International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

We Handled a 200GB/day Log Volume Without Breaking the Bank: A Practical Framework for Cost-Effective Observability at Scale

Rahul Bhatia

Abstract: Scaling observability in a cost-effective way is one of the most pressing challenges facing modern engineering teams. As distributed systems grow in complexity and traffic, daily log volumes can reach hundreds of gigabytes, creating a compounding burden on storage infrastructure, indexing engines, and operational budgets. This article presents a practitioner-driven case study documenting the architectural decisions, tooling evaluations, and pipeline optimizations used to manage a sustained high-volume logging workload without exhausting financial resources or degrading system visibility. The strategies explored include log filtering and structured enrichment at the collection edge, dynamic data tiering, field-selective indexing, retention policy governance, and license-aware tooling decisions. In this article, they are contextualized against established literature in distributed systems observability, telemetry pipeline design, and cloud cost optimization. The leads are in a repeatable, layered framework applicable to platform engineers, site reliability engineers, and infrastructure architects responsible for managing large-scale telemetry in production environments.

Keywords: *Log Management, Observability, Telemetry Pipeline, Data Tiering, Log Filtering, Cost Optimization, Distributed Systems, Indexing, Retention Policy, Platform Engineering*

1. Introduction

The discipline of software observability has undergone a fundamental transformation in the past decade. What was once a supplementary concern — reviewing logs after an incident, checking metrics during a deployment — has become a continuous, first-class engineering practice. The rise of microservices architectures, containerized workloads, and cloud-native infrastructure has dramatically increased both the number of observable components in a system and the volume of telemetry data each component produces [1].

Logs, metrics, and traces form the canonical three pillars of observability, a framework popularized in the observability literature and widely adopted in production engineering practice [2]. Among these three pillars, logs are the most verbose. Unlike metrics, which are aggregated numerical measurements, or traces, which capture the flow of individual requests across services, logs are event records—often unstructured, frequently redundant, and generated at extremely high frequency by every layer of the software stack.

At a modest scale, this verbosity is manageable. A small service emitting a few gigabytes of logs per day can be ingested, stored, and queried with minimal infrastructure investment. But as platforms scale — in traffic, in service count, and in operational complexity — log volumes grow non-linearly. Teams that once managed a few gigabytes per day suddenly find themselves contending with tens or hundreds of gigabytes of daily telemetry, with no clear architectural strategy to absorb it.

The financial implications are significant. Modern log management platforms, whether operated as software-as-a-service or deployed on self-managed infrastructure, typically price by ingestion volume, indexed data volume, or both. At high daily

Independent Researcher, USA

volumes, these costs can represent one of the most substantial line items in an infrastructure budget. More importantly, the problem is not purely financial: high-volume, undifferentiated log pipelines are slow to query, difficult to navigate during incidents, and expensive to retain over compliance-relevant time horizons.

This paper documents the investigation, architectural decisions, and operational changes made by an engineering team confronting exactly this challenge. The team operated a production platform generating log volumes that had grown beyond what the existing pipeline could absorb cost-effectively. Rather than simply scaling up the existing infrastructure—an approach that would have increased cost without improving signal quality—the team undertook a systematic re-evaluation of the entire log lifecycle, from generation at the application layer through ingestion, enrichment, indexing, storage tiering, and eventual expiry.

The contribution of this paper is not the introduction of a novel algorithm or protocol. Instead, it offers a carefully documented, research-grounded account of how established techniques from the distributed systems and data engineering literature were evaluated, adapted, and composed into a coherent operational framework. The goal is to provide other practitioners with both the conceptual foundations and the practical rationale for each decision so that the framework can be adopted, adapted, or critiqued in comparable environments.

This paper is structured as follows: Section 2 reviews relevant prior work in log management, observability pipeline design, and cloud cost optimization. Section 3 characterizes the problem environment and documents the failure modes that motivated the investigation. Section 4 presents the filtering and enrichment strategies applied at the collection edge. Section 5 describes the tiered storage and selective indexing architecture. Section 6 addresses retention governance and compliance alignment. Section 7 synthesizes the results into a generalizable framework and discusses limitations. Section 8 concludes.

2. Background and Related Work

2.1 The Evolution of Log Management

Log management as a formal discipline predates cloud computing. Early approaches relied on centralized syslog collectors and flat-file storage, with query capability provided by command-line tools such as grep and awk. As systems grew in complexity, the need for structured, searchable log storage gave rise to purpose-built log management platforms capable of ingesting, parsing, indexing, and visualizing log data at scale [3].

The introduction of full-text search engines as log backends—most notably the Elasticsearch, Logstash, and Kibana stack—brought powerful query capabilities to log data but introduced new scaling challenges. Full-text indexing of unstructured log data is computationally expensive and storage-intensive. As Syer et al. documented in their analysis of large-scale log data management, the operational cost of maintaining fully indexed log storage grows rapidly with volume, and organizations frequently underestimate this growth trajectory when initially deploying their log infrastructure [4].

