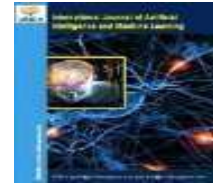




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

Observability and Automated Operational Intelligence in Modern Cloud Data Platforms

Sreekanth Nalla

Independent Researcher, USA. ORCID: 0009-0009-6121-230X

Abstract

Cloud data platforms have grown into distributed systems whose internal state is no longer knowable through conventional monitoring alone. Operators increasingly depend on observability to resolve failures before they compromise data integrity or service availability. This article examines how observability and automated operational intelligence together shape the reliability of modern cloud data platforms. It synthesizes literature on the three observability pillars, metrics, logs, and traces, and their extension through automated anomaly detection and self-healing response, arguing that observability's value depends less on telemetry volume than on the speed and accuracy with which that telemetry converts into corrective action. It also discusses an integrated Detect-to-Correct framework that ties instrumentation and automated response together, which is based on distributed tracing, AIOps, and site reliability engineering (SRE) research. It concludes that organizations benefit most when observability is treated as an operational capability embedded in system design, not a monitoring layer added after deployment.

Keywords: Observability; Cloud Data Platforms; Automated Operational Intelligence; Distributed Tracing; Anomaly Detection; Site Reliability Engineering

1. Introduction

A cloud data platform rarely fails all at once. More often it degrades in pieces, a queue backs up quietly, a downstream consumer starts retrying against a service that has slowed but not crashed, and a partition in a distributed log falls behind its replicas without tripping any single alert threshold. By the time a human operator notices, the platform has usually been unwell for some time. This is the operational reality that has pushed observability from a peripheral concern to a central design requirement in cloud-native data engineering.

The term "observability" originated in control theory and refers to the degree to which the internal state of a system can be determined by its external outputs (Majors et al., 2022). Applied to software systems, observability asks a narrower but still demanding question: when something goes wrong in a distributed platform, can an operator reconstruct what happened, where, and why, using only the signals the system already emits? This is distinct from traditional monitoring, which checks predefined conditions against predefined thresholds and alerts when a threshold is crossed. Monitoring answers questions anticipated in advance. Observability is meant to answer questions nobody thought to ask before the incident occurred (Majors et al., 2022).

Modern cloud data platforms make this distinction consequential. A platform built from managed streaming services, serverless compute, container orchestration, and multiple data stores does not fail in the ways a monolithic application fails. Failures propagate across service boundaries, retry logic absorbs and re-emits them, and they often surface far from their root cause. Majors et al. (2022) describe this shift as the reason observability practice had to mature beyond dashboards built for infrastructure metrics toward instrumentation built for understanding arbitrarily complex, previously unseen system states. Their argument, grounded in production experience at scale, is that the defining feature of a modern distributed system is that most of its failure modes are ones nobody has seen before, which makes pre-built dashboards insufficient by construction.

This article addresses two questions that the literature treats separately more often than it should. First, what does effective observability actually require in a cloud data platform context specifically, as distinct from general distributed systems observability? Second, how does the growing body of automated operational

intelligence, anomaly detection systems, AIOps platforms, and self-healing response mechanisms change what observability is for? The working argument developed here is that these two questions are connected by a single throughline: the value of an observability practice is not measured by how much telemetry it collects but by how quickly and accurately that telemetry can be converted into a correct operational response. A platform that collects everything and helps no one is not observable in any meaningful sense. This argument is developed through a Detect-to-Correct framework that treats detection and correction as a single operational loop rather than as sequential, loosely coupled activities.

The remainder of the article proceeds as follows. Section 2 reviews the three observability pillars and their specific application to data platform components. Section 3 examines automated anomaly detection and the AIOps literature. Section 4 develops the Detect-to-Correct framework and connects it to automated and human-in-the-loop response practice, including the role of chaos engineering in validating that response mechanisms function under real failure conditions. Section 5 addresses capacity and resource observability as a distinct but related concern. Section 6 discusses organizational and cultural preconditions for effective observability adoption. Section 7 concludes with implications for practice.

2. Materials and Methods

This article proceeds through structured literature synthesis rather than primary empirical study. No original benchmark data, experiments, or proprietary datasets were collected; all findings, figures, and comparative results are drawn from and attributed to published sources. The reference set was assembled to cover four categories: foundational and seminal works establishing the core vocabulary and technical approach for distributed systems observability and tracing; theoretical models and standards, including service level objective frameworks and open instrumentation standards; highly cited or authoritative industry reports documenting adoption patterns and operational outcomes across surveyed organizations; and recent empirical studies, concentrated in 2020 through the present, evaluating specific detection, diagnosis, and response mechanisms. Each candidate reference and each quantitative figure drawn from it was independently verified against the original source before inclusion; where a figure could not be traced to a directly quotable, attributable finding, the corresponding claim was either discarded or restated qualitatively rather than approximated. The synthesis method proceeds in three stages: cataloging observability practices and mechanisms against the categories established in the sections that follow, cross-referencing each cataloged practice against the empirical and industry literature to determine whether published findings corroborate or complicate its assumed effectiveness, and using points of convergence and divergence to construct the integrated Detect-to-Correct framework presented in Section 4.

