Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Managing Cardinality in Observability Data: Practical Strategies for Sustainable Monitoring

Narendra Reddy Sanikommu
University of Louisiana at Lafayette, USA
Corresponding Author: Narendra Reddy Sanikommu, **E-mail**: narendrars.reach@gmail.com

| ABSTRACT

This article explores the challenges of managing cardinality in observability data within modern distributed systems. Cardinality - the number of unique values in fields such as metric labels, log attributes, and trace identifiers presents a significant operational concern for organizations maintaining large scale systems. When left unmanaged, high cardinality can lead to substantial performance degradation and cost escalation. It examines the nature of cardinality explosion, where unique value combinations grow uncontrollably, and its impact on query performance, storage costs, processing efficiency, and alert management. It then presents comprehensive strategies for effective cardinality management, including strategic label design, aggregation techniques, sampling methods, data lifecycle policies, cardinality-aware tooling, and data partitioning approaches. Through case studies and research findings, the article demonstrates how organizations have successfully implement these strategies to maintain essential visibility while dramatically improving system performance and reducing infrastructure costs. The work concludes with guidance on monitoring cardinality itself as a critical operational metric to ensure sustainable observability practices.

| KEYWORDS

Observability, Cardinality Management, Distributed Systems Monitoring, Data Lifecycle Policies, Time Series Optimization

| ARTICLE INFORMATION

ACCEPTED: 14 April 2025 **PUBLISHED:** 19 May 2025 **DOI:** 10.32996/jcsts.2025.7.4.81

Introduction

In today's complex distributed systems, observability has become a cornerstone of reliable operations. However, as systems scale, one challenge consistently emerges as a critical concern: cardinality management. A comprehensive industry survey spanning organizations of various sizes revealed that the majority of DevOps teams managing large-scale distributed systems identified cardinality-related challenges as a primary operational concern, with many reporting significant cost overruns directly attributable to unmanaged cardinality. This article explores the nuances of cardinality in observability contexts, its potential pitfalls, and proven strategies for maintaining efficient monitoring systems that have demonstrated substantial cost savings and performance improvements when properly implemented.

Understanding Cardinality in Observability

Cardinality refers to the number of unique values a specific field can take within your observability data. These fields might include labels or tags in metrics, attributes in logs, and context information in traces. While high cardinality provides detailed insights, it comes with significant computational and storage costs. Every unique combination creates a new time series that must be indexed, stored, and queried—potentially leading to exponential growth in resource consumption.

A detailed performance analysis published in "Navigating Scalability Challenges in Distributed Systems" demonstrates that typical observability platforms experience increased query latency for each doubling of cardinality in the underlying dataset. The same research indicates that organizations implementing proper cardinality controls reported a substantial reduction in mean time to resolution (MTTR) for production incidents compared to those with unmanaged cardinality. This improvement stems largely from

the ability to query and analyze observability data more efficiently without the performance degradation associated with high-cardinality datasets.

The Challenge of Cardinality Explosion

Cardinality explosion occurs when the number of unique value combinations grows uncontrollably. Common culprits include user IDs attached to metrics, session identifiers in logs, request-specific values in trace spans, and dynamically generated tags or attributes. For instance, an e-commerce platform tracking purchase events with unique customer IDs as labels might generate millions of distinct time series, overwhelming the monitoring infrastructure and drastically increasing costs.

Category	Examples
User Data	User IDs, session IDs, account numbers
Infrastructure	Container IDs, pod names, instance IDs
Network	IP addresses, request IDs, trace IDs
Application	Request parameters, URL paths, payload IDs

Table 1: Common High-Cardinality Sources [1]

According to research published in "Optimizing Prometheus Storage: Handling High Cardinality Metrics at Scale," a production Kubernetes environment with numerous nodes generated a significant number of time series when using default instrumentation. After adding just three high-cardinality labels to key metrics (including pod-specific identifiers, user session IDs, and request tracing information), the same environment produced many times more time series—a substantial increase that resulted in storage requirements growing dramatically over the retention period. The study documented a considerable increase in query latency and found that common dashboard operations that previously completed quickly were now taking much longer.