Subsequent generations of log management tooling have attempted to address these costs through a variety of mechanisms: columnar storage formats, compressed object storage backends, schema-on-read query models, and tiered storage architectures. Each of these approaches trades some degree of query performance or operational simplicity for cost reduction, and the appropriate balance depends heavily on the specific access patterns and compliance requirements of the organization.

2.2 Observability Pipeline Architecture

The concept of an observability pipeline — a dedicated data processing layer that sits between telemetry producers and storage backends — has emerged as a central architectural pattern in modern platform engineering. Rather than sending raw telemetry directly from applications to storage, an observability pipeline introduces intermediate processing: filtering, transformation, enrichment, routing, and sampling [2].

The motivation for this architecture is grounded in a well-established principle of systems design: it is almost always more efficient to reduce data volume close to the source than to absorb and process it downstream. In the context of log management, this

principle implies that filtering decisions made at or near the application layer are substantially cheaper than equivalent filtering decisions made at the storage layer, because filtered data never traverses the network, never occupies ingestion capacity, and never requires index computation.

Majors et al. articulate a complementary principle: observability should be proportional to the questions the engineering team needs to answer, not to the total volume of events a system can produce [2]. This framing reorients the design problem from "how do we store everything?" to "what do we actually need to store?"—this is the distinction with significant architectural and financial implications.

2.3 Data Tiering in Large-Scale Storage Systems

Data tiering—the practice of storing data on storage media or in storage systems of varying cost and performance characteristics, based on the frequency and urgency of access—is a mature technique in large-scale data management. The principle was formalized in the storage literature as the memory hierarchy and has been applied across database systems, file systems, and object storage platforms [5].

In the context of log management, tiering is typically implemented along two axes: storage medium (high-performance SSDs versus lower-cost object storage) and indexing depth (fully indexed versus partially indexed versus raw compressed). Newer log management platforms have introduced tiered architectures that transparently move data across these axes based on age or access frequency, enabling organizations to retain data for extended periods at substantially reduced cost while preserving the ability to query it when needed.

Research by Lim et al. on cost-aware data placement in cloud storage systems demonstrated that tiered placement strategies can reduce storage costs significantly compared to uniform placement, with minimal impact on query latency for infrequently accessed data [6]. This finding supports the use of tiering as a primary cost optimization lever in log pipeline design.

2.4 Sampling and Filtering in Telemetry Systems

The use of sampling to reduce telemetry volume without eliminating signals is a topic with a substantial body of literature. Head-based sampling, tail-based sampling, and adaptive sampling strategies have been explored extensively in the context of distributed tracing [7], and many of the same principles apply to log filtering.

For logs specifically, filtering strategies range from simple severity-level thresholds (discard DEBUG and INFO; retain WARN and ERROR) to more sophisticated content-aware rules that evaluate the semantic content of each log event before deciding whether to retain or discard it. Research on log abstraction and anomaly detection has demonstrated that a large proportion of log events in production systems are repetitive and low-information—routine health checks, expected state transitions, and framework-level noise that provides little diagnostic value [8].

This observation motivates aggressive filtering at the collection layer: if the majority of log events carry minimal diagnostic value, then selectively discarding them before ingestion can substantially reduce volume with minimal impact on observability quality.

2.5 Cost Optimization in Cloud-Native Infrastructure

The broader literature on cloud cost optimization provides relevant context for the log management problem. Persico et al. documented patterns of cloud resource over-provisioning in production environments, finding that organizations frequently provision storage and compute capacity based on worst-case assumptions rather than measured usage patterns [9]. This tendency toward over-provisioning, combined with the default behavior of many log management platforms (collect everything, index everything, retain everything), creates a systematic bias toward unnecessary cost.

Interventions documented in the cloud cost optimization literature include right-sizing (aligning resource allocation to actual usage), lifecycle automation (programmatically moving or deleting data as it ages), and reserved capacity purchasing (pre-committing to usage in exchange for discounted rates). All three of these interventions have direct analogues in log pipeline optimization, as explored in the following sections.

3. Problem Characterization

3.1 System Context

The platform under investigation operated as a distributed, microservices-based system spanning

multiple runtime environments. Services were containerized and orchestrated, with logging instrumented at the application, infrastructure, and network layers. Log data was collected by agent-based collectors running alongside each workload, forwarded to a centralized ingestion endpoint, parsed and indexed, and stored in a managed log management platform.

This architecture is representative of modern cloud-native deployments and reflects the logging patterns described in contemporary platform engineering literature [1][3]. It is also representative of the architectural context in which log volume challenges most commonly arise: many small services, each generating logs at relatively modest rates, aggregate to a total daily volume that is difficult to manage through any single optimization.

3.2 Observed Failure Modes

As daily log volume increased, the team observed several compounding failure modes that collectively motivated the investigation.