3. The Three Observability Pillars in Data Platform Context

3.1 Metrics, Logs, and Traces

The core elements of observability are metrics, logs, and traces, which remain the most common framework in both industry and academic research (Majors et al., 2022; Sigelman et al., 2010). Metrics are numerical measurements aggregated over a time range. Metrics can indicate that something has changed, but are often not very helpful in determining the cause. Logs are discrete, timestamped records of events, valuable for reconstructing a sequence of what happened but expensive to query at scale and easy to drown in volume. Traces follow a single request or unit of work as it moves across service boundaries, which makes them the pillar best suited to diagnosing where in a distributed call path a failure or slowdown originated (Sigelman et al., 2010).

In a cloud data platform, the three pillars map to different architectural layers with different observability properties. In the ingestion layer (the streaming or batch interface through which data enters the system), metrics like throughput and latency are often sufficient until the steady state, at which point logs may be necessary to diagnose processing failures from malformed keys or unexpected values. At the processing layer, where transformations and enrichment occur, traces are disproportionately valuable because a single record's path through multiple processing stages is exactly the kind of cross-boundary journey traces are designed to capture. At the storage layer, metrics dominate again, since the primary observability questions there concern capacity, replication lag, and query performance rather than individual record journeys.

Sigelman et al.'s (2010) foundational work on Dapper, Google's production distributed tracing system, established much of the vocabulary and technical approach still used in trace-based observability today, including the concept of a trace as a tree of nested spans representing the causal and temporal relationships between operations in a distributed request. What is less often discussed is how directly Dapper's original design constraints, low overhead, application-transparent instrumentation, and sampling to control data volume continue to shape the tradeoffs observability engineers face in cloud data platforms specifically.

Sigelman et al. (2010) report that the trace collection daemon consumes less than 0.3% of one CPU core and that trace data represents less than 0.01% of total network traffic in Google's production environment, overhead low enough that tracing could remain continuously enabled rather than switched on only after an incident begins. A data pipeline that processes millions of events per second cannot trace every single event without the tracing overhead itself becoming a performance and cost concern, which is why sampling strategy is not an implementation detail but a design decision with direct consequences for what an operator can and cannot see during an incident.

2.2 The Limits of the Three-Pillar Model

Recent work has begun to question whether metrics, logs, and traces, treated as three independent signal types, are the right mental model at all. Beyer et al. (2016), writing from Google's site reliability engineering practice, argue that the more useful unit of analysis is not the signal type but the service level indicator, a carefully chosen measurement that reflects what users actually experience, from which service level objectives can be derived and error budgets calculated. Under this framing, metrics, logs, and traces are simply different ways of populating the same underlying question: is the system meeting the reliability commitment it has made to its users, and if not, why not?

This reframing matters for cloud data platforms because data pipelines often lack an obvious end user in the way a web application has one. A data platform's users are often other systems, other pipelines, downstream analytics consumers, and machine learning training jobs that depend on data arriving complete, correctly transformed, and on time. Defining a service level indicator for a data pipeline therefore requires translating pipeline health into terms that mirror user-facing reliability even when no human user directly observes the pipeline's output in real time. Freshness, completeness, and correctness function as the data platform analogue to the latency and availability indicators more familiar from request-response systems, and instrumentation strategy should be built around measuring these specifically rather than defaulting to generic infrastructure metrics because they are the easiest to collect.

The Google SRE team's (2018) implementation guidance for service level objectives is direct on this point: an SLO is only as trustworthy as the measurement pipeline behind it, and teams frequently overestimate how faithfully a chosen service level indicator reflects actual user or downstream-system experience until an incident exposes the gap. The companion Site Reliability Workbook (Beyer et al., 2018) extends this principle into implementation practice specifically, noting that error budgets function only when paired with monitoring capable of measuring the relevant indicator accurately and continuously; a service level objective defined against a metric the organization cannot reliably measure is not a functioning error budget; it is an aspiration. This is where the observability discussion above and the SLO framework converge directly: the telemetry infrastructure that makes distributed tracing and correlated logging possible is the same infrastructure that makes accurate, continuous service level indicator measurement possible, and an organization that under-invests in one under-invests in the other as a consequence, whether or not that consequence is intended.

2.3 Instrumentation Overhead and Sampling Strategy

A practical constraint runs through nearly all of the empirical observability literature: instrumentation is not free. Every span recorded, every log line written, and every metric emitted consumes compute, storage, and network resources, and at sufficient scale, these costs become material. Sigelman et al.'s (2010) low reported Dapper overhead reflects a broader principle worth stating explicitly: observability that must be manually enabled after a failure has already begun is observability that arrives too late to help with that failure. The AWS Well-Architected Framework's operational excellence guidance (Amazon Web Services, 2024) treats this design tension directly, recommending that telemetry collection be scoped deliberately against cost and signal value rather than maximized by default, since indiscriminate collection produces storage and processing costs that scale with data volume regardless of whether the additional telemetry improves an operator's ability to detect or diagnose a fault. Microsoft's parallel guidance for Azure workloads (Microsoft Ignite, 2024) reaches a similar conclusion from the operational excellence pillar specifically, framing observability tooling selection as an ongoing cost-value tradeoff rather than a one-time architectural decision, which reinforces that sampling and retention policy deserve the same periodic review as any other operational cost center.