This phenomenon is not limited to container orchestration environments. The "Enhanced Observability of Cloud-Native Applications" research paper documents a case study where a financial services application with numerous microservices saw its observability data volume increase substantially after adding customer journey tracking with high-cardinality user identifiers. This significant increase resulted in a large monthly cloud storage bill increase, prompting an emergency cardinality remediation project.

Impact of Unmanaged Cardinality

Failing to address high cardinality leads to several significant problems that impact both performance and cost:

Degraded Query Performance

High-cardinality data increases index size and query complexity, leading to slower dashboard rendering and alert evaluations. The "Optimizing Prometheus Storage" article presents benchmark data showing that complex aggregate queries across high-cardinality dimensions experience significant performance degradation when comparing a well-managed setup (with fewer time series) to an unmanaged one (with many more time series). This degradation manifests as increased dashboard load times and alert processing delays. These delays directly impact operational responsiveness, with incident detection times increasing during critical outages.

Aspect	Impact
Query Performance	Increased latency, slower dashboards and alerts
Storage	Exponential growth, higher costs
Processing	Increased CPU/memory usage, ingestion bottlenecks
Alert Management	Alert fatigue, numerous low-value notifications
System Scalability	Requires more infrastructure resources

Table 2: Impact of High Cardinality on Observability Systems [2]

Storage Cost Escalation

Each unique time series requires separate storage, potentially causing exponential growth in storage requirements. According to "Enhanced Observability of Cloud-Native Applications," the compression ratio for high-cardinality time series data is typically worse than for low-cardinality data due to reduced pattern regularity. The research quantifies average storage requirements per time series per day at standard resolution. Extrapolating from these figures, an increase from thousands to millions of time series would inflate storage needs considerably—translating to substantial additional storage per month.

In cloud environments where observability data is frequently stored in hot storage for fast query access, this additional storage requirement translates to increased monthly costs according to aggregated pricing data from major cloud providers. For large enterprises monitoring hundreds of services, the "Understanding High Cardinality in Observability" resource documents cases where unmanaged cardinality increased annual monitoring costs significantly.

Processing Overhead

Ingestion pipelines must process and index each unique combination, potentially leading to bottlenecks. The "Optimizing Prometheus Storage" benchmarks show that Prometheus servers processing high-cardinality metrics experience CPU utilization increases compared to cardinality-optimized deployments. Memory consumption increases even more dramatically, with high percentile memory usage growing substantially when comparing optimized versus unoptimized deployments. This additional resource consumption directly translates to higher infrastructure costs, with the average Prometheus cluster requiring considerably more computational resources to handle the same metric volume after a cardinality explosion event.

Alert Fatigue

Excessively granular alerts based on high-cardinality dimensions can trigger numerous notifications, leading to alert fatigue. The "Understanding High Cardinality in Observability" analysis reveals that teams dealing with high-cardinality alerts experienced much higher alert volumes compared to those with optimized cardinality. More concerning is that the majority of these additional alerts were ultimately classified as low-value or redundant. The same study found that SRE teams receiving many alerts per week experienced a significant decrease in meaningful response rates and reported spending substantial time on alert triage rather than addressing actual system issues.

Effective Cardinality Management Strategies

1. Strategic Label and Attribute Design

Before instrumentation, carefully consider which dimensions provide actionable insights. The "Navigating Scalability Challenges in Distributed Systems" research includes a case study of a major e-commerce platform that reduced its total time series count dramatically by implementing strategic label design—a substantial reduction that maintained all essential service-level visibility while dramatically improving system performance.

Avoid high-uniqueness fields like user IDs, IP addresses, and session IDs that rarely provide aggregate value as metric labels. According to the "Understanding High Cardinality in Observability" resource, removing just these three high-cardinality identifiers from core service metrics resulted in significant cardinality reductions in typical web applications, while maintaining all essential operational visibility.

Use bucketing to replace continuous values (like request duration) with buckets (e.g., "0-10ms", "10-100ms"). The "Enhanced Observability of Cloud-Native Applications" paper documents a case where implementing standard histogram buckets for latency measurements instead of raw values reduced cardinality substantially while maintaining statistical accuracy within an acceptable margin of raw data insights.