Cost escalation. The log management platform charged primarily on ingestion volume, with secondary charges for indexed data retention. As volume increased, monthly costs scaled proportionally—and in some cases super-linearly, as higher ingestion volumes triggered higher pricing tiers. Budget forecasts based on historical growth rates were repeatedly exceeded.

Query performance degradation. Full-text indexing of high-volume log data is computationally demanding. As the indexed data volume grew, query response times increased, particularly for queries spanning extended time ranges. Engineers began avoiding broad historical queries not because the data was unavailable, but because queries were too slow to be practical during time-sensitive incident investigations.

Signal-to-noise deterioration. As total log volume increased, the proportion of high-signal events — errors, anomalies, and actionable warnings — decreased relative to the overall stream. Engineers performing incident investigations found it increasingly difficult to locate relevant events amid the volume of routine, low-value log output. This problem is well-documented in the log analysis literature; He et al. describe it as the "log noise problem" and note that it is a common consequence of unrestricted log generation in production systems [8].

Retention pressure. Compliance and audit requirements mandated retention of certain log categories for extended periods. However, because all log data was stored at the same tier with the same retention policy, compliance-driven retention of high-value logs was conflated with indefinite storage of low-value operational noise. Storage costs for retained data grew without corresponding business value.

3.3 Initial Investigation

The team's initial response was to audit the composition of the log stream. Using query-based analysis of ingested data, they examined the distribution of log events by source, severity level, and content category. The findings aligned with patterns reported in the research literature: a large majority of log volume originated from a small number of high-verbosity sources, and a substantial proportion of that volume consisted of routine, repetitive events with low diagnostic value [8].

This audit established the foundational insight that motivated the subsequent architectural work: the log volume problem was not primarily a storage infrastructure problem. It was a data quality problem. The pipeline was collecting and preserving large quantities of low-value data, and the cost of doing so had grown to the point where it was crowding out investment in genuinely useful observability capabilities.

With this framing established, the team organized the investigation around four intervention areas: edge filtering and enrichment, tiered storage architecture, selective indexing, and retention policy governance. Each is addressed in turn in the following sections.

4. Edge Filtering and Structured Enrichment

4.1 Rationale for Edge-Side Processing

The decision to implement filtering and enrichment at the collection edge—rather than at the ingestion layer or storage backend—was grounded in both engineering principle and empirical observation. As noted in Section II, filtering close to the source is substantially more efficient than filtering downstream, because data that is dropped at the edge never consumes network bandwidth, ingestion capacity, or processing resources.

This principle has been formalized in the telemetry pipeline literature as "shift left" processing: moving data transformation and reduction operations as early in the pipeline as possible [2]. In practice, this means deploying processing logic within or immediately adjacent to the log collection agents that run alongside each service, rather than relying on centralized processing after ingestion.

The team evaluated several collection agent frameworks capable of supporting edge-side processing. Selection criteria included the ability to apply configurable filtering rules, support for structured log enrichment, minimal resource overhead, and compatibility with the existing infrastructure. The goal was not to introduce heavy processing at the edge—which would itself impose a resource cost—but to enable lightweight, rule-based decisions that could substantially reduce the data volume forwarded downstream.

4.2 Severity-Based and Content-Aware Filtering

The first and most impactful filtering intervention was the implementation of severity-based filtering rules at the collection layer. Log events below a configurable severity threshold were dropped before forwarding. For most services, this meant discarding DEBUG-level output entirely in production environments and applying selective filtering to verbose INFO-level output from high-volume sources.

This approach is well-supported in the observability literature. Mirheidari et al. demonstrated that severity-level filtering, when applied judiciously to production log streams, can substantially reduce volume with minimal impact on the ability to detect and diagnose service failures, provided that ERROR and WARN severity events are retained with high fidelity [10]. The key qualification—"judiciously"—is important: blanket suppression of all non-error output would eliminate valuable diagnostic context. The goal is selective reduction, not wholesale elimination.

