The Eastasouth Journal of Information System and Computer Science
Vol. 1, No. 02, December, pp. 325-337

Observability for Java Microservices: Metrics, Tracing, and Logging in Cloud-Native Systems

Tathagata Roy
Independent Researcher, Colorado, USA

Article Info

Article history:

Received December, 2023
Revised December, 2023
Accepted December, 2023

Keywords:

Broker Telemetry;
Cloud Native Telemetry;
Consumer Lag;
Container Resource Limits;
Datadog;
Distributed Tracing;
Dynatrace;
Elastic Stack;
Grafana;
JVM Metrics;
Kafka Monitoring;
Kubernetes Monitoring;
Metrics Instrumentation;
Micrometer;
MTTR Reduction;
New Relic;
Observability;
OpenTelemetry;
Performance Diagnostics;
Producer Metrics;
Prometheus;
Splunk;
Spring Boot microservices;
SRE Practices;
Structured Logging

ABSTRACT

Distributed Java microservices require comprehensive observability to maintain reliability, diagnose performance degradation, and support operational continuity across diverse cloud native and hybrid environments. This paper presents a unified framework for monitoring Spring Boot based microservices through metrics instrumentation, distributed tracing, structured log aggregation, and infrastructure level telemetry. The study identifies essential JVM, application, and platform indicators required for effective performance analysis, including garbage collection behavior, thread pool utilization, connection pool saturation, request latency, and error propagation patterns. Techniques for instrumenting Spring Boot applications using Micrometer, OpenTelemetry, and structured logging are examined, along with strategies for exporting telemetry to platforms such as Dynatrace, Prometheus, Datadog, New Relic, Splunk, and Elastic Stack. The framework extends to asynchronous processing pipelines by outlining critical Kafka producer, consumer, broker, and topic level metrics such as consumer lag, partition health, request latency, and under replicated partitions that enable early detection of backpressure and message flow disruptions. Infrastructure level signals, including container resource limits, node pressure, CPU throttling, pod restarts, and autoscaling activity, are incorporated to support SRE driven diagnostics and correlation. The paper defines recommended dashboards, alert thresholds, and telemetry correlation models that reduce mean time to resolution and provide deep visibility into Java microservices operating at scale.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Name: Tathagata Roy
Institution: Independent Researcher, Colorado, USA
Email: tatha.roy@gmail.com

1. INTRODUCTION

The transition from monolithic systems to distributed Java microservices has reshaped the operational landscape of enterprise software [1], [2]. As organizations adopt decentralized service ownership and deploy components across cloud-native and hybrid environments, the complexity of managing service interactions increases

substantially [3], [4]. A single request may traverse numerous independently deployed services, each subject to its own resource constraints, failure modes, and scaling behavior [5], [6], [7]. These characteristics make identifying performance bottlenecks and systemic degradation significantly more challenging, particularly within transient platforms such as Kubernetes (K8s), where containers are ephemeral, and workload patterns fluctuate rapidly [3], [8], [9]. Traditional log-centric monitoring approaches can't capture the contextual and temporal relationships required to reconstruct distributed execution paths or diagnose asynchronous failures, leaving Site Reliability Engineering (SRE) teams with fragmented signals and prolonged Mean Time to Resolution (MTTR) [10], [11], [12], [13], [14].

This work addresses these challenges by defining a unified observability framework that integrates metrics, distributed tracing, and structured logging with telemetry from Apache Kafka pipelines and underlying infrastructure layers [15], [16], [17], [18], [19], [20]. The framework demonstrates how standardized instrumentation in Java 17 and Spring Boot 2.7.x can provide consistent visibility across synchronous and asynchronous workflows, enabling correlation between application behavior and platform health in real time [17], [21], [22], [23]. The study focuses on implementations using OpenTelemetry (OTel) as a vendor-neutral standard for telemetry collection and evaluates deployments across Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and on-premises virtual machine environments [15], [24], [25]. The framework establishes practical guidelines for instrumentation, metric collection, and alerting strategies that support reliable operation in high-concurrency distributed environments [26], [27], [28].

2. BACKGROUND AND ARCHITECTURAL CONTEXT

2.1 Observability in Distributed Systems

Observability extends beyond traditional monitoring by enabling an understanding of internal system behavior through externally emitted signals [3], [5], [10], [13], [19], [27]. Monitoring focuses on predefined indicators that describe whether a system operates within expected boundaries, while observability provides the ability to infer system state from telemetry that captures dynamic execution paths and emergent interactions [3], [5]. Distributed microservice architectures amplify these challenges because requests traverse numerous independently deployed services, each with its own resource profile, latency characteristics, and failure modes [1], [2], [6], [7]. Metrics supply quantitative measurements of system behavior, distributed tracing captures causal and temporal relationships across service boundaries, and structured logging records discrete events with contextual detail [14], [15], [16], [17], [19], [20]. Effective observability requires the correlation of these signals to overcome fragmentation and support coherent analysis of complex multi-service workflows [15], [16], [20].