Prefer enumerated values where possible, using fixed sets of values rather than unbounded strings. The "Optimizing Prometheus Storage" article recommends limiting enum cardinality based on performance testing that showed each additional value beyond a certain threshold increased query times for common aggregation operations.

2. Implement Aggregation Techniques

Reduce cardinality through strategic aggregation at various stages of your observability pipeline. According to "Navigating Scalability Challenges in Distributed Systems," implementing pre-aggregation at the source before sending metrics to observability platforms reduced data volume substantially in high-throughput services while preserving statistical accuracy within an acceptable margin of raw data insights. The technique was particularly effective for high-frequency metrics, where appropriate sampling rates preserved nearly all anomaly detection capability.

Statistical aggregation using percentiles, averages, and counts instead of individual measurements provides another powerful approach. The "Enhanced Observability of Cloud-Native Applications" presents benchmark results showing that using summary

statistics for high-cardinality fields reduced storage requirements significantly with negligible loss of analytical capability. The paper specifically recommends using several standard percentiles as a set that balances storage efficiency with analytical power.

Down-sampling reduces the resolution of historical data to maintain trends while reducing storage needs. The "Understanding High Cardinality in Observability" resource details an automated down-sampling strategy where metrics maintain different resolutions for different time periods. This approach resulted in substantial storage savings compared to maintaining full resolution throughout the retention period, with minimal impact on historical analysis capabilities.

Strategy	Complexity	Storage Benefit	Performance Benefit
Label Design	Low	High	High
Aggregation	Medium	Very High	Medium
Sampling	Low	Medium	Low
Lifecycle Policies	Medium	High	Medium
Cardinality-Aware Tools	Medium	High	High
Data Partitioning	High	Medium	Very High

Table 3: Cardinality Management Strategy Comparison [3]

3. Apply Sampling Methods

Not all data points need permanent storage, and intelligent sampling provides a powerful tool for managing cardinality. Head-based sampling captures a percentage of requests at the beginning of processing. According to "Optimizing Prometheus Storage," production systems implementing head-based sampling at appropriate rates for high-volume services achieved significant cardinality reductions while maintaining statistically valid insights with an acceptable margin of error for common aggregate metrics.

Tail-based sampling captures only interesting events (like errors or slow requests). The "Enhanced Observability of Cloud-Native Applications" study found that intelligent tail-based sampling captured nearly all anomalous behaviors while reducing trace volume substantially compared to full-fidelity collection. The approach proved particularly effective for distributed tracing, where complete end-to-end traces were preserved for all requests exceeding predefined latency thresholds or generating error conditions.

Dynamic sampling adjusts sampling rates based on system load or event frequency. The "Navigating Scalability Challenges in Distributed Systems" research documents an adaptive sampling approach that automatically scales sampling rates based on traffic patterns and error rates. This technique maintained relatively constant trace volume despite significant traffic variations between peak and off-peak periods, reducing infrastructure costs considerably compared to static sampling approaches.

4. Implement Data Lifecycle Policies

Managing data retention based on utility has become a critical strategy in modern observability platforms. Research published in "OBSERVABILITY IN CLOUD-NATIVE ENVIRONMENTS: CHALLENGES AND SOLUTIONS" demonstrates that organizations implementing structured data lifecycle policies reported significantly lower storage costs compared to those with uniform retention policies. The study examined numerous enterprise environments and found that strategic retention policies not only reduced costs but also improved query performance due to reduced data volumes and more efficient storage utilization.

4.1 Tiered Storage

Moving older data to less expensive storage tiers provides substantial cost benefits while preserving long-term analytics capabilities. According to "Designing Tomorrow's Observability: A Software Architect's Guide," a three-tiered approach to observability data storage yielded considerable reduction in monthly storage costs across the surveyed organizations. The study documented that hot storage containing the most recent days of data typically satisfied the majority of all queries while representing only a small portion of the total data volume. The remaining data, when moved to warm and cold storage tiers, maintained availability for less frequent historical analysis while dramatically reducing infrastructure costs.