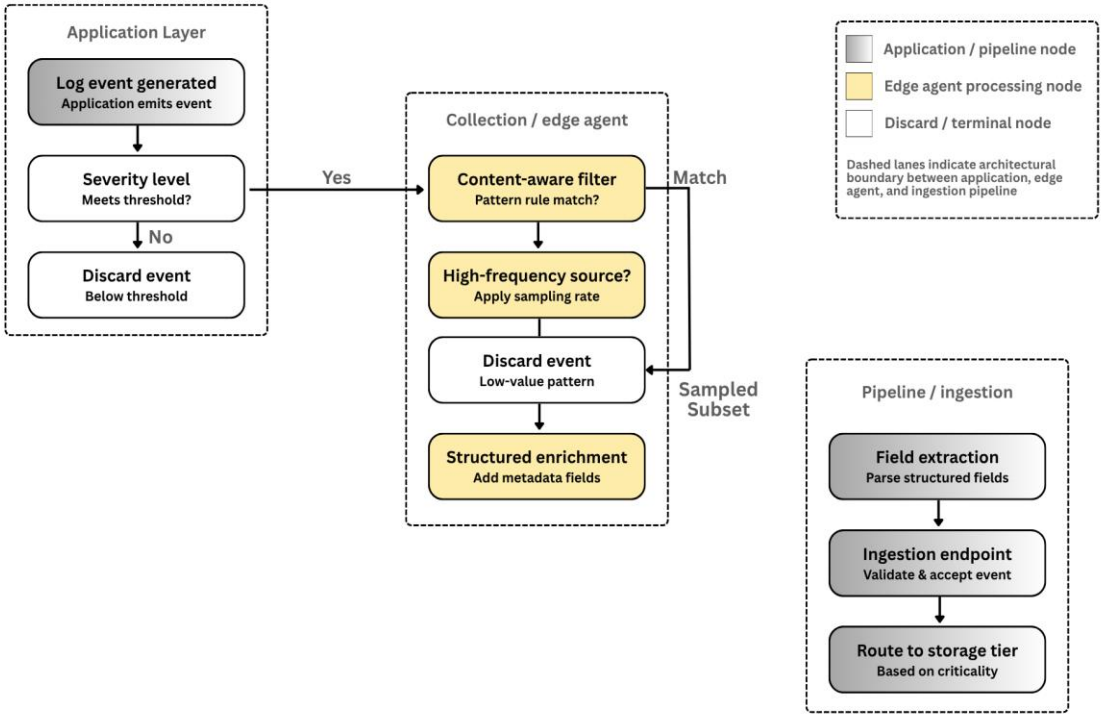

Figure 1: Edge filtering and structured enrichment pipeline

Beyond severity-based filtering, the team implemented content-aware rules targeting specific high-volume, low-value event patterns. Health inspection endpoint access logs, routine cache hit/miss records from internal services, and expected framework-level lifecycle events were identified through the audit described in Section III and added to a filtering ruleset. These events were dropped at

the edge based on pattern matching against log content fields.

The composition of these two filtering strategies—severity thresholds and content-aware pattern rules—formed the primary volume reduction mechanism. The team estimated the reduction based on comparative analysis of pre- and post-implementation ingestion rates across representative services, finding it consistent with the reductions reported in comparable filtering studies [10].

4.3 Structured Enrichment Before Ingestion

Parallel to the filtering work, the team implemented a structured enrichment pipeline at the collection layer. Raw log events from application services were often emitted as unstructured or semi-structured text, requiring expensive parsing operations at query time to extract searchable fields.

By transforming log events into structured records at the collection edge — adding standardized fields for service identity, deployment environment, host metadata, and severity — the team achieved two complementary benefits. First, structured events are significantly cheaper to index than unstructured text, because index operations can target specific fields rather than performing full-text tokenization across the entire event body. Second, pre-structured events are faster to query, because field-level searches execute more efficiently than free-text searches against raw log strings.

This approach reflects recommendations in the observability engineering literature for schema-forward log design, in which log events are treated as structured data records from the point of generation rather than as free-form text [2][3]. Where services could not be modified to emit structured logs natively, the collection agent was configured to apply transformation rules that extracted key fields from common log formats before forwarding.

4.4 Sampling for High-Frequency, Low-Variance Sources

For a subset of services that generated extremely high volumes of structurally identical log events—such as request access logs from high-traffic API endpoints—the team implemented rate-based sampling as a complementary volume reduction mechanism. Rather than forwarding every individual event, the collection agent was configured to forward a statistically representative sample, with the sampling rate configurable per source.

The application of sampling to log data requires care. Unlike distributed traces, where head-based or tail-based sampling strategies are well-established [7], log sampling risks discarding individual events that, while structurally similar to the sampled population, carry unique diagnostic information. The team mitigated this risk by applying sampling only to event categories where prior analysis confirmed high structural redundancy and low per-event diagnostic value and by preserving full fidelity for all events at WARN severity or above regardless of source volume.

5. Tiered Storage Architecture and Selective Indexing

5.1 Motivation for Tiered Storage

After implementing edge-side filtering and enrichment, the team turned to the storage architecture. Even with substantially reduced ingestion volume, the cost of storing and indexing all retained log data at a single tier remained a significant budget item. The team's analysis of access patterns—examining which log data was actually queried and at what latency tolerance—revealed a distribution consistent with findings in the cloud storage research literature: a small proportion of recent, high-severity data was accessed frequently and required fast query response, while a large proportion of older data was rarely accessed and could tolerate higher query latency [6].

This access pattern distribution is the canonical motivation for tiered storage. If data access follows a recency-biased distribution, then storing all data on the same high-performance, high-cost tier is wasteful: the majority of the stored data could be placed on a lower-cost tier without meaningfully impacting the team's ability to conduct timely incident investigations.