2.2 Spring Boot Telemetry Foundations

Spring Boot provides a consistent telemetry foundation through its integration with Micrometer, which serves as a vendor-neutral facade for application metrics [17], [21]. Micrometer enables standardized instrumentation while supporting export to platforms such as Prometheus, Dynatrace, Datadog, and New Relic [19], [27], [28], [29]. Distributed tracing is supported through the OTel instrumentation model, which standardizes trace context propagation, span creation, and automatic instrumentation of common libraries [15], [16], [20], [30], [31]. Structured JavaScript Object Notation (JSON) logging further enhances diagnostic capabilities by embedding trace and span identifiers within log events, enabling correlation between logs and traces during incident analysis [10],

[14]. Together, these components establish a unified telemetry layer that supports consistent visibility across diverse deployment environments [15], [17], [21].

2.3 Runtime and Platform Considerations

The behavior of the Java Virtual Machine (JVM) changes significantly when applications run inside containerized environments [8], [9], [22], [23]. Resource limits defined by Control Groups influence heap sizing, garbage collection behavior, and thread scheduling, while CPU throttling can introduce latency spikes that affect both synchronous and asynchronous workloads [8], [9]. Autoscaling mechanisms add further variability by adjusting the number of running instances in response to workload fluctuations [8], [9], [25]. Event-driven and asynchronous processing pipelines introduce additional complexity because execution paths may span multiple threads, message queues, and Kafka topics, making correlation dependent on consistent trace context propagation [6], [15], [16], [18], [30], [31]. Instrumentation strategies vary between bytecode-level auto-instrumentation using Java agents and library-native instrumentation provided by Spring Boot Actuator [15], [21], [30], [31]. Auto-instrumentation offers broad coverage with minimal code changes, while library-native instrumentation provides finer control and lower overhead. Selecting an approach requires balancing instrumentation cost against the granularity and completeness of the collected telemetry [15], [21].

3. APPLICATION METRICS AND JVM PERFORMANCE INDICATORS

3.1 JVM-Level Metrics

Monitoring the JVM provides early visibility into resource pressure and runtime inefficiencies that influence application performance [22], [23]. Garbage collection activity is a primary indicator of memory behavior, and tracking pause times in the Garbage First Garbage Collector helps identify periods where application threads are halted [22]. As shown in Table 1, G1GC pause times exceeding 200ms at the 99th percentile indicate memory pressure requiring immediate investigation, as these pauses directly manifest as request latency spikes. Heap and non-heap memory usage reveal allocation patterns, while Metaspace growth provides insight into class metadata consumption and potential class loading leaks [22], [23]. Sustained heap utilization exceeding 85% (Table 1) triggers increased garbage collection frequency, creating a cascade effect where GC overhead compounds application latency. Thread pool utilization reflects the ability of the JVM to schedule concurrent tasks, and sustained saturation often leads to silent queuing and increased latency [8], [9], [22]. Class loading activity, safe-point statistics, and Just-In-Time (JIT) compilation signals influence both startup behavior and steady-state performance [22], [23]. These factors are particularly important in autoscaling environments, where new instances must warm up quickly [8], [9], [25]. Extended JIT warmup or heavy class loading during startup can delay readiness and cause scale-out events to lag behind demand, reducing the effectiveness of horizontal scaling [22], [23]. The JVM metrics consolidated in Table 1 provide early warning indicators of runtime degradation before application-layer symptoms manifest, enabling proactive intervention during autoscaling events.

3.2 Application-Level Metrics

Application-level metrics provide direct visibility into service responsiveness and reliability [1], [11], [12], [17], [19], [21]. Request latency and throughput quantify performance characteristics, while error rates and

exception patterns highlight functional or systemic issues [1], [11], [12]. As shown in Table 1, HTTP request latency exceeding 500ms at the 95th percentile indicates user-perceptible degradation, typically resulting from resource saturation rather than application logic defects. Resource pools are central to understanding saturation because they govern concurrency and connection management across multiple layers of the application [1], [2]. Database connectivity relies on HikariCP, while HTTP clients, embedded servers, and asynchronous task executors maintain their own connection and thread pools [1], [2], [21], [32]. Exhaustion in any of these pools leads to queuing and increased latency, often without immediate error signals [1], [2]. HikariCP active connections approaching 90% of maximum pool size (Table 1) represent a critical threshold, as connection exhaustion manifests as silent request queuing rather than explicit errors. Cache hit and miss ratios influence backend load distribution and overall responsiveness [1], [6]. Spring Boot Actuator exposes these metrics through the metrics endpoint and supports custom MeterBinders for domain-specific instrumentation [17], [21]. Monitoring thread pool execution in embedded servers such as Tomcat or Jetty, along with connection pool health across database and HTTP clients, helps identify bottlenecks that degrade system performance even when error rates remain low [1], [2], [21].