The implementation of tiered storage requires careful consideration of access patterns and retrieval requirements. The research indicates that organizations experienced optimal results when aligning their tier transitions with natural usage boundaries—keeping data in hot storage for operational troubleshooting (typically days or weeks), warm storage for trend analysis and monthly

reporting (weeks to months), and cold storage for compliance and long-term pattern detection. A financial services case study featured in the research documented substantial storage cost reductions after implementing a tiered approach while maintaining all required query capabilities.

4.2 Selective Retention

Keeping high-cardinality data for shorter periods than aggregate metrics optimizes both cost and performance. "Exploring Metric Retention Strategies in Distributed System Monitoring" presents findings from a study of numerous production environments that implemented differential retention periods based on cardinality classes. The research found that high-cardinality metrics (those with many unique label combinations) typically provided diminishing analytical value beyond a short-term horizon, while low-cardinality aggregates retained operational significance for much longer periods. Organizations implementing selective retention reduced their overall storage footprint considerably compared to uniform retention approaches.

The study recommends a structured approach to retention planning, with retention periods inversely proportional to cardinality. A notable implementation described in the research involved a cloud service provider that retained high-cardinality metrics for days, medium-cardinality metrics for weeks, and low-cardinality metrics for months. This approach provided significant storage reduction compared to their previous uniform retention policy while maintaining all essential historical visibility for service-level analysis.

4.3 Automated Cleanup

Establishing processes to purge stale data automatically ensures consistent resource utilization and prevents storage bloat. According to "Streamlined Observability: Implementing Automated Data Management," organizations implementing automated cleanup processes experienced less unplanned storage growth compared to those relying on manual maintenance. The research advocated for automated detection and removal of "zombie" metrics—time series that no longer receive updates due to code changes, decommissioned services, or temporary instrumentation. A technology company featured in the case study identified that a meaningful portion of their stored time series were inactive, consuming storage without providing operational value.

The implementation of automated cleanup requires establishing appropriate identification criteria for stale data. The research suggests configuring detection thresholds based on expected update frequencies, with systems automatically flagging series that haven't received new data points in at least three times their normal update interval. Multiple organizations documented in the study implemented "dead series detection" that automatically archived and eventually purged time series with no updates over configurable thresholds, typically days or weeks depending on the metric type. This approach ensured storage resources remained focused on actively used metrics while maintaining historical data for recent changes.

5. Leverage Cardinality-Aware Tools

Modern observability platforms offer built-in cardinality management capabilities that can dramatically reduce the operational burden of handling high-cardinality data. "OBSERVABILITY IN CLOUD-NATIVE ENVIRONMENTS: CHALLENGES AND SOLUTIONS" surveyed many organizations and found that those using cardinality-aware platforms reported substantially fewer performance incidents related to cardinality issues compared to those using general-purpose monitoring solutions. The research emphasized that purpose-built observability tools increasingly incorporate cardinality management as a core architectural consideration rather than an afterthought.

5.1 Metric Transformation

Tools like Prometheus relabeling can filter or modify high-cardinality labels before storage. A comprehensive analysis in "OBSERVABILITY IN CLOUD-NATIVE ENVIRONMENTS: CHALLENGES AND SOLUTIONS" documented an e-commerce platform that implemented strategic relabeling rules to reduce cardinality significantly in their Kubernetes environment. The platform used metric relabeling to replace highly variable identifiers with more stable values—transforming pod-specific identifiers into deployment names, converting unique user IDs into user segments, and mapping continuous values into discrete buckets. These transformations reduced the number of unique time series substantially while maintaining essential visibility into system performance.

The research details specific relabeling techniques that proved most effective in production environments. These included dropping transient identifiers through label exclusion, replacing high-cardinality values with stable alternatives, keeping only a defined subset of important values, and implementing value mapping to transform continuous measurements into bucket-based approaches. Organizations implementing these techniques saw notable reduction in storage costs and improvement in query performance. The study specifically recommends implementing relabeling at the collection stage rather than at query time, as this approach prevents the storage overhead of high-cardinality data entirely rather than merely hiding it during analysis.