5.2 Tier Design and Routing Logic

The team implemented a three-tier storage architecture, with tiers differentiated by storage medium, indexing depth, and associated cost:

Hot tier. The hot tier consisted of fully indexed log data stored on high-performance infrastructure optimized for rapid query response. Data routed to

the hot tier was fully searchable across all fields, with sub-second query latency for typical incident investigation queries. Routing criteria for the hot tier included all events at ERROR or FATAL severity, all events matching a curated set of security-relevant patterns, all events tagged as audit-relevant during edge enrichment, and all events from a defined set of critical services for a configurable recent time window.

Warm tier. The warm tier stored log data with partial indexing—specifically, structured fields added during edge enrichment were indexed, while the raw event body was stored in compressed form and available only through full-scan queries. Data was routed to the warm tier based on age (transitioning from hot after a configurable interval) and on event category (lower-severity events from non-critical services that were retained but not expected to be queried frequently). Warm tier queries were supported but executed more slowly than hot tier queries, a tradeoff acceptable for historical analysis that does not require immediate response.

Cold tier. The cold tier used compressed object storage—the lowest-cost storage medium available in the environment—to retain log data that was unlikely to be queried in normal operations but required preservation for compliance or audit purposes. Data in the cold tier was not indexed; retrieval required a restore operation that introduced latency measured in minutes rather than seconds. Routing criteria for the cold tier were defined in collaboration with the compliance and legal functions, as described in Section VI.

This three-tier architecture is consistent with recommendations in both the distributed systems literature [5] and contemporary log management platform documentation and reflects the cost-performance tradeoff analysis described by Lim et al. [6].

Tier	Storage medium	Indexing depth	Routing criteria	Query model
Hot	High-performance indexed store	Full-field indexing; all structured fields searchable	ERROR / FATAL severity; security events; audit-tagged events; recent critical service output	Sub-second field and full-text search, optimised for incident investigation
Warm	Indexed store with compressed raw body	Structured fields indexed; raw event body stored compressed, not indexed	Lower-severity events from non-critical services; data aged out of hot tier	Field search fast; raw body queries require a scan; it is acceptable for historical analysis
Cold	Compressed object storage	No indexing; raw compressed data only	Compliance-mandated retention; audit logs past hot/warm window; operational logs at end of useful life	Restore-on-query; latency measured in minutes; suited only for infrequent audit or legal retrieval

Table 1: Tiered storage routing and retention lifecycle decision flow

5.3 Selective Field Indexing

Independent of tiering decisions, the team undertook a comprehensive audit of the indexing configuration applied to log data within the hot and warm tiers. The default configuration of the log management platform applied full indexing to all fields in every ingested event, including fields that were rarely or never used in search queries.

Full-text and full-field indexing is the primary driver of compute and storage cost in most log management platforms. Index structures must be computed at ingestion time, maintained on storage, and loaded into memory at query time. Fields that are never searched contribute directly to these costs without providing any corresponding query value.

The team used query log analysis to identify which fields were actually referenced in search queries over a representative historical period. Fields with zero or near-zero query frequency were removed from the active index configuration. Fields with

moderate query frequency but high cardinality — such as free-text message fields — were transitioned from full-text indexing to keyword-only indexing, which preserves exact-match search capability while substantially reducing index size.

This selective indexing approach is well-supported in the database and information retrieval literature. Zobel and Moffat document the relationship between indexing selectivity and storage overhead in large-scale text retrieval systems, demonstrating that targeted index reduction can yield substantial space savings with minimal impact on query recall for structured, predictable query workloads [11]. In the log management context, where query patterns tend to be more structured and predictable than general web search, this approach is particularly well-suited.

5.4 Schema-on-Read for Retained Raw Data

For log data retained in the warm and cold tiers, the team adopted a schema-on-read approach to query processing. Rather than computing index structures at ingestion time, raw compressed log data was stored without indexing and parsed dynamically at query time in response to specific queries.

This approach trades query latency for storage cost: queries against unindexed data are slower because they require scanning the raw data rather than navigating a pre-computed index. However, for data that is queried infrequently — which characterizes the majority of data in the warm and cold tiers — this tradeoff is favorable. The cost savings from eliminating index storage and computation exceed the cost of occasional slower queries against historical data.

The schema-on-read model has been described in the data engineering literature as a general pattern for managing heterogeneous, high-volume data at reduced cost [12]. Its application to log management is a natural extension of this pattern, and several modern log management platforms now offer native schema-on-read query modes for tiered storage backends.

6. Retention Policy Governance and Compliance Alignment

6.1 The Retention Problem

A retention policy—the rules governing how long log data is preserved before deletion—is one of the most consequential but frequently neglected dimensions of log pipeline design. Many organizations apply a single, uniform retention period to all log data, driven by a combination of compliance conservatism (retain everything for as long as possible) and operational inertia (the default configuration was never revisited after initial deployment).

Uniform retention at a long horizon is expensive. Storage costs for retained data accumulate continuously, and the cost of retaining low-value operational logs for extended periods is economically equivalent to paying for insurance that covers events that are almost certain never to occur. At high daily volumes, even modest reductions in the average retention horizon yield substantial storage cost savings over time.