This guidance has direct implications for how cloud data platform operators should think about sampling. A common but flawed approach samples uniformly, capturing a fixed percentage of all traces regardless of whether they represent normal or anomalous behavior. As Aktas et al. show (2025), traces and logs produced under actual or simulated failure are much more valuable for validating the system's resilience (via deliberate failure injection) than those produced under controlled constant load, as the failure-adjacent traces will be the foundation of diagnostic reconstruction after an incident. A sampling strategy that treats all traffic as equally worth observing wastes storage on redundant steady-state data while potentially under-sampling the rare failure conditions that operators most need visibility into. Adaptive or tail-based sampling, which retains traces

disproportionately when they exhibit elevated latency or error signals, addresses this asymmetry directly, though it requires additional infrastructure to buffer and evaluate traces before the sampling decision is finalized.

3. Automated Anomaly Detection and the AIOps Literature

3.1 From Threshold Alerting to Learned Baselines

Traditional alerting depends on static thresholds set by an operator who has to guess, in advance, what value of a metric constitutes a problem. This approach fails in two directions simultaneously. Thresholds set too conservatively generates alert fatigue, the well-documented phenomenon in which operators become desensitized to frequent low-value alerts and begin ignoring or delaying response to all alerts, including genuine incidents. Thresholds set too loosely miss real problems until they have already caused user-visible impact. Cloud data platforms make static thresholding particularly brittle because pipeline load is often naturally variable, a nightly batch job, a weekly aggregation, or a seasonal spike in event volume, such that a threshold appropriate for Tuesday afternoon traffic may be wildly wrong for Sunday's overnight batch window. The AIOps literature, artificial intelligence for IT operations, has developed largely in response to this specific problem. Rather than a fixed threshold, a learned baseline model characterizes what normal behavior looks like for a given metric, log pattern, or trace shape, accounting for daily and weekly seasonality, and flags deviations from that learned baseline rather than deviations from a static number. Implementations of adaptive alerting at Uber, for example, found that moving from static and hard tuning for multiple years to learned-baseline anomaly detection, only 11 alarms needed human investigation out of 232 that may have been investigated previously for only one production system (Uber Engineering, 2023). Although this shows in at least one case the magnitude of the noise reduction that adaptive, seasonality-aware alerting can achieve relative to static thresholding, it is only a single internal case study rather than a universal benchmark.

Voutsas et al.'s (2024) framing of this appears to fit within a machine learning framing of the challenge of alert fatigue specifically. They argue that the problem is not static threshold accuracy per se, but rather that static thresholds treat all deviations the same despite differing operational consequences. Their proposed mitigation approach models alert relevance jointly with anomaly likelihood so that a statistically genuine deviation with negligible operational impact is suppressed or deprioritized rather than surfaced with the same urgency as a deviation the model estimates carries meaningful consequence. This reframing matters for cloud data platforms because not every anomalous metric reading indicates a problem an operator should act on immediately, a brief spike in queue depth during a known batch window is statistically anomalous relative to steady-state traffic but operationally unremarkable, and an alerting system that cannot distinguish the two will continue generating fatigue-inducing noise even after it has adopted learned baselines in place of static thresholds.

3.2 Log Anomaly Detection

Logs present a distinct anomaly detection problem from metrics because they are unstructured or semi-structured text rather than numerical time series. Jernberg et al.'s (2020) systematic review of software fault and failure taxonomies makes clear that faults surface through highly heterogeneous log signatures across systems, which is a central obstacle to building generalizable anomaly detection over log data. A log anomaly detector trained on one system's log format and vocabulary frequently transfers poorly to another system, even within the same organization, because logging conventions vary by team, by language runtime, and by the age of the codebase being instrumented.

In the context of log anomaly detection, in their 2025 systematic literature review of AIOps approaches to this task, De la Cruz Cabello et al. show how the transfer problem of log parsing and anomaly detection to LLMs (large language models) has started to be addressed through LLM-based approaches framing it as a natural language understanding task versus treating it as a static pattern matching problem. Similarly, Lim et al. (2025) compare a fine-tuned LLM approach against a template-based baseline on standard log anomaly detection datasets. On the HDFS dataset, a parameter-efficient fine-tuned Llama3 model achieved an F1 score of 0.997, compared to 0.774 for LogBERT and 0.577 for DeepLog. On the more heterogeneous BGL dataset, the gap widened further, an F1 of 0.969 for the fine-tuned Llama3 approach against 0.250 for LogBERT and 0.285 for DeepLog. This widening gap on BGL specifically, where log formats are less uniform, illustrates the transfer problem Jernberg et al. (2020) describe: template-based methods degrade sharply when log vocabulary diverges from their training distribution, while the language model-based approach retained substantially higher accuracy.

This performance advantage does not arrive without cost. De la Cruz Cabello et al. (2025) note that large language model-based log anomaly detectors show improved robustness to previously unseen log formats compared to earlier template-mining approaches, but these gains come with increased computational cost and,

in several of the studies reviewed, reduced interpretability, since a language model's anomaly judgment is harder for a human operator to audit than a rule derived from an explicit template match. For a cloud data platform operator, this tradeoff is not abstract. A pipeline processing regulated data may need anomaly detection decisions that can be explained during an audit, which weighs against adopting the least interpretable but most accurate available technique without qualification.