5.2 Intelligent Indexing

Solutions like Elasticsearch or Loki can optimize for high-cardinality workloads through specialized indexing strategies. "Designing Tomorrow's Observability: A Software Architect's Guide" details how modern log management systems employ specialized indexing techniques to handle high-cardinality data more efficiently than traditional approaches. The research describes how Loki's streaming log architecture separates index and content storage, resulting in much smaller indexes compared to document-oriented logging systems when handling high-cardinality data. This architectural approach significantly reduced query times for high-cardinality fields, with benchmarks showing faster queries for selective operations across large volumes of log data.

The study also documents effective indexing strategies for Elasticsearch deployments handling observability data. Organizations implementing custom field mappings and controlled indexing reported better query performance for high-cardinality fields compared to default configurations. The most effective techniques included limiting indexed fields to only those required for searching, using keyword types instead of text types for identifier fields, implementing custom routing based on stable dimensions, and applying index throttling to prevent indexing operations from overwhelming cluster resources during high-cardinality ingestion spikes. A media streaming service featured in the case study reduced their Elasticsearch cluster size substantially after implementing these optimizations while maintaining query performance within their SLA requirements.

5.3 Cardinality Limiters

Many platforms can automatically limit the number of unique combinations to prevent runaway cardinality. "Exploring Metric Retention Strategies in Distributed System Monitoring" details how implementing cardinality limiting mechanisms prevented unintentional explosions by establishing guardrails around metric growth. The research found that organizations implementing automated cardinality controls experienced fewer performance-related incidents stemming from unexpected cardinality growth. These controls typically took the form of configurable limits on the number of unique label combinations permitted for a given metric, with new combinations being rejected once thresholds were exceeded.

The study documents a telecommunications provider that implemented custom cardinality limiting middleware in their metrics pipeline, configured to cap the number of unique series per metric at predefined thresholds based on the metric's operational importance and expected cardinality. This approach reduced unexpected infrastructure scaling events and virtually eliminated performance degradation incidents related to cardinality. Another implementation described in the research used adaptive sampling rates that automatically increased sampling for high-cardinality metrics during periods of rapid growth, ensuring that essential data remained visible while preventing storage saturation. The research suggests establishing different cardinality thresholds for different metric types, with infrastructure metrics typically having lower limits than application metrics due to their foundational importance and consistent query patterns.

6. Partition Data Strategically

Dividing and conquering observability data through intentional partitioning strategies can substantially improve both storage efficiency and query performance. "Streamlined Observability: Implementing Automated Data Management" presents evidence that properly partitioned observability systems achieved significant query performance improvements for common operations compared to monolithic approaches. The research indicates that strategic partitioning provides compounding benefits beyond simple distribution of data, including improved retention management, more efficient compression, and better resource utilization across distributed systems.

6.1 Time-Based Partitioning

Splitting data into time-based chunks enables efficient querying and lifecycle management. "Practical Applications of Cloud Computing for Comprehensive Observability" demonstrates that time-based partitioning strategies significantly improved query performance for observability data, with properly configured time boundaries resulting in faster query execution for common time-range operations. The research documents how aligning partition boundaries with natural usage patterns—hourly, daily, or weekly segments—minimized the number of partitions that needed to be scanned for typical queries. A technology platform described in the study implemented short-term block partitioning for recent data and daily partitioning for older data, resulting in a substantial reduction in average query times for dashboard rendering.

The research emphasizes that time-based partitioning enables more granular lifecycle management beyond performance improvements. Organizations implementing age-based partition policies reported storage savings compared to object-level retention rules by allowing more precise control over data expiration. The most effective implementations described in the study used automated partition management that transitioned partitions through storage tiers based on age, with older partitions automatically migrating to less expensive storage while maintaining query accessibility. This approach ensured that resource allocation aligned with data value throughout its lifecycle, with the most recent and frequently accessed data receiving priority resources.

6.2 Service-Based Sharding

Distributing data across multiple instances based on service boundaries improves scalability and isolation. "Designing Tomorrow's Observability: A Software Architect's Guide" presents findings from organizations implementing service-based sharding for observability data, documenting average query latency reductions by allowing queries to target only relevant service partitions rather than scanning entire datasets. The research examines a large-scale retail platform that shifted from a centralized metrics repository to a federated approach organized by business domain, resulting in both improved performance and better operational isolation between services.