3.3 Exporting Metrics

Micrometer provides the primary mechanism for exporting application metrics by acting as a facade that standardizes measurement collection and supports multiple backend registries [17]. The Prometheus exposition format is widely used in cloud-native environments, enabling metrics scraping and integration with visualization tools such as Grafana [19], [33]. Application Performance Monitoring platforms, including Dynatrace, Datadog, and New Relic, support several ingestion paths for metrics [27], [28], [29]. Vendor-specific agents collect JVM and system-level metrics directly from the runtime and can automatically detect common frameworks without requiring code changes [14], [27], [28], [29]. Micrometer also offers native registry implementations that push application-defined metrics to these platforms through their ingestion APIs [17]. In addition to these mechanisms, many platforms integrate with OTel exporters to unify metrics with traces and logs [15], [16], [20], [30], [31]. Selecting an export strategy requires balancing operational simplicity, metric fidelity, and the need for consistent correlation across telemetry signals [15], [17], [19]. The 'Source' column in Table 1 maps each metric to its primary collection mechanism, JMX for JVM, Micrometer for application indicators, and platform-specific agents (cAdvisor, kubelet, node-exporter) for infrastructure telemetry, guiding observability platform configuration.

4. DISTRIBUTED TRACING AND CONTEXT PROPAGATION

4.1 Trace Context Propagation

Distributed tracing reconstructs the execution path of requests as they traverse microservice architectures, enabling visibility into service dependencies and latency contributors [15], [20], [30], [31]. The W3C Trace Context specification defines a standardized format for propagating identifiers such as trace identifiers and span identifiers across process boundaries, ensuring interoperability across instrumentation libraries and platforms [16]. Many systems historically relied on correlation identifiers to associate related events, but these identifiers lacked a consistent

propagation model and provided only partial visibility [20], [31]. Modern architectures require reliable propagation across HTTP requests, messaging systems, and asynchronous workflows [15], [16], [20], [30], [31]. In message-driven systems such as Kafka or RabbitMQ, producers must inject the trace context into message headers, and consumers must extract it to preserve the causal chain [6], [18], [30], [31]. Consistent propagation across thread pools, message queues, and reactive pipelines is essential for assembling a complete end-to-end trace [15], [16], [20], [30], [31].

4.2 Instrumentation Techniques

Instrumentation determines how spans are generated and how context is captured across service boundaries [15], [20]. OTel auto-instrumentation provides broad coverage by automatically creating spans for common frameworks, HTTP clients, database drivers, and messaging libraries without requiring code modification [15], [30], [31]. Spring Boot Sleuth historically provided automatic trace context propagation and span creation for Spring-based applications, enabling integration with tracing backends such as Zipkin [21], [31]. Manual instrumentation remains necessary for business-critical operations where domain-specific spans provide deeper visibility into workflow stages or performance-sensitive logic [15], [20]. Asynchronous and event-driven architectures require careful management of span context because execution may occur across detached threads or message handlers [15], [16], [20], [30], [31]. Correctly linking parent and child spans ensures that asynchronous workflows appear as unified traces rather than fragmented segments [15], [20], [30], [31].

4.3 Trace Analysis

Trace analysis provides insight into the performance characteristics of distributed systems by exposing latency contributions, dependency chains, and failure propagation patterns [15], [16], [20], [30], [31]. Visualizing the duration and ordering of spans helps identify latency hotspots and isolate the specific service or operation responsible for performance degradation [20]. Cascading failures often manifest as repeated timeouts or retries across dependent services, and traces reveal the sequence of events that lead to these conditions [11], [12], [20]. Correlating traces with logs and metrics strengthens diagnostic accuracy by aligning temporal and contextual information across telemetry pillars [10], [14], [15], [16], [20]. Sampling strategies influence the completeness and cost of trace collection [15], [20]. Head-based sampling makes decisions at the start of a trace and reduces overhead but may exclude rare failure scenarios [15], [20]. Tail-based sampling evaluates traces after completion and retains those with anomalous characteristics, improving diagnostic value at higher computational cost [20]. Adaptive sampling adjusts sampling rates dynamically based on traffic patterns or error conditions, balancing visibility with resource consumption [15], [20].

5. STRUCTURED LOGGING AND LOG AGGREGATION

5.1 Structured JSON Logging

Structured logging provides a consistent and machine-readable representation of application events, enabling reliable parsing and correlation across distributed systems [10], [14]. The adoption of JSON logging formats allows applications to emit normalized fields such as timestamps, log levels, service identifiers, and thread identifiers rather than unstructured text [10], [14]. This normalization eliminates the need for resource-intensive regular expression parsing at the ingestion layer, which is a common bottleneck in platforms such as

Splunk or Elastic Stack [13], [14]. Embedding trace identifiers and span identifiers within each log entry enables direct correlation between logs and distributed traces, allowing operators to align textual evidence with specific spans and reconstruct execution paths [15], [16], [20], [30], [31]. Structured logging, therefore, strengthens diagnostic workflows by providing high-fidelity qualitative information that complements metrics and traces [10], [13], [14].