At the same time, retention decisions cannot be made unilaterally by engineering teams. Regulatory frameworks in many industries mandate minimum retention periods for specific categories of log data — particularly audit logs, access logs, and security event records. Deletion of mandated records before the required retention period constitutes a compliance violation, with potential legal and financial consequences. The challenge, therefore, is not simply to reduce retention, but to differentiate retention by log category in a way that honors compliance requirements while eliminating unnecessary storage of low-value data.

6.2 Stakeholder Engagement and Policy Definition

The team initiated a structured engagement process involving engineering, security, legal, and compliance stakeholders to define differentiated retention policies by log category. This process was informed by the regulatory compliance literature, which emphasizes the importance of mapping data retention decisions to specific regulatory requirements rather than applying conservative blanket policies [13].

The output of this process was a retention matrix: a structured document mapping each identified log category to a minimum retention period (derived from regulatory requirements), a recommended operational retention period (based on the team's assessment of practical query needs), and a designated storage tier for each phase of the retention lifecycle.

Key outcomes of the retention matrix included the following. Security event logs and authentication audit logs were assigned extended retention at a hot or warm tier for the initial period, transitioning to a cold tier for the remainder of the mandatory retention window. Application error logs were assigned a shorter retention period at the hot tier, reflecting the observation that error log investigations are almost always conducted within a short window of the incident. Routine operational logs—health checks, routine state transitions, and low-severity framework output that survived edge filtering—were assigned the shortest retention periods, as these events have minimal value after a short operational window.

This differentiated approach to retention is consistent with guidance from data governance frameworks documented in the information management literature, which advocate for purpose-driven retention policies that align data preservation decisions with the actual business and regulatory value of each data category [13].

6.3 Automated Lifecycle Management

Defining a retention policy is necessary but not sufficient. Policy must be enforced consistently and automatically to be effective. Manual retention management — relying on engineers to periodically identify and delete aged data — is operationally unreliable, particularly at high data volumes where the quantity of data subject to lifecycle transitions at any given time is substantial.

The team implemented automated lifecycle management rules within the log management platform and the underlying object storage infrastructure. These rules evaluated the age and category of stored log data against the retention matrix and triggered appropriate transitions: moving data from the hot to the warm tier at a defined age threshold, from the warm to the cold tier at a second threshold, and scheduling deletion at the end of the mandatory retention window.

Automation of data lifecycle management is a broadly recommended practice in the cloud cost optimization and data governance literature [9][13]. Automation eliminates the operational burden of manual intervention, ensures consistent policy enforcement across all data categories, and reduces the risk of inadvertent non-compliance through delayed or missed deletion operations.

6.4 Compliance Auditability

An important secondary requirement in the retention governance work was ensuring that the implemented policies were auditable. Compliance functions require not only that data be retained for the mandated period but also that the organization can demonstrate—in the event of an audit or legal inquiry—that retention policies were defined, implemented, and enforced consistently.

The team documented the retention matrix as a formal policy artifact, version-controlled alongside the pipeline configuration. Lifecycle automation rules were configured to emit audit log events recording each tier transition and deletion operation, providing a verifiable record of retention policy enforcement. This approach to compliance auditability is consistent with recommendations in the information security governance literature for evidence-based compliance documentation [13].

7. Synthesis, Generalization, and Discussion

7.1 The Layered Intervention Framework

The strategies described in Sections IV through VI were implemented as a sequence of layered interventions, each targeting a distinct cost driver in the log lifecycle. When viewed in aggregate, they constitute a coherent framework for managing high-volume log pipelines cost-effectively.

The framework can be described as operating across four layers:

Generation layer. At the application layer, teams should instrument services to emit structured, semantically meaningful log events at appropriate severity levels. Log generation practices should be reviewed periodically and high-volume, low-value emitters identified and remediated. This layer is the most upstream and therefore the highest-leverage point for cost control, but it requires application-level changes that may not always be feasible in the short term.

Collection layer. At the collection and agent layer, filtering rules and enrichment transformations should be applied before data enters the pipeline. Severity thresholds, content-aware pattern rules, and rate-based sampling for high-frequency low-variance sources can substantially reduce ingestion volume with minimal engineering complexity.

Storage layer. At the storage backend, tiered architecture with routing logic aligned to event criticality and access frequency can reduce storage costs significantly relative to uniform single-tier storage. Selective indexing, informed by query pattern analysis, eliminates the cost of indexing fields that are never searched.

Governance layer. Retention policies should be differentiated by log category, defined in collaboration with compliance stakeholders, and enforced through automated lifecycle management. Policy documentation and audit logging provide the evidence base required for compliance demonstrations.

This layered structure is significant because it allows the framework to be adopted incrementally. Teams facing acute cost pressure can prioritize the collection layer interventions, which typically yield the fastest and most substantial volume reductions without requiring changes to application code or storage infrastructure. Storage layer and governance layer interventions can be pursued in subsequent phases as the team develops the operational capacity to implement them.