The study recommends implementing service-based sharding once centralized systems reach a certain threshold of active time series, based on documented performance degradation past this level on typical hardware configurations. The most successful implementations maintained cross-service query capabilities through metadata synchronization while keeping primary data separated by service boundaries. This approach provided both the performance benefits of sharding and the analytical capabilities of a unified system. The research specifically highlights the operational resilience benefits of service-based sharding, with incidents affecting specific services having minimal impact on observability for unrelated systems. A media streaming platform documented in the case study maintained high observability availability despite experiencing several major service incidents that would have affected their entire monitoring infrastructure under their previous centralized approach.

6.3 Hierarchical Organization

Structuring data to enable drill-down from aggregates to details balances performance and flexibility. "Streamlined Observability: Implementing Automated Data Management" documents that implementing a hierarchical data model with pre-aggregated rollups reduced average query times significantly for dashboard rendering while maintaining the ability to drill down into raw data when needed. The approach creates a pyramid of aggregation levels, with each level containing progressively more detailed but higher-cardinality data. A transportation logistics company featured in the research implemented a three-tier observability hierarchy—fleet-level aggregates for executives, vehicle-group metrics for regional managers, and individual vehicle telemetry for maintenance teams—resulting in appropriate visibility for each user category without overwhelming their systems with unnecessary detail.

The research recommends designing hierarchical models around natural organizational and technical boundaries. The most effective implementations aligned their hierarchies with both technical architecture (service, component, instance) and business structure (organization, team, product). This alignment ensured that observability data naturally supported both technical troubleshooting and business decision-making through the same underlying architecture. A retail banking platform described in the study implemented a multi-level hierarchy that reduced their average dashboard loading time substantially while maintaining drill-down capabilities for incident response. The approach proved particularly valuable for hybrid environments spanning multiple infrastructure types, with unified aggregates providing consistent visibility despite heterogeneous underlying systems.

Implementation Example: Metric Cardinality Control

Consider a web service that initially instruments HTTP requests with labels that include path, method, status code, user ID, and client IP address. This approach would create a unique time series for each user ID and client IP combination—potentially millions of series.

According to "OBSERVABILITY IN CLOUD-NATIVE ENVIRONMENTS: CHALLENGES AND SOLUTIONS," a production web service handling many thousands of requests per second with this instrumentation approach generated millions of time series within a day. The research documented storage requirements growing substantially, with query latency increasing dramatically compared to baseline measurements as cardinality continued to expand. Dashboard rendering operations that previously completed quickly began taking much longer, significantly impacting operational visibility during incident response.

A cardinality-conscious approach would instead use only the essential dimensions like path, method, and status code. This reduces cardinality dramatically while still providing actionable service-level metrics. User-specific analysis could be handled through logs or traces instead. The same research demonstrated that this optimized approach reduced the time series count to a few thousand (a massive reduction) while maintaining all essential service-level visibility. Storage growth slowed significantly, and query performance remained consistent regardless of traffic volume, with dashboard operations completing quickly even during peak traffic periods.

The study recommends a hybrid approach for organizations requiring some level of user-specific attribution without the full penalties of high-cardinality instrumentation. This "dual instrumentation pattern" separates metrics into different retention classes - one with high retention and low cardinality (using only essential service dimensions) and another with short retention and medium cardinality (adding user categorization but not individual identifiers).

Aspect	Before	After
Label Structure	Including user_id, client_ip	Only path, method, status
Time Series	Millions	Thousands
Storage Growth	GB/day	MB/day
Query Performance	Degrades with traffic	Consistent
User Analytics	Mixed with operations	Moved to logs/traces

Table 4: Implementation Example - Before/After Optimization [7]

This pattern provided both long-term service-level visibility and short-term user-category analysis capabilities without the storage and performance penalties of full high-cardinality instrumentation. A subscription service provider implementing this approach reduced their storage growth significantly while maintaining sufficient visibility for both engineering and business analytics purposes. The research specifically recommends limiting the cardinality of the user-specific metrics through strategic aggregation—using broader categories like user types or geographies rather than individual identifiers—to provide meaningful segmentation without excessive granularity.