Table 1. Comparison of Alerting and Anomaly Detection Mechanisms by Signal Type

Mechanism	Signal Type	Strength	Limitation
Static threshold alerting	Metrics	Simple to implement and reason about	Brittle under variable/seasonal load; drives alert fatigue
Learned-baseline anomaly detection	Metrics	Adapts to seasonality; reduces false positives	Requires sufficient historical data; less interpretable
Template-based log parsing (e.g., DeepLog, LogBERT)	Logs	Computationally efficient; interpretable	Poor transfer across log formats and vocabularies
LLM-based log anomaly detection	Logs	Strong cross-format generalization; higher F1 in benchmark comparisons	Higher computational cost; reduced interpretability
Distributed trace anomaly detection	Traces	Localizes anomaly to specific span/service boundary	Requires consistent instrumentation across service boundaries

3.3 Distributed Tracing as an Anomaly Detection Substrate

Beyond individual metrics or log lines, trace data offers a structural view of system behavior that can itself be mined for anomalies. A trace anomaly is not necessarily a single slow span, it can be an unusual pattern in how spans relate to one another, a service that is normally called once per request suddenly being called five times, or a call path that skips a service it normally passes through. Sigelman et al.'s (2010) original Dapper work anticipated this use case, noting that aggregate trace analysis across many requests reveals systemic patterns invisible in any single trace, and subsequent AIOps literature has extended this concept into automated structural anomaly detection over trace graphs specifically, rather than only over the latency or error-rate metrics traces happen to produce as a byproduct.

Zhang et al. (2024) develop this line of work directly, proposing a trace-based, multi-dimensional method for locating root causes of performance issues in microservice systems that treats the trace graph itself, not just individual span latencies, as the object of anomaly analysis. Their approach cross-references structural deviations in call topology against latency deviations at the span level, on the reasoning that a service degradation frequently changes both how a request is routed and how long each step takes, and that examining these two dimensions jointly localizes root causes more precisely than examining either alone. This is a direct empirical instance of the diagnostic-speed argument developed above: a localization method that narrows a fault to a specific service and specific structural deviation, rather than surfacing only an aggregate latency alert shortens the diagnostic path an operator or automated system must traverse before corrective action can begin.

The practical payoff of this approach for cloud data platforms is diagnostic speed. When metrics-based alerting fires, it typically identifies that a problem exists in some service or component but not necessarily where in a multi-stage pipeline the problem originated. Trace-based anomaly detection, by contrast, can localize the anomaly to a specific span or service boundary directly, which shortens the interval between detection and the point where a human operator, or an automated system, can begin corrective action. This connects directly to the argument developed in Section 4: detection that cannot be quickly translated into localized, actionable information has limited operational value regardless of how statistically sound the underlying anomaly detection model is.

4. The Detect-to-Correct Framework

4.1 Detection Without Correction Is Incomplete

A recurring pattern in both the practitioner and academic observability literature treats detection as the terminal step, as though identifying an anomaly were itself the goal. This framing undersells what observability is actually for in an operational context. An anomaly detected but not acted upon has produced information without producing reliability. The Detect-to-Correct framework developed here treats detection and correction

as two halves of a single operational loop, borrowing structurally from the classical control theory sense-decide-act cycle from which the term "observability" itself derives (Majors et al., 2022), and argues that observability practice in cloud data platforms should be evaluated against how tightly that loop closes, not against how sophisticated the detection stage alone is.

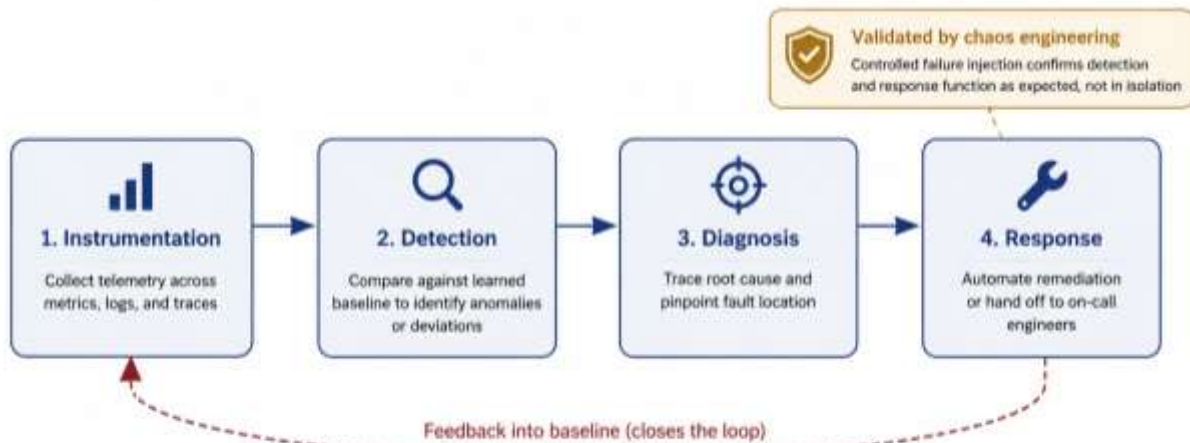


Figure 1. The Detect-to-Correct observability loop shows instrumentation, detection, diagnosis, and response as a closed cycle validated by chaos engineering.