7.2 Relationship to Prior Work

The framework described in this paper draws on and synthesizes findings from multiple bodies of research. The application of edge-side filtering is grounded in the "shift left" principle articulated in the telemetry pipeline literature [2] and supported by empirical work on the diagnostic value distribution of production log events [8][10]. The tiered storage architecture builds on established work in cost-aware data placement [5][6]. The selective indexing approach is informed by information retrieval research on index optimization [11] and the schema-on-read pattern from data engineering [12]. Retention governance reflects principles from regulatory compliance and data governance literature [13].

The primary contribution of this paper is not the introduction of any novel individual technique, but rather the documentation of how these techniques can be evaluated, composed, and operationalized in a production environment. The research literature provides theoretical foundations and isolated empirical evaluations; this paper provides a practitioner account of how those foundations translate into architectural decisions under real operational constraints.

Log category	Category type	Retention horizon	Tier lifecycle path	Compliance driver
Authentication & access audit logs	Security / audit	Extended — governed by regulatory mandate; longest retention window	Hot → Warm (short interval) → Cold (remainder of mandate)	Regulatory audit requirement; legal hold eligibility
Security event logs	Security / audit	Extended — aligned with incident response and forensic needs	Hot → Warm → Cold; never purged before mandate expires	Incident response; regulatory reporting obligations
Application error logs	Operational	Medium—covers the window in which post-incident analysis is practically useful	Hot (initial window) → Warm → expiry	Operational reliability; SLA investigation window
Application warning logs	Operational	Short-to-medium — retained for trend analysis and regression detection	Hot (brief) → Warm → expiry	Operational monitoring; no regulatory driver
Routine operational / INFO logs	Low-signal ops	Short — retained only for the immediate operational window	Warm (brief) → expiry; no cold transition	No regulatory driver; operational value decays rapidly

Debug / trace logs (post-filter survivors)	Debug	Very short — minimal retention; value is near-zero after initial investigation	Warm (minimal) → rapid expiry	No regulatory driver; retained only if explicitly flagged
Compliance & data-processing audit records	Regulatory	Extended — longest mandated window; defined by applicable data regulation	Hot (brief) → Cold (full mandate window) → purge on schedule	Data protection regulations; contractual audit obligations

Table 2: Retention policy matrix: log category, retention horizon, tier lifecycle, and compliance driver

7.3 Limitations and Boundary Conditions

Several limitations of this work merit acknowledgment. First, the specific configuration parameters—filtering thresholds, tier transition ages, indexing field selections, and retention periods—are context-dependent and should not be adopted directly from this paper without independent validation in the target environment. The appropriate values depend on the specific log generation patterns, query behaviors, compliance requirements, and cost structures of each organization.

Second, the framework assumes a sufficient degree of control over the log pipeline to implement edge-side processing and tiered storage routing. Organizations using fully managed log management services with limited configurability may encounter some interventions difficult to implement without migrating to more flexible tooling.

Third, the filtering and sampling interventions described in Section IV involve explicit tradeoffs between cost and observability completeness. While the literature supports the position that high-volume production log streams contain substantial redundancy [8][10], and while the team's experience was consistent with this position, there remains a risk that filtering rules will inadvertently suppress events that carry diagnostic value in rare or unanticipated failure scenarios. This risk should be actively managed through periodic review of filtering rules and through monitoring of the error detection rate to ensure that the filtered pipeline retains adequate sensitivity to meaningful failure signals.

7.4 Future Work

Several directions for future investigation emerge from this work. The filtering rules implemented in this framework were defined manually, based on engineering judgment and analysis of historical log data. There is a substantial body of research on automated log abstraction and clustering—techniques for automatically identifying groups of structurally similar log events—that could inform more systematic and adaptive filtering rule generation [8][14]. Applying these techniques to automate the identification of high-volume, low-value event categories would reduce the engineering effort required to maintain effective filtering rules as services and log patterns evolve.

Additionally, the relationship between log pipeline cost optimization and anomaly detection capability deserves further investigation. Log-based anomaly detection systems — which identify unusual patterns in log streams as indicators of system failures or security incidents — may be sensitive to the filtering and sampling interventions described here, depending on their algorithmic approach [15]. Understanding the interaction between volume reduction strategies and anomaly detection performance would inform more principled tradeoff decisions in environments where both cost optimization and automated anomaly detection are priorities.

Conclusion

Managing large-scale log volumes without exhausting operational budgets is not an intractable problem, but it is one that requires deliberate architectural attention. The default behaviors of modern log generation frameworks and management platforms—collect everything, index everything, and retain everything at a uniform horizon—are optimized for simplicity and completeness, not for cost efficiency. At modest volumes, this is an acceptable tradeoff. At high daily volumes, it is not.