5.2 Log Aggregation Pipelines

Centralized log aggregation relies on multi-stage pipelines that collect logs from distributed containers or virtual machines, buffer them to prevent data loss during traffic spikes, and forward them to indexing platforms for analysis [13], [14], [28]. Sidecar containers and node-level daemons are commonly used to ship logs to centralized clusters, where indexing strategies determine query performance and retention cost [8], [9]. Platforms such as Splunk, Elastic Stack, and cloud-native logging services provide search and visualization capabilities that benefit from structured log formats [13], [14], [28]. Transitioning to JSON logging improves ingestion efficiency by removing the need for pattern-based extraction and reducing parsing overhead [10], [13], [14]. Exception analysis is enhanced through the grouping of identical stack traces, enabling Site Reliability Engineering (SRE) teams to distinguish systemic defects from transient infrastructure failures and prioritize remediation efforts [11], [12], [13], [14].

6. OBSERVABILITY FOR ASYNCHRONOUS PROCESSING AND KAFKA WORKLOADS

Asynchronous communication patterns introduce unique challenges for telemetry because request-response cycles are decoupled [2], [3], [5], [6]. In systems utilizing Apache Kafka, a transaction often spans multiple services and messaging boundaries, requiring a specialized set of indicators to maintain visibility [6], [18].

6.1 Producer Metrics

Producer-side telemetry provides early indicators of throughput limitations and reliability issues in asynchronous pipelines [6], [18]. Send rate, error rate, and retry rate collectively describe the producer's ability to deliver messages under varying load conditions [18]. Request latency reflects the time required for the broker to acknowledge produce operations and is sensitive to network conditions, broker load, and storage performance [18]. As shown in Table 1, producer send latency exceeding 100ms at the 99th percentile indicates broker-side backpressure or network congestion. Buffer exhaustion occurs when the producer's internal memory buffers reach capacity, causing backpressure and increased queuing delays [6], [18]. Monitoring buffer pool wait time helps identify cases where application threads are blocked while waiting for available buffer space [18]. Client quotas introduce additional constraints that can silently degrade performance [18]. Quota throttling metrics, including produce throttle time and fetch throttle time, reveal when the broker is deliberately delaying client operations due to configured throughput limits [18]. These signals help identify misconfigured quotas or imbalanced resource allocation across tenants [18], [6].

6.2 Consumer Metrics

Consumer metrics provide visibility into the ability of downstream services to keep pace with incoming message volume [6], [18]. Consumer lag represents the difference between the latest committed offset and the head of the partition log, serving as a primary indicator of backlog accumulation [18]. Consumer lag exceeding 10,000

messages (Table 1) typically indicates insufficient processing capacity or downstream bottlenecks, though optimal thresholds vary by workload characteristics and message processing time. Fetch latency measures the time required for consumers to retrieve data from brokers and is influenced by network conditions, broker load, and partition distribution [6], [18]. Commit failures indicate instability in offset management and can lead to message duplication or processing gaps [6], [18]. Rebalance events occur when consumer group membership changes or when partition assignments shift, and frequent rebalances disrupt processing continuity [18]. Monitoring these metrics helps identify slow consumers, inefficient partition assignments, or downstream bottlenecks that impede message flow [6], [18].

6.3 Broker and Cluster Metrics

Broker-level telemetry provides insight into the health and stability of the Kafka cluster [6], [18]. Under replicated partitions indicate that replicas are not fully synchronized, increasing the risk of data loss during broker failures [18]. As shown in Table 1, any sustained non-zero value for under-replicated partitions represents a critical alert condition, as this metric directly indicates compromised fault tolerance and potential data durability violations. Changes in the in-sync replica set, including shrink and expand events, reflect replication instability and may signal network congestion or storage performance degradation [6], [18]. Disk input and output throughput, network throughput, and garbage collection behavior influence the broker's ability to sustain high message rates [6], [18], [22]. Elevated disk latency or network saturation can lead to increased request latency for both producers and consumers [6], [18]. Monitoring these metrics enables operators to detect cluster imbalance, hardware degradation, or resource contention that affects end-to-end message delivery [10], [6].

6.4 Topic-Level Metrics

Topic-level metrics provide visibility into data distribution and retention behavior across partitions [18], [6]. Partition skew occurs when message volume or processing load is unevenly distributed, leading to hotspots that degrade throughput and increase consumer lag [18]. Retention and compaction status influence storage utilization and determine how long messages remain available for consumption or replay [6], [18]. Message backlog growth reflects the rate at which messages accumulate relative to consumer processing capacity and serves as an early indicator of downstream performance issues [6], [18]. Sustained backlog growth that drives consumer lag beyond threshold values (Table 1) requires immediate intervention—either through consumer scaling, partition rebalancing, or downstream optimization—to prevent cascading delays across dependent services. These metrics help organizations maintain balanced workloads, optimize storage policies, and ensure predictable message delivery across asynchronous pipelines [6], [18].