The framework's four stages map onto the discussion in Sections 2 and 3. Instrumentation corresponds to the three observability pillars discussed in Section 2, the raw signal collection layer. Detection corresponds to the anomaly identification methods discussed in Section 3, whether threshold-based, learned-baseline, or trace-structural. Diagnosis is the step most often underexamined in the literature, the process of localizing a detected anomaly to a specific root cause, which trace-based analysis and log correlation both contribute to but which frequently still requires human judgment to complete. A response is the corrective action itself, which may be automated (for example, a self-healing mechanism that restarts a failed component or reroutes traffic) or may require a human operator; the choice between these two options is a design decision that has reliability consequences.

4.2 Automated Response and Its Limits

Self-healing and automated remediation systems attempt to close the Detect-to-Correct loop without human involvement for a defined class of known failure patterns, such as restarting a crashed process, scaling out a resource-constrained service, or failing over to a healthy replica. Automated remediation is one of the more mature applications within the broader AIOps toolkit specifically because the corrective actions available are typically well-bounded and reversible, such as restart, scale, and failover, rather than open-ended. This boundedness matters. An automated system empowered only to take reversible, well-understood corrective actions carries much lower risk than one empowered to take arbitrary corrective action based on a learned anomaly signal, because a false-positive anomaly detection paired with an irreversible automated response can itself become an incident.

For securing cloud workloads, when incident response is automated, Chagovec et al. (2026) argue that the speed of remediation must be balanced against the chance of causing disruption to a legitimate workload with an incorrect automated response. For this reason, the authors propose to restrict the scope of an automated response to the set of pre-authorized actions with a deliberately limited scope, which is continuously reviewed. The reasons for this are also similar: the value of automation in the Detect-to-Correct loop comes from automation being trusted to act in a bounded action space. Following this logic towards failure prediction, Meng et al. (2026) propose a cloud disaster recovery scheme leveraging failure prediction, since proactive recovery based on predictive signals is superior to static recovery procedures with fixed failure rates as failure patterns a priori unknown. Rather than defining a single recovery procedure applied uniformly regardless of failure cause, a predictive model can differentiate between failure modes, distinguishing a gradual resource exhaustion pattern that permits proactive migration from a sudden infrastructure failure that requires immediate failover, and route each toward the recovery procedure best suited to it. This predictive extension

does not replace the validated, bounded automated response discussed above; it operates upstream of it, shifting some fraction of detection earlier in the loop so that response can begin before a failure fully manifests. This is precisely where chaos engineering enters the observability discussion, not as a separate discipline but as the validation mechanism for automated response specifically. An automated remediation system that has never been exercised against a real or realistically simulated failure is a system whose correctness is theoretical. Jernberg et al.'s (2020) taxonomy of software faults and failures provides the vocabulary for reasoning systematically about what failure modes an automated response system should be validated against, rather than testing only against the failure modes engineers happen to imagine. One example is the controlled failure injection studied by Aktas et al. (2025). This directly validates whether the detect-respond stages of an operational loop function in the expected circumstances of production failures rather than the idealized ones experienced in non-production environments during unit tests or failure simulations. Their study found that automated responses validated only in these circumstances behave in unexpected ways in a production service under the resource contention, network variability, and partial failure conditions of production incidents. This is the main reason to treat chaos engineering as a required crop for automated response design rather than an optional maturity practice for the most advanced organizations. Surveys from the industry have shown a broad adoption of chaos engineering in practice- Gremlin's (2021) State of Chaos Engineering survey concluded that 60% of respondents had run at least one chaos engineering experiment, and that the practice of controlled failure injection has moved from a niche practice toward a mainstream validation discipline among organizations operating distributed systems at scale.

In practice, this means an automated response system should be periodically and deliberately exercised through controlled fault injection, such as terminating instances, introducing artificial latency, or simulating dependency failures, in a manner that allows both the detection stage and the response stage of the Detect-to-Correct loop to be evaluated together, rather than validating detection and response as independent components. A detector that correctly identifies a failure but triggers a response mechanism that does not actually resolve the underlying condition has not closed the loop; it has merely relocated the failure, and this kind of gap is precisely what chaos engineering practice is positioned to surface before it is discovered during an actual incident.

4.3 Human-in-the-Loop Response and the Diagnosis Gap

Not every failure pattern is well-bounded enough for automated remediation, and this gap is where the diagnosis stage becomes operationally decisive. When an anomaly does not match a known, pre-authorized response pattern, the loop must route to a human operator, and the quality of that handoff depends entirely on how much diagnostic work the observability platform has already done before the operator becomes involved. An operator handed a raw anomaly alert with no localization information must reconstruct the diagnostic path from scratch, which is slow and error-prone under incident pressure. An operator handed a trace-localized, log-correlated anomaly with a plausible root-cause candidate already surfaced can begin corrective action immediately.

This gap is measurable at an industry level. New Relic's 2023 Observability Forecast found that approximately two-thirds of surveyed organizations (65%) reported that their mean time to resolution had improved to some extent following observability investment, including 31% reporting an improvement of 25% or more, and that resolution time for high-business-impact outages improved by 26% year over year among adopters. This is the practical argument for why Sections 2 and 3's technical discussion, sampling strategy, service level indicators, and trace-based anomaly localization matter beyond their own domain. Every improvement in detection precision and diagnostic automation shortens the human portion of the Detect-to-Correct loop, which is consistently the slowest portion of the loop in organizations that have not invested deliberately in closing this specific gap (Gaggar, 2024).

