform to that standard—and to build toward the level of autonomy the problem now demands.

REFERENCE

Glossary

AI SRE: The platform category: the agentic systems built to do core SRE tasks at scale, including causal investigation, autonomous remediation, and continuous reasoning over large-scale production environments.

AIOps: A generation of operational tooling built to find correlation. AIOps platforms cluster alerts that fire together, flag metrics that deviate from baseline, and group signals by temporal and statistical proximity.

Accuracy: The signal added to RED in the RADES framework. It sets the ceiling on how reliably a system can be diagnosed, based on the quality of its telemetry, documentation, and system knowledge.

Agentless Data Capture™: Traversal's access layer. It connects over read-only APIs to the telemetry, code, and infrastructure a team already runs, and adds a continuous, event-driven stream of data as it is generated.

Causal Indexer™: Traversal's indexing layer. It continuously turns raw MELT data into precise causal dependencies without sampling, so that reasoning at enterprise scale stays fast and affordable without sacrificing accuracy.

Causal Search Engine™: Traversal's agentic reasoning layer, and the one most visible during an incident. It searches over many hypotheses in parallel to find root cause, including causes far from the initial symptom.

Change velocity: One of the three forces compounding production complexity: the system evolves continuously beneath the people trying to operate it, so last quarter's system map is no longer reliable.

Distributed depth: One of the three forces compounding production complexity: the system spans many services, runtimes, and teams, and a failure in one can surface as a symptom several boundaries away.

Inner loop: The part of the software development lifecycle where AI coding agents operate, and which has accelerated dramatically.

Knowledge Bank™: Traversal's operational-memory layer. It builds and maintains an understanding of an environment autonomously, so that each investigation benefits from the ones before it.

L0–L5 maturity ladder: A progression from manual troubleshooting (L0) to self-driving production (L5), modeled on the levels of autonomy used in the self-driving car industry.

LLM: Large language model.

MELT: Metrics, events, logs, and traces: the four types of telemetry that teams instrument across production systems.

ORCA-BENCH: A benchmark published by the Traversal AI Research team, in collaboration with Columbia University and Cornell Tech, that tests frontier agents against realistic, time-delayed, ambiguous incident reports with the full evidence stack an engineer would use.

Outer loop: Everything past the inner loop of the software development lifecycle: CI/CD, deployment, and production operations. Closing it means production knowledge flows continuously back into development.

Production World Model™: Traversal's live, AI-readable map of the entities across an enterprise's production environment, connected causally.

RAG: Retrieval-augmented generation: a pattern in which a system answers a question by retrieving relevant context and passing it to a language model.

RADES: Rate, Accuracy, Duration, Errors, Saturation. The framework that extends RED for AI-powered systems, developed by Traversal.

RCA: Root cause analysis.

RED: Rate, Errors, Duration. Metrics introduced by Tom Wilkie at Weaveworks in 2015 for monitoring request-driven services.

SDLC: Software development lifecycle.

Semantic variability: One of the three forces compounding production complexity: AI-enabled components can return a successful response on time while producing wrong, stale, or unsafe output.

Semantic failure: A failure mode specific to AI-powered systems: output quality degrades while the request still succeeds, so the problem shows up in a support ticket or a churn metric rather than in an alert.

Sev1 / Sev2: Incident severity levels. Lower numbers denote more severe, more widely felt incidents.

SLI: Service level indicator: a measured signal of how a service is performing.

SRE: Site reliability engineering. A discipline that gives organizations a rigorous way to manage the complexity of production systems through explicit objectives, honest risk accounting, and continuous learning from failure.

SRE in the age of AI: The discipline: practicing reliability engineering in an environment AI has fundamentally changed.

Traversal-Bench: A benchmark covering 25 real production incidents at severity 1 through 3 at a large financial-services enterprise, comparing Traversal against frontier models wired directly to the same observability stack.

REFERENCE

Index of Figures

FIG 1.1	Production complexity has outpaced human investigation capacity	8
FIG 1.2	More telemetry increases visibility up to a point	12
FIG 2.1	Traversal closes the outer loop of the AI-native SDLC	23
FIG 3.0	The five layers of Traversal's architecture	29
FIG 3.1	Traversal is 2–3x faster than roll-your-own setups	33
FIG 3.2	Every single model release makes the Traversal gap wider	34
FIG 3.3	An explosion of use cases, January vs. September 2026	36

Traversal

Traversal is the AI site reliability engineer (AI SRE) for the enterprise, helping engineering teams autonomously diagnose, remediate, and prevent production incidents. Its Production World Model™ and Causal Search Engine™ give AI a continuously updated understanding of production systems, their dependencies, and the root causes of incidents.

Traversal is trusted by companies including Nubank, Harvey, PepsiCo, and many more. Founded by causal machine learning researchers from MIT, Berkeley, Cornell, and Columbia, Traversal is backed by Sequoia Capital, Kleiner Perkins, Redpoint Ventures, and Khosla Ventures. Learn more at traversal.com.