7. INFRASTRUCTURE AND PLATFORM TELEMETRY

7.1 Workload-Level Telemetry

Workload-level telemetry provides visibility into the behavior of applications running inside containerized environments [8], [9]. CPU usage relative to configured limits indicates whether a workload is approaching throttling conditions that degrade responsiveness [8], [9]. As shown in Table 1, CPU throttling exceeding 5% of container runtime directly degrades application latency through request queueing, even when CPU utilization appears moderate from

the application's perspective. Memory usage relative to limits must be monitored to prevent out-of-memory events, which result in immediate termination of the container by the kernel [8], [9]. Node memory pressure conditions (Table 1) trigger Kubernetes pod eviction mechanisms, making this a critical platform stability indicator that precedes cascading application failures. Pod restart loops often reflect resource misconfiguration, memory leaks, or platform-induced evictions [8], [9]. Restart counts exceeding 5 per hour (Table 1) indicate CrashLoopBackOff risk, requiring root cause analysis before exponential backoff delays impact service capacity. Events such as OOM kills, CPU throttling, and triggers from the Horizontal Pod Autoscaler frequently correlate with application latency spikes or throughput degradation [8], [9], [25]. These signals, often sourced via cAdvisor within the kubelet, help operators determine whether performance issues originate from workload behavior or from resource saturation within the orchestration environment [8].

7.2 *Node and Host Telemetry*

Node and host telemetry captures the health of the compute layer supporting both containerized and non-containerized workloads [8], [9]. CPU throttling at the node level can occur when aggregate demand exceeds host capacity, even if individual workloads remain within their limits [8], [9]. Memory pressure may trigger page faulting or disk swapping, which significantly affects the performance of the JVM by increasing pause times and reducing allocation throughput [8], [9], [22]. In Kubernetes environments, the transition of a node to a MemoryPressure or DiskPressure state is a critical early warning signal that precedes pod eviction [8], [9]. Node memory pressure conditions (Table 1, Row 9) correlate with increased pod restart activity (Table 1, Row 12) and CPU throttling (Table 1, Row 8). Disk I/O wait times indicate storage contention, while network interface saturation affects communication between services [6], [8]. As shown in Table 1 (Rows 13-14), disk I/O utilization and network bandwidth exceeding 80% represent the most common infrastructure bottlenecks. Node NotReady conditions persisting beyond 5 minutes trigger automated pod rescheduling, but sustained node instability indicates underlying infrastructure issues requiring intervention. These signals apply equally to Kubernetes nodes, virtual machines, and bare-metal hosts, providing a unified view of resource contention across cloud, hybrid, and on-premise environments [24], [25], [28].

7.3 *Orchestration and Control Plane Signals*

Orchestration-level telemetry provides insight into the scheduling and management behavior of the platform [8], [9]. Pending pod queue length indicates whether the scheduler is unable to place new workloads due to insufficient resources or node availability [8], [9]. Autoscaling activity reflects the responsiveness of the cluster to changing demand and helps identify scenarios where scaling decisions lag behind workload fluctuations [8], [9], [25]. Control plane latency measures the responsiveness of core orchestration components and influences the timeliness of scheduling, configuration updates, and state reconciliation [8], [9]. Monitoring these signals enables SRE teams to distinguish between localized workload failures and broader platform instability, ensuring that infrastructure behavior is interpreted within the context of application performance and defined Service Level Objectives (SLOs) [11], [12].

Table 1. Key Performance Metrics, Alert Thresholds, and Degradation Impacts

Table I: Key Performance Metrics by Layer

Metric Category	Specific Indicator	Source	Threshold Guidance	Impact When Degraded
LAYER 1: JVM RUNTIME				
JVM Memory	Heap utilization	JMX	>85% sustained	Increased GC pause times
JVM GC	G1GC pause time	JMX	>200ms P99	Request latency spikes
LAYER 2: APPLICATION				
Application	HTTP request latency	Micrometer	>500ms P95	User-visible delays
Application	HikariCP active connections	Micrometer	>90% of max pool	Connection exhaustion
LAYER 3: MESSAGING INFRASTRUCTURE				
Kafka Producer	Send latency	JMX	>100ms P99	Message delivery delays
Kafka Consumer	Consumer lag	JMX	>10,000 messages	Processing backlog accumulation
Kafka Broker	Under-replicated partitions	JMX	>0 sustained	Data loss risk during broker failure
LAYER 4: INFRASTRUCTURE & PLATFORM				
Container	CPU throttling	cAdvisor	>5% time throttled	Application latency degradation
Node	Memory pressure	kubelet	MemoryPressure=true	Pod eviction risk
Kubernetes Pod	Readiness probe failures	kubelet	>3 consecutive failures	Pod removed from service endpoints
Kubernetes Pod	Liveness probe failures	kubelet	>1 failure	Pod restart (disrupts in-flight requests)
Kubernetes Pod	Restart count	kubelet	>5 restarts/hour	CrashLoopBackOff risk
Node Resources	Disk I/O utilization	node-exporter	>80% sustained	Database latency spikes
Node Resources	Network bandwidth saturation	node-exporter	>90% of interface limit	Packet loss, increased latency
Cluster Health	Node NotReady status	kube-state-metrics	>1 node for >5 min	Reduced capacity, pod evictions
Cloud Platform	Load balancer backend health	Azure Monitor	<2 healthy backends	Service degradation risk

8. TELEMETRY CORRELATION AND ROOT-CAUSE ANALYSIS

The effectiveness of an observability framework depends on the ability to synthesize isolated telemetry signals into a coherent diagnostic narrative [3], [5], [10], [14], [15], [16], [19], [20]. Metrics, logs, and traces each provide valuable information, but without coordinated analysis, they produce fragmented insights [5], [10], [13], [14]. Multi-layer correlation workflows are therefore essential for converting raw telemetry into actionable intelligence, particularly when diagnosing complex performance regressions in distributed Java environments [14], [15], [16], [20].