5. Capacity and Resource Observability

Observability's diagnostic and anomaly-detection role, the focus of Sections 2 through 4, is only part of its operational function in cloud data platforms. A closely related but distinct concern is capacity observability, understanding whether the platform's provisioned resources, compute, memory, storage, and network throughput are sufficient for current and projected load and whether they are being used efficiently. Underprovisioning risks the same reliability failures discussed throughout this article, while overprovisioning carries a direct and continuous cost penalty that compounds at cloud scale.

Ouyang et al. (2022) show that observability-informed container deployment on internet-of-things clouds leads to more efficient resource provisioning than static provisioning based on worst-case resource demands. Their accelerated particle swarm optimization approach to container placement uses observed resource utilization patterns as direct input to placement decisions, reporting that the proposed strategy significantly

improved performance parameters including service transmission overhead, container aggregation, and resource utilization relative to existing deployment strategies. This is structurally the same principle discussed throughout this article applied to a different operational question, that the value of an observability signal lies in how directly it can inform a corrective or optimizing action, not merely in its collection. A capacity metric that is collected but never fed into a placement, scaling, or provisioning decision has the same limited operational value as an anomaly that is detected but never corrected.

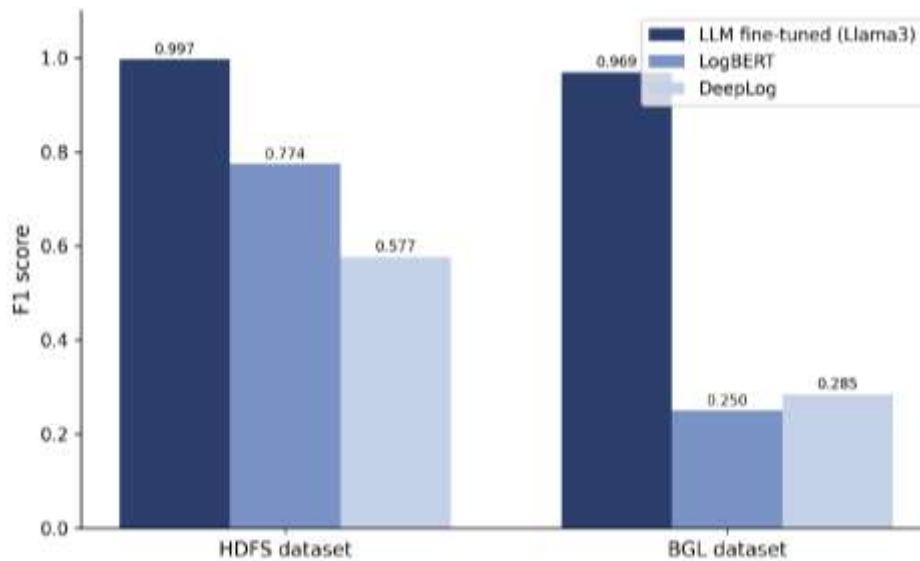
This discussion connects to container orchestration specifically because most cloud-native data platform components run as containerized workloads under an orchestrator responsible for scheduling and resource allocation decisions. The orchestrator's own visibility into resource utilization, request and limit configuration, and actual consumption against those limits functions as a specialized observability domain in its own right, one that intersects with but is not fully subsumed by the metrics, logs, and traces framework discussed in Section 2. Resource utilization observability tends to be treated as an infrastructure operations concern separate from application-level observability, but the two are more entangled in cloud data platforms than this separation suggests, since a pipeline experiencing resource contention at the container level will produce application-level symptoms, elevated latency, timeout errors, and retry storms that are indistinguishable from application-level failures unless resource-level telemetry is examined alongside application traces and logs during diagnosis.

Čilić et al. (2023) provide direct empirical grounding for how much orchestration platform choice affects the observability an operator has into this resource layer under constrained conditions, evaluating four orchestration platforms, Kubernetes, K3s, KubeEdge, and ioFog, in edge computing environments and finding substantial variation in service startup latency and memory overhead depending on platform and controller design. Platforms with heavier controller message processing overhead recovered more slowly from node failure and produced coarser-grained resource telemetry, meaning the choice of orchestration platform shapes both how quickly a system responds to resource pressure and how precisely an operator can observe that pressure building in the first place. In Industry 4.0 containerized environments, Barletta et al. (2024) propose criticality-aware monitoring and orchestration, routing resource and fault signals differently, depending on the downstream consequence of the workload experiencing contention. Their approach treats resource observability as inseparable from prioritization, on the reasoning that a resource-constrained container running a noncritical batch job and one running a latency-sensitive control loop process warrant different monitoring granularity and different escalation paths even when both are technically reporting the same underlying metric of CPU or memory pressure (Shaw, 2026).

Table 2. Capacity and Resource Observability Framework

Observability Question	Signal Source	Operational Decision Informed	Time Horizon
Is current utilization approaching provisioned limits?	Container/orchestrator or resource metrics	Scaling or reprovisioning trigger	Reactive / near-term
Is placement efficient across available nodes?	Orchestrator scheduling telemetry	Placement or scheduler policy adjustment	Near-term
What is the utilization trend over weeks or months?	Aggregated historical metrics	Forward capacity planning and budget forecasting	Forward-looking
Is resource contention producing application-level symptoms?	Cross-referenced container and application traces	Diagnosis routing between infrastructure and application teams	Reactive / diagnostic

Capacity observability also has a forward-looking dimension that anomaly detection, which is fundamentally reactive to deviations from a baseline, does not fully address. Trend analysis over resource utilization metrics allow an organization to forecast when current capacity will become insufficient under projected growth, converting capacity management from a reactive scaling exercise triggered by degraded performance into a planned provisioning decision made ahead of need. This forecasting function depends on the same underlying metrics infrastructure discussed in Section 2 but applies a different analytical lens to it, trend and projection rather than deviation and alert, which is a useful reminder that a single observability signal often serves more than one operational purpose depending on how it is analyzed.