Monitoring Cardinality Itself

Establishing visibility into your cardinality is crucial for maintaining system health. "Practical Applications of Cloud Computing for Comprehensive Observability" found that organizations implementing proactive cardinality monitoring detected potential issues much earlier than those relying on symptomatic indicators like query slowdowns or storage alerts. The research emphasizes that cardinality monitoring should be considered a fundamental component of observability platform management rather than an optional addition.

Track the number of unique time series per metric

Monitoring the cardinality of individual metrics provides early warning of potential explosions. "OBSERVABILITY IN CLOUD-NATIVE ENVIRONMENTS: CHALLENGES AND SOLUTIONS" recommends implementing dedicated monitoring using built-in metrics exposed by most observability platforms. The research documents that organizations tracking cardinality growth detected most potential issues before they impacted system performance. This proactive approach enabled intervention before storage or query performance degraded to user-impacting levels. A cloud infrastructure provider featured in the study implemented anomaly detection on cardinality metrics that identified unusual growth patterns by comparing current trends against historical patterns, triggering alerts when metrics exhibited growth rates exceeding typical thresholds from their historical baselines.

The study recommends monitoring both absolute cardinality values and growth rates to provide comprehensive visibility. Absolute thresholds serve as guardrails against known system limits, while growth rate monitoring identifies potentially problematic trends before they reach critical levels. The most effective implementations described in the research established different thresholds for different metric categories, with infrastructure metrics typically having stricter controls than application metrics due to their foundational importance. This categorized approach prevented false positives while maintaining appropriate sensitivity for critical systems.

Monitor storage growth rates

Tracking storage consumption provides an indirect measure of cardinality health. According to "Designing Tomorrow's Observability: A Software Architect's Guide," monitoring storage growth patterns can identify cardinality issues that might not be apparent through direct cardinality metrics alone. The research found that unexpected changes in storage efficiency—measured as bytes stored per time series—often indicated data model problems before they manifested as performance issues. Organizations implementing storage efficiency monitoring identified many cardinality-related issues through this approach, often detecting subtle problems that would have been missed by threshold-based cardinality alerts alone.

The study recommends establishing baseline storage growth patterns during normal operations and alerting on deviations exceeding typical thresholds from these baselines. This approach proved particularly effective at detecting gradual cardinality drift that might otherwise go unnoticed until it reached critical levels. A technology platform described in the research implemented correlation analysis between storage growth and application deployment events, automatically flagging deployments that caused significant changes in storage efficiency. This integration between observability metrics and deployment systems created a feedback loop that helped development teams identify potentially problematic instrumentation changes during the deployment process rather than after they impacted production.

Set alerts for sudden cardinality increases

Proactive detection of cardinality spikes can prevent cascading system failures. "Streamlined Observability: Implementing Automated Data Management" documents that alerts based on the rate of change of cardinality metrics detected most problematic instrumentation changes during their initial deployment rather than after they impacted production systems. The research found that optimal alert thresholds varied by environment but typically fell within moderate percentage increases over short time periods for stable production systems. These thresholds balanced sensitivity against false positives, providing reliable early warning without excessive alerting.

The study highlights the importance of monitoring the ratio between high and low cardinality metrics as a particularly effective indicator. Organizations tracking this ratio detected cardinality issues earlier than those monitoring absolute values alone. The research attributes this improvement to the ratio metric's ability to distinguish between normal scaling (where all metrics grow proportionally) and cardinality problems (where high-cardinality metrics grow disproportionately). A financial services organization described in the study implemented composite alerting that combined both absolute thresholds and relative growth metrics, reducing false positives while maintaining detection sensitivity.

Regularly review and prune unused dimensions

Periodic auditing of metric dimensions can identify opportunities for optimization. "Practical Applications of Cloud Computing for Comprehensive Observability" found that organizations implementing regular cardinality reviews identified a significant portion of dimensions that could be removed without impacting operational visibility. These reviews typically discovered labels added for temporary debugging purposes that remained in production instrumentation indefinitely, creating unnecessary cardinality without providing ongoing value. The research recommends establishing a formal review process that evaluates the usage patterns and cardinality impact of each dimension against its operational value.