8.1 Correlating Metrics, Logs, and Traces

Correlation across telemetry pillars enables diagnostic workflows that extend beyond isolated signals [10], [14], [15], [16], [19], [20]. Metrics provide aggregate indicators of system health, while logs supply contextual detail and traces capture end-to-end execution paths [10], [14], [15], [16], [20]. Trace-driven log retrieval allows operators to align log entries with specific spans, reducing the search space during incident analysis [14], [15], [16], [20], [30], [31]. Metric-driven trace sampling complements this approach by capturing detailed traces only when anomalies such as latency spikes or elevated error rates occur [15], [19], [20]. Metric exemplars, supported by systems such as Prometheus, embed trace identifiers within individual metric samples, enabling direct transitions from aggregate trends to discrete execution paths [19]. These correlation techniques transform raw telemetry into coherent diagnostic narratives and reduce the

cognitive effort required to isolate failure domains [14], [15], [16], [20].

8.2 Detecting Early Warning Signals

Early warning signals provide predictive insight into emerging instability before user-visible degradation occurs [11], [12]. Backpressure patterns often manifest when downstream services cannot sustain incoming load, reflected in queue growth, increased retry activity, or prolonged wait times in connection pools [6], [18]. Latency propagation across service boundaries indicates that delays in one component are cascading through dependent services, revealing bottlenecks that may not be visible through isolated metrics [15], [16], [20]. Resource saturation trends, including gradual increases in CPU utilization, memory consumption, or disk latency, highlight conditions that precede performance collapse [6], [8], [9], [22]. Identifying these patterns enables organizations to intervene proactively and maintain system stability [11], [12].

8.3 Reducing Mean Time to Resolution

Reducing mean time to resolution requires telemetry systems that provide actionable context rather than fragmented alerts [11], [12]. Alert grouping consolidates related signals across application and infrastructure layers, reducing noise and preventing operators from responding to secondary symptoms [11], [12], [28]. Context-rich notifications incorporate trace identifiers, recent log exceptions, and relevant platform events to accelerate triage [10], [14], [15], [16], [20]. Automated triage patterns capture diagnostic artifacts such as thread dumps or heap profiles when a service enters a degraded state, preserving information that may otherwise be lost after a restart [22], [23], [28]. These practices streamline incident response and enable faster isolation of the underlying cause [11], [12], [28].

8.4 Regulatory Demands

Telemetry systems must comply with regulatory requirements that govern the handling of sensitive operational data [34], [35]. Avoiding the inclusion of personally identifiable information in logs is essential, and masking strategies ensure that sensitive fields are redacted or tokenized before ingestion [13], [14], [35]. Secure transport of telemetry data protects signals in transit and prevents unauthorized interception or modification [24], [25], [34], [35]. Role-based access to dashboards ensures that only authorized personnel can view operational data, reducing the risk of inadvertent disclosure [24], [25], [35]. These controls maintain compliance while preserving the diagnostic value of observability systems [13], [14], [34], [35].

9. OPERATIONAL CONSIDERATIONS AND ALERTING STRATEGIES

Production observability isn't just about collecting metrics. It requires translating the framework into practical dashboards and alert configurations that help teams respond quickly when things go wrong [11], [12], [13], [27], [33]. This section covers how to organize dashboards, implement SLO-based alerting, and build diagnostic workflows using the metrics consolidated in Table 1.

9.1 Dashboard Organization Principles

Production environments work best with a tiered dashboard organization that lets operators drill down from symptoms to causes [11], [12]. Application-layer dashboards surface golden signals (latency, throughput, and error rates) along with JVM runtime metrics like heap utilization, garbage collection pauses, and thread pool saturation [1], [17], [19], [21], [22], [23]. Messaging-layer dashboards track producer send latency, consumer lag, and broker under-replicated partitions. Infrastructure

dashboards reveal container CPU throttling, node memory pressure, and other resource constraints that eventually show up as application problems [6], [8], [9], [18]. All these metrics are defined in Table 1 with their corresponding thresholds and collection mechanisms.

This layered structure helps operators move from what users are experiencing to what's actually broken underneath. Correlation between metrics and distributed traces lets you jump directly from a latency spike to the specific requests that caused it [15], [16], [19], [20], [30], [31]. Autoscaler behavior dashboards are particularly useful because they show how scaling events affect JVM warmup times during traffic surges [8], [9], [22], [23], [25].