Source: Lim, Zhu, & Pang (2025), arXiv:2503.08045, Table 1.

Figure 2. F1 score comparison for log anomaly detection by method and dataset. Source: Lim et al. (2025).

6. Organizational and Cultural Preconditions

Technical instrumentation and automated detection capability, however sophisticated, do not by themselves guarantee that an organization achieves the reliability outcomes this article has described. Beyer et al.'s (2016) account of site reliability engineering practice at Google emphasizes repeatedly that observability tooling functions within an organizational structure built specifically to use it, defined on-call rotations, blameless postmortem practice, and error budgets that create shared accountability between the teams building a system and the teams operating it. An organization that deploys sophisticated distributed tracing and anomaly detection but has not built the operational practices to act on what those tools surface will not realize the reliability benefits the tooling is capable of providing.

Standardization has emerged as one organizational precondition worth naming explicitly. OpenTelemetry's consolidation as the de facto standard for cloud-native observability instrumentation (Cloud Native Computing Foundation, 2026) reflects a broader industry recognition that metrics, logs, and traces must be correlated across service boundaries using a shared instrumentation vocabulary rather than collected through tool-specific or team-specific conventions. Prior to this kind of standardization, an organization operating a dozen microservices might find each team instrumenting its own service with a different logging format and no shared identifier connecting a request as it traverses multiple services, so that diagnosing a slow request required manually correlating timestamps across disconnected log streams rather than following a single distributed trace end to end. Gartner's (2024) infrastructure and operations management guidance identifies this kind of tooling fragmentation as a persistent maturity gap across organizations broadly, noting that operations teams frequently accumulate observability tooling incrementally, one platform per incident category or per team preference, without a coordinated strategy connecting the resulting signals, which reproduces at an organizational scale the same correlation problem OpenTelemetry's standardization was designed to solve at a technical one.

This approach has a specific implication for the alert fatigue problem introduced in Section 3. Alert fatigue is sometimes framed as a purely technical problem, solvable by better anomaly detection that reduces false positive rates. The organizational literature suggests this framing is incomplete. Even a well-tuned detection system generates alerts that require triage, and if the organizational process for triaging and routing those alerts is not clearly defined, the technical improvement in detection quality does not translate into a proportional improvement in operational outcomes. Observability investment and on-call process design are complementary, not substitutable, and treating the technical and organizational dimensions as separate projects, rather than a single coordinated effort, is a common source of the gap between an organization's observability tooling maturity and its actual reliability outcomes (Rekulapalli et al., 2026).

7. Conclusion

This article has argued that observability in cloud-native data platforms should be understood as an operational capability oriented toward corrective action, not as a monitoring layer whose purpose ends at signal collection or anomaly detection. The Detect-to-Correct framework developed here connects instrumentation practice, drawn from the three observability pillars and their platform-specific application, to automated and human-in-the-loop response mechanisms, arguing that the tightness of this loop, not the sophistication of any single stage within it, determines an observability practice's operational value. Automated anomaly detection and the broader AIOps toolkit have meaningfully advanced the detection and diagnosis stages of this loop, particularly through learned baseline modeling and language model-based log analysis, but these advances only translate into reliability gains when connected to validated response mechanisms, and chaos engineering functions as the validation discipline that confirms this connection holds under realistic failure conditions rather than only in theory. Capacity and resource observability, though sometimes treated as a separate operational domain from anomaly detection and diagnosis, follows the same underlying principle: that observability signals earn their value through the decisions they inform rather than through their volume or granularity alone. Organizations building or operating cloud data platforms should evaluate observability investment against this standard, not against how much telemetry is collected but against how quickly and reliably that telemetry moves from detection through diagnosis to correction.

Conflict of Interest

The author declares no conflict of interest.

Acknowledgment

The author received no external funding or third-party assistance in the preparation of this manuscript beyond the resources cited in the References.

CRediT Author Contributions Statement

Sreekanth Nalla: Conceptualization, Methodology, Investigation, Writing – Original Draft, Writing – Review & Editing, Visualization.