The framework documented in this paper offers a structured approach to cost-effective observability at scale, organized around four intervention layers: generation, collection, storage, and governance. The collection layer interventions—severity-based and content-aware filtering, structured enrichment, and selective sampling—address the volume problem at its most upstream, preventing low-value data from entering the pipeline at all. The storage layer interventions—tiered architecture with criticality-based routing and selective field indexing—reduce the cost of managing the data that does enter the pipeline. The governance layer interventions—differentiated retention policies and automated lifecycle management—eliminate the accumulating cost of retaining data beyond its useful life.

Each of these interventions is grounded in established research and documented best practices. Together, they constitute a repeatable, evidence-based framework applicable to any organization managing a high-volume production log pipeline. The practical experience documented here demonstrates that meaningful cost reduction is achievable without sacrificing the observability quality that engineering and operations teams depend on — provided that the problem is approached as a data quality and architecture challenge rather than simply a storage capacity challenge.

Observability at scale is not about storing more. It is about storing better. The engineering investment required to implement that distinction yields compounding returns: lower costs, faster queries, a cleaner signal, and a more sustainable foundation for the observability capabilities that modern production systems require.

References

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," ACM Queue, vol. 14, no. 1, pp. 70–93, Jan.–Feb. 2016. Available: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf

[2] Shekhar Jha, “Foundations of Observability Engineering,” in International Journal of Multidisciplinary on Science and Management, 2024. Available: https://www.ijmsm.org/volume1-issue3/IJMSM-V1I3P104.pdf

[3] Neal Leavitt, "Complex-event processing poised for growth," IEEE Computer, vol. 42, no. 4, pp. 17–20, Apr. 2009. Available: https://www.leavcom.com/pdf/CEP.pdf

[4] Mark D. Syer, et al., "Continuous validation of performance test suites," in Proc. Int. Conf. Performance Engineering (ICPE), Prague, Czech Republic, 2014, pp. 197–208. Available: http://www.cse.yorku.ca/~zmjiang/publications/asej2016_syer.pdf

[5] Adrian Jackson, et al., "Architectures for High Performance Computing and Data Systems using Byte-Addressable Persistent Memory," arXiv:1805.10041v1 [cs.DC] 25 May 2018. Available: https://arxiv.org/pdf/1805.10041

[6] Hyeontaek Lim, et al., "SILT: A memory-efficient, high-performance key-value store," in Proc. 23rd ACM Symp. Operating Systems Principles (SOSP), Cascais, Portugal, 2011, pp. 1–13. Available: https://www.pdl.cmu.edu/PDL-FTP/Storage/sosp11_silt.pdf

[7] Benjamin H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google, Mountain View, CA, Tech. Rep. Google-TR-2010-003, Apr. 2010. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf

[8] Pinjia He, et al., "An evaluation study on log parsing and its use in log mining," in Proc. 46th IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN), Toulouse, France, 2016, pp. 654–661. Available: https://pinjiahe.github.io/files/pdf/research/DSN16.pdf

[9] Valerio Persico, et al., "Measuring network throughput in the cloud: The case of Amazon EC2," Computer Networks, vol. 93, pp. 408–422, Dec. 2015. Available: http://wpage.unina.it/valerio.persico/pubs/tput_cloud_AWS_comnet.pdf

[10] Seyed Ali Mirheidari, et al., "Alert correlation algorithms: A survey and taxonomy," in Proc. Int. Conf. Cyberspace Safety and Security (CSS), Zhangjiajie, China, 2013, pp. 183–197. Available: https://arxiv.org/pdf/1811.00921

[11] Justin Zobel and Alistair Moffat, "Inverted files for text search engines," ACM Computing Surveys, vol. 38, no. 2, pp. 6–es, Jul. 2006. Available:

https://dmice.ohsu.edu/bedricks/courses/cs506-problem-solving-with-large-clusters/articles/week1/zobel_invertedindex.pdf

[12] Matin Kleppmann, “Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems,” Sebastopol, CA: O'Reilly Media, 2017. Available: https://unidel.edu.ng/focelibrary/books/Designing%20Data-Intensive%20Applications%20The%20Big%20Ideas%20Behind%20Reliable,%20Scalable,%20and%20Maintainable%20Systems%20by%20Martin%20Kleppmann%20(z-lib.org).pdf

[13] Royal Borough of Kingston upon Thames, "Information Security and Governance Policy and Framework," Information Systems Frontiers, vol. 21, no. 4, pp. 935–949, Aug. 2019. Available: https://www.kingston.gov.uk/sites/default/files/2025-05/Information_Security_and_Governance_Policy_and_Framework___RBK__Approved_.pdf

[14] Wei Xu, et al., "Detecting large-scale system problems by mining console logs," in Proc. 22nd ACM Symp. Operating Systems Principles (SOSP), Big Sky, MT, 2009, pp. 117–132. Available: https://www.sigops.org/s/conferences/sosp/2009/papers/xu-sosp09.pdf

[15] Min Du, et al., "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in Proc. 2017 ACM SIGSAC Conf. Computer and Communications Security (CCS), Dallas, TX, 2017, pp. 1285–1298. Available: https://users.cs.utah.edu/~lifeifei/papers/deeplog.pdf