9.2 Alerting Philosophy and SLO Integration

Static threshold alerts generate too much noise. Better alerting strategies use SLO-based models that focus on error budget consumption across multiple time windows [11], [12]. The key is distinguishing between issues that need immediate attention (page-worthy) and problems that can wait until business hours (ticket-worthy) [11], [12]. The thresholds defined in Table 1 translate directly into alert rules with multi-window evaluation strategies. A typical pattern uses 5-minute windows for initial warnings and 15-minute sustained breaches for escalation to page-worthy severity [11], [12]. For example, HTTP request latency exceeding its P95 threshold triggers investigation when the 5-minute average crosses the line, but only escalates to a page when sustained across 15 minutes [11], [12]. Consumer lag follows similar logic, starting as a ticket-worthy alert at moderate levels but becoming critical when combined with elevated producer latency and broker instability [6], [11], [12], [18].

Multi-signal correlation reduces false positives by requiring confirmation across multiple indicators before escalating alerts [10], [14], [15], [16], [19], [20]. The metrics in Table 1 are designed to support these correlation patterns. High heap utilization alone might reflect normal workload variation, but when it appears alongside increased GC pauses and rising HTTP latency, the combined signal indicates genuine memory pressure requiring intervention [11], [12], [22], [23]. Similarly, consumer lag paired with broker under-replicated partitions and elevated disk I/O suggests infrastructure saturation rather than application-level processing delays [10], [6], [8], [9]. Alert configurations should evaluate these relationships rather than treating each metric in isolation, using the threshold values from Table 1 as starting points that teams tune based on their specific workload characteristics and SLO requirements [11], [12].

9.3 Diagnostic Workflow Methodology

Effective incident response depends on moving smoothly between different telemetry sources [14], [15], [16], [20], [30], [31]. Operators typically start with unified dashboards that show golden signal anomalies, then use the metrics framework to work their way down. You begin with user-facing symptoms like HTTP latency, investigate contributing factors such as database connection pool exhaustion or consumer lag, and eventually find root causes in infrastructure like CPU throttling or node memory pressure [6], [8], [9], [11], [12], [18]. Table 1 organizes these metrics by layer to guide this top-down diagnostic approach.

Cross-layer correlation makes diagnosis faster. When HTTP latency increases at the same time as consumer lag, you should check Kafka broker health and infrastructure metrics to see if storage or network saturation is affecting both synchronous and asynchronous paths [6], [8], [9], [18]. Distributed traces show you which operations are slow, logs give you context about specific

failures, and correlation mechanisms connect aggregate metric spikes to individual traces for deeper investigation [10], [14], [15], [16], [19], [20]. Automated triage helps by capturing thread dumps, heap profiles, and slow query logs before pods restart, preserving evidence you'll need for post-incident analysis [22], [23], [28].

10. CONCLUSION

This work examines the transition from reactive monitoring to proactive observability in distributed Java microservices and demonstrates how a unified telemetry framework enables reliable operation in complex cloud environments [3], [5], [10], [11], [12]. By integrating metrics, distributed tracing, and structured logging with asynchronous signals from Kafka and platform telemetry from Kubernetes, the framework provides comprehensive visibility across synchronous and asynchronous execution paths [14], [15], [16], [17], [18], [19], [20], [25]. The analysis shows that correlating these signals is essential for identifying root causes of performance degradation, particularly in transient environments where ephemeral workloads and autoscaling behavior can obscure systemic failures [8], [9], [14], [15], [16], [20]. The operational metrics and alerting strategies defined in this framework give SRE teams the diagnostic precision needed to maintain service reliability across distributed architectures [6], [15], [17], [18], [21].

The work also highlights the importance of adopting vendor-neutral telemetry standards to ensure long-term operational flexibility [15], [16]. Standards such as OTel and the World Wide Web Consortium Trace Context provide consistent instrumentation across diverse runtimes, messaging systems, and infrastructure platforms without binding organizations to proprietary agents or data formats [15], [16]. As distributed systems continue to expand in scale and complexity, the ability to correlate application behavior with underlying platform resource conditions will remain central to maintaining reliability [8], [9], [14], [15], [16], [20]. By prioritizing structured telemetry and unified correlation workflows, SRE teams can reduce MTTR and ensure that enterprise applications meet their SLOs in dynamic, high-concurrency environments [11], [12], [13], [27].

REFERENCES

[1] S. Newman, *Building microservices: designing fine-grained systems*, 2nd ed. " O'Reilly Media, Inc.," 2021.

[2] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[3] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A Survey on Observability of Distributed Edge & Container-Based Microservices," *IEEE Access*, vol. 10, pp. 86904–86919, 2022, doi: 10.1109/ACCESS.2022.3193102.

[4] Cloud Native Computing Foundation (CNCF), "CNCF Cloud Native Interactive Landscape," Cloud Native Computing Foundation (CNCF). [Online]. Available: https://landscape.cncf.io/

[5] S. Niedermaier, F. Koetter, A. Freymann, and S. Wagner, "On Observability and Monitoring of Distributed Systems – An Industry Interview Study," in *Service-Oriented Computing*, S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, Eds., Cham: Springer International Publishing, 2019, pp. 36–52. doi: https://doi.org/10.1007/978-3-030-33702-5_3.

[6] M. Kleppmann, *Designing data-intensive applications*. O'Reilly Beijing, 2017.

[7] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[8] Cloud Native Computing Foundation (CNCF), "Kubernetes Documentation: System Metrics," Cloud Native Computing Foundation (CNCF). [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/system-metrics/

[9] Cloud Native Computing Foundation (CNCF), "Kubernetes Documentation: Resource Management for Pods and Containers," Cloud Native Computing Foundation (CNCF). [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[10] N. Kratzke, "Cloud-native observability: the many-faceted benefits of structured and unified logging—a multi-case study," *Futur. Internet*, vol. 14, no. 10, p. 274, 2022, doi: https://doi.org/10.3390/fi14100274.

[11] N. R. Murphy, B. Beyer, C. Jones, and J. Petoff, *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.," 2016.

[12] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The site reliability workbook: practical ways to*

implement SRE. " O'Reilly Media, Inc.," 2018.
[13] Splunk Inc, "The State of Observability Report 2022," Splunk Inc. [Online]. Available: https://www.splunk.com/en_us/form/state-of-observability.html
[14] Elastic, "Elasticsearch Guide: Observability and Monitoring," Elastic. [Online]. Available: https://www.elastic.co/guide/en/observability/current/
[15] OpenTelemetry Authors, "OpenTelemetry Specification," OpenTelemetry Authors. [Online]. Available: https://opentelemetry.io/docs/specs/
[16] World Wide Web Consortium (W3C), "Trace Context Level 1," W3C Recommendation. [Online]. Available: https://www.w3.org/TR/trace-context/
[17] VMware Inc, "Micrometer Application Observability Facade," VMware, Inc. [Online]. Available: https://micrometer.io/docs
[18] Apache Software Foundation, "Apache Kafka Documentation: Operations and Monitoring," Apache Software Foundation. [Online]. Available: https://kafka.apache.org/documentation/#monitoring
[19] Cloud Native Computing Foundation (CNCF), "Prometheus Documentation: Concepts and Metrics," Cloud Native Computing Foundation (CNCF). [Online]. Available: https://prometheus.io/docs/concepts/metric_types/
[20] A. Parker, D. Spoonhower, J. Mace, R. Isaacs, and B. Sigelman, *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.
[21] VMware Inc, "Spring Boot Reference Documentation," VMware, Inc. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/
[22] Oracle Corporation, "Java SE 17 HotSpot Virtual Machine Garbage Collection Tuning Guide," Oracle Corporation. [Online]. Available: https://docs.oracle.com/en/java/javase/17/gctuning/
[23] Oracle Corporation, "Java SE 17 JVM Tool Interface Specification," Oracle Corporation. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html
[24] Amazon Web Services Inc, "Observability at Amazon," AWS Whitepapers. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/observability-at-amazon/
[25] Microsoft Corporation, "Monitoring Azure Kubernetes Service (AKS) clusters," Microsoft Corporation. [Online]. Available: https://learn.microsoft.com/en-us/azure/aks/monitor-aks
[26] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo, "Microservices integrated performance and reliability testing," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, in AST '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 29–39. doi: 10.1145/3524481.3527233.
[27] New Relic Inc, "Observability Forecast 2022," New Relic, Inc. [Online]. Available: https://newrelic.com/resources/report/observability-forecast
[28] Dynatrace LLC, "Microservices and Container Monitoring," Dynatrace LLC. [Online]. Available: https://www.dynatrace.com/platform/microservices-monitoring
[29] Datadog Inc, "The State of Cloud-Native Security," Datadog, Inc. [Online]. Available: https://www.datadoghq.com/state-of-cloud-native-security/
[30] Cloud Native Computing Foundation (CNCF), "Jaeger Distributed Tracing Platform Documentation," Cloud Native Computing Foundation (CNCF). [Online]. Available: https://www.jaegertracing.io/docs/latest/architecture/
[31] OpenZipkin Contributors, "Zipkin: A distributed tracing system," OpenZipkin Contributors. [Online]. Available: https://zipkin.io/
[32] B. Wooldridge, "HikariCP: High-performance JDBC connection pool," GitHub Repository. [Online]. Available: https://github.com/brettwooldridge/HikariCP
[33] Grafana Labs Inc, "Grafana Documentation: Introduction to Observability," Grafana Labs, Inc. [Online]. Available: https://grafana.com/docs/grafana/latest/
[34] IEEE, "IEEE Standard for Software Reviews and Audits," in *IEEE Std 1028-2008*, 2008, pp. 1–53. doi: 10.1109/IEEESTD.2008.4601584.
[35] A. Shostack, *Threat modeling: Designing for security*. John wiley & sons, 2014.