International Journal of Multidisciplinary Research and Growth Evaluation.

Defining Operability for Web Services: Principles, Metrics, and Practices

Nikhita Kataria
Independent Researcher Manager, Software Engineering, USA

* Corresponding Author: **Nikhita Kataria**

Article Info

ISSN (online): 2582-7138
Volume: 06
Issue: 04
July - August 2025
Received: 14-05-2025
Accepted: 15-06-2025
Published: 09-07-2025
Page No: 686-690

Abstract
This text addresses the important of production ready operability for web services specifically for the ones running in cloud. It outlines key system metrics and Java Virtual Machine (JVM) metrics that are essential for monitoring health and performance of cloud services. It also outlines key upstream and downstream metrics that are needed to ensure end to end monitoring is built before a service is said to be production ready. In addition to the metrics, we highlight the importance of other aspects such as centralized logging, distributed tracing that allows software engineers and other personas to quickly debug incidents and perform effective root cause analysis. We outline last mile aspects of what an effective incident response run book should contain which can be deemed effective ad ready to be used by on call engineer's applications.

DOI: https://doi.org/10.54660/.IJMRGE.2025.6.4.686-690

Keywords: Operability, High Availability, System Metrics, JVM Metrics, Upstream/Downstream Monitoring, Centralized Logging, Distributed Tracing, Incident Response Runbooks

Introduction

Operability in distributed systems is comprised of multiple facets with most important bein usability, serviceability, and practicality. A system is considered *highly operable* when it reduces the time and effort required to detect issues, identify root causes, and restore service reliability during unplanned incidents. To achieve this, systems must adhere to key operational standards across four core domains:

- **Monitoring and Reporting:** Systems must expose real-time metrics at both infrastructure and application levels. Effective monitoring, combined with alerting and visualization tools, enables early anomaly detection and rapid response.
- **High Availability (HA):** Resilience must be built into the architecture through redundancy, load balancing, and automated failovers. High availability ensures minimal downtime and contributes directly to meeting SLOs
- **Traceability:** Centralized logging and distributed tracing are essential for understanding request flows and diagnosing failures in complex, interconnected services. These tools improve visibility across system boundaries
- **Documentation:** Comprehensive, up-to-date documentation—covering runbooks, architecture, and escalation procedures—empowers teams to respond efficiently, especially during critical incidents.

In the following sections we explore more details on recommended metrics that would help faster detection thus reducing Mean Time to Detect (MTTD) and effective runbooks to reduce Mean Time to Recover (MTTR). Prioritizing operability leads to more resilient systems and a convenient on-call.

Monitoring and Reporting

Understanding which metrics are needed for which service is a necessity in todays distributed stack. It ensures that system regressions and outages can be diagnosed in timely manner and exact root causes are spotted faster. In this section we explore metrics, their role, common aggregations to monitor along with thresholds studied from industry experience and local test. They have proven to be instrumental in spotting the exact root cause faster to provide fast resolution of unplanned incidents.

A. Service Availability and Degradation Indicators

To ensure continuous service reliability, it is essential to implement comprehensive monitoring solutions that reflect the real-time health and responsiveness of the application.

1. **Synthetic Monitoring:** Tools like Nagios or other external health-check frameworks should be employed to continuously test the availability of core endpoints. These tests must go beyond administrative endpoints (e.g., /admin) and include well-formed requests to user-facing or business-critical endpoints (e.g., RESTful GET operations).
2. **Service Reporting Dashboard (SRD) Integration:** Integrating with internal or external service dashboards is critical to track and visualize key performance indicators (KPIs). These dashboards should offer insight into:

a) **Availability:** It should be an indicator of if the service is running fine such as uptime and also if it is reliable as in if its producing correct results.
b) **Latency:** Latency for various operations aggregated at percentiles of P50, P90 as a quick check for client experience.
c) **Error Rate:** Rate of errors segregated by different operations such as get error rate is to be monitored separately than the put error rate.
d) **Throughput:** Overall throughput to quickly detect peak traffic.

When services are instrumented with these metrics, engineering teams can spot issues faster and trace them back to specific infrastructure events or changes in the code. This is key to handling incidents effectively and building long-term reliability.

B. System Level Metrics

For applications with single instances i.e. single-tenant applications where these instances can be deployed on a dedicated virtual machine or probably on a single host in a data center due to security reasons, it is important to monitor the system level metrics at per instance granularity. These metrics include various resource utilization metrics for CPU, memory, disk and even network. For instance, if a service is the only service on a host these metrics can be attributed directly otherwise a correlation is needed with noisy neighbors if present. In use cases where multiple instances and mostly different web services run on a single virtual machine or even a host with local containers i.e., metrics need to be tracked at group level so that an observer can get clear attribution as to which metric is related to which service. It becomes necessary in such cases to monitor system metrics on container level granularity. This approach prevents noisy-neighbor effects and enables more accurate capacity planning and incident debugging.

Table 1 breaks down the core infrastructure metrics every team should be tracking. It shows what to measure, how granular your data should be, and what "healthy" looks like—based on lessons learned from real production environments at scale.

Table 1: System Metrics for tracking

Title	Purpose	Granularity	Threshold
Disk Utilization	Track disk usage	P99, P95, P50, Average	P99 < 99%, P95 < 70%, P50 < 50%
Disk Performance (iowait)	Track disk performance	P99, P95, P50, Average	P99 < 20%, P95 < 10%, P50 < 5%
CPU Utilization	Track CPU usage	P99, P95, P50, Average	P99 < 99%, P95 < 70%, P50 < 50%
CPU steal time	Track CPU contention due to hypervisor stealing	P99, P95, P50, Average	P99 < 5%, P95 < 2%, P50 < 1%
Memory Utilization	Track memory usage	P99, P95, P50, Average	P99 < 99%, P95 < 70%, P50 < 50%
Memory + Swap Utilization	Track memory w/ swap usage	P99, P95, P50, Average	P99 < 99%, P95 < 70%, P50 < 50%
Network interface utilization	Track network utilization	P99, P95, P50, Average	P99 < 99%, P95 < 70%, P50 < 50%
OOM Killer (groups Metrics)	Track how often the process is being killed by the OOM killer.	Rate over time interval	Rate ≤ 0.1% per hour

C. JVM Metrics (applicable only for java services)

Monitoring JVM metrics is critical for ensuring the health and performance of Java web services in production environments. These metrics provide insights into resource utilization, garbage collection behavior, and