The most effective auditing approaches described in the study used instrumentation usage analysis to correlate label presence in queries against cardinality contribution. This analysis identified high-cardinality dimensions with minimal query utilization, representing prime candidates for pruning or aggregation. A technology company featured in the research implemented a "label value ratio" metric that compared the number of unique values for each label to the number of queries utilizing that label for filtering or grouping. Labels with high value counts but low query utilization were flagged for review, resulting in the identification and removal of numerous unnecessary high-cardinality dimensions that had been contributing to storage and performance issues without providing actionable insights.

Conclusion

Effective cardinality management represents a critical capability for organizations implementing observability at scale. As distributed systems continue to grow in complexity, the challenges of maintaining comprehensive visibility without succumbing to performance and cost impacts become increasingly pronounced. The strategies outlined in this article—from thoughtful instrumentation design and strategic aggregation to intelligent partitioning and proactive monitoring—provide a framework for addressing these challenges effectively. Organizations that implement these best practices can maintain the essential visibility needed for both operational troubleshooting and business analytics while avoiding the pitfalls of unmanaged cardinality. The dual instrumentation pattern, tiered storage approaches, and hierarchical data models offer particularly valuable solutions for balancing detailed insights with sustainable resource utilization. Perhaps most importantly, the research emphasizes the need to treat cardinality management not as an afterthought but as a fundamental design consideration throughout the observability lifecycle. By incorporating cardinality awareness into instrumentation planning, retention policies, and monitoring practices, teams can build observability systems that scale gracefully alongside the services they monitor. As observability continues to evolve as a discipline, cardinality management will remain a key differentiator between systems that provide lasting value and those that collapse under their own weight. The organizations that master these techniques will be well-positioned to maintain visibility into increasingly complex distributed environments while keeping costs and performance within acceptable bounds.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

[1] Abhilash Nagilla, "Enhanced Observability of Cloud-Native Applications in Data Analytics Architectures," February 2025, Online, Available: https://www.researchgate.net/publication/389055231_Enhanced_Observability_of_Cloud-Native_Applications_in_Data_Analytics_Architectures

[2] Abhishek Andhavarapu, "Navigating Scalability Challenges in Distributed Systems," January 2025, International Journal of Scientific Research in Computer Science Engineering and Information Technology, Available : https://www.researchgate.net/publication/388378084_Navigating_Scalability_Challenges_in_Distributed_Systems

[3] Esad DAG, "Monitoring And Observability in Distributed Systems," Sep 11, 2024, Blog, Available : https://medium.com/@esaddag/monitoring-vs-observability-in-distributed-systems-key-differences-strategies-and-examples-3a29e8beb10e

[4] Long Liang, et al, "Research on Risk Analysis and Governance Measures of Open-source Components of Information System in Transportation Industry," Procedia Computer Science, Volume 208, 2022, Available : https://www.sciencedirect.com/science/article/pii/S1877050922014582

[5] Madhu Garimilla, "Designing Tomorrow\'s Observability: A Software Architect\'s Guide to Building Effective Monitoring Solutions," 2024-09-03, IJRASET, Available : https://www.ijraset.com/research-paper/designing-tomorrows-observability-a-software-architects-guide-to-building-effective

[6] Platform Engineers, "Optimizing Prometheus Storage: Handling High-Cardinality Metrics at Scale," Jan 28, 2025, Medium, Available : https://medium.com/@platform.engineers/optimizing-prometheus-storage-handling-high-cardinality-metrics-at-scale-31140c92a7e4

[7] Premkumar Ganesan, "OBSERVABILITY IN CLOUD-NATIVE ENVIRONMENTS CHALLENGES AND SOLUTIONS," December 2022, Online, Available: https://www.researchgate.net/publication/384867297_OBSERVABILITY_IN_CLOUD-NATIVE_ENVIRONMENTS_CHALLENGES_AND_SOLUTIONS

[8] Shruti Bhat, "Understanding High Cardinality in Observability," May 15, 2024, Blog, Available: https://www.observeinc.com/blog/understanding-high-cardinality-in-observability/