References

1. Aktas, E. U., Tuzlutas, B., & Yesiltas, B. (2025). Designing a custom chaos engineering framework for enhanced system resilience at Softtech. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering* (pp. 797–800). <https://doi.org/10.1145/3756681.3757009>
2. Amazon Web Services. (2024). Reliability pillar – AWS Well-Architected Framework. Available: <https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/welcome.html>
3. Barletta, M., Cinque, M., De Simone, L., & Della Corte, R. (2024). Criticality-aware monitoring and orchestration for containerized Industry 4.0 environments. *ACM Transactions on Embedded Computing Systems*, 23(1), 1–28. <https://doi.org/10.1145/3604567>
4. Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). Site reliability engineering: How Google runs production systems. O'Reilly Media. Available: <https://www.oreilly.com/library/view/site-reliability-engineering/9781491929117/>
5. Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., & Thorne, S. (2018). The site reliability workbook: Practical ways to implement SRE. O'Reilly Media / Google Research. Available: <https://research.google/pubs/the-site-reliability-workbook/>
6. Chagovec, A., Bakardjieva, T., Ivanova, A., Sapundzhi, F., Spasova, V., & Ivanova, A. (2026). AI-driven threat detection and automated incident response for securing cloud workloads. *Applied Sciences*, 16(13), 6454. <https://doi.org/10.3390/app16136454>
7. Čilić, I., Krivić, P., Podnar Žarko, I., & Kušek, M. (2023). Performance evaluation of container orchestration tools in edge computing environments. *Sensors*, 23(8), 4008. <https://doi.org/10.3390/s23084008>
8. Cloud Native Computing Foundation. (2025). State of cloud native development report, Q3 2025. Available: https://www.cncf.io/wp-content/uploads/2025/11/cncf_report_stateofcloud_111025a.pdf
9. Cloud Native Computing Foundation. (2026). Cloud Native Computing Foundation announces OpenTelemetry's graduation, solidifying status as the de facto observability standard. Available:

- <https://www.cncf.io/announcements/2026/05/21/cloud-native-computing-foundation-announces-opentelemetrys-graduation-solidifying-status-as-the-de-facto-observability-standard/>
10. De la Cruz Cabello, M., Prince Sales, T., & Machado, M. R. (2025). AIOps for log anomaly detection in the era of LLMs: A systematic literature review. *Intelligent Systems with Applications*, 2025, 200608. <https://www.sciencedirect.com/science/article/pii/S2667305325001346>
 11. Gaggar, H. (2024). Two Planes of Truth: Execution Fidelity and Behavioral Correctness in Tool-Using LLM Agents. *International Journal of Computational and Experimental Science and Engineering*, 10(3). <https://doi.org/10.22399/ijcesen.5498>
 12. Gartner. (2024). I&O operations management primer for 2024. Available: <https://www.gartner.com/en/documents/5084131>
 13. Google SRE Team. (2018). Implementing SLOs. Google Site Reliability Engineering Workbook. Available: <https://sre.google/workbook/implementing-slos/>
 14. Gremlin. (2021). State of chaos engineering 2021. Available: <https://www.gremlin.com/state-of-chaos-engineering/2021>
 15. Jernberg, H., Runeson, P., & Engström, E. (2020). Getting started with chaos engineering – design of an implementation framework in practice. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–10). <https://doi.org/10.1145/3382494.3421464>
 16. Lim, W., Zhu, Z., & Pang, G. (2025). Adapting large language models for parameter-efficient log anomaly detection. *arXiv*. <https://arxiv.org/abs/2503.08045>
 17. Majors, C., Fong-Jones, L., & Miranda, G. (2022). Observability engineering: Achieving production excellence. O'Reilly Media. Available: <https://www.oreilly.com/library/view/observability-engineering/9781492076438>
 18. Meng, K., Li, M., Ding, J., & Zhou, H. (2026). Cloud disaster recovery model based on failure prediction. *Journal of Cloud Computing*, 15(1), 33. <https://doi.org/10.1186/s13677-026-00846-0>
 19. Microsoft Ignite. (2024). Azure Well-Architected Framework: Operational excellence quick links. Available: <https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/>
 20. New Relic. (2023). Observability forecast 2023: Service-level metrics. Available: <https://newrelic.com/resources/report/observability-forecast/2023/state-of-observability/service-level-metrics>
 21. OpenTelemetry Project. (2026). OpenTelemetry documentation. Cloud Native Computing Foundation. Available: <https://opentelemetry.io/docs/>
 22. Ouyang, M., Xi, J., Bai, W., & Li, K. (2022). Band-area resource management platform and accelerated particle swarm optimization algorithm for container deployment in internet-of-things cloud. *IEEE Access*, 10, 86844–86863. <https://ieeexplore.ieee.org/abstract/document/9857907>
 23. Rekulapalli, K., Nilagiri, H., & Vakkanthula, M. (2026). A comparative evaluation of national FHIR Implementation Guides: strategic methodologies for enhancing interoperability, regulatory alignment, and clinical comprehensiveness in global health IT. *Health Systems*, 1–13.
 24. Shaw, R. K. (2026, July). Kerberos vs. Public Key Authentication in z/OS Environments: Performance, Scalability, and Security Trade-offs. In *2026 9th International Conference on Computing Methodologies and Communication (ICCMC)* (pp. 1305–1312). IEEE.
 25. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report. Available: <https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/>
 26. Uber Engineering. (2023). uVitals: An anomaly detection and alerting system. Uber Engineering Blog. Available: <https://www.uber.com/blog/uvitals-an-anomaly-detection-alerting-system/>
 27. Voutsas, F., Violos, J., & Leivadeas, A. (2024). Mitigating alert fatigue in cloud monitoring systems: A machine learning perspective. *Computer Networks*, 250. <https://doi.org/10.1016/j.comnet.2024.110543>
 28. Zhang, C., Dong, Z., Peng, X., Zhang, B., & Chen, M. (2024). Trace-based multi-dimensional root cause localization of performance issues in microservice systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. <https://doi.org/10.1145/3597503.3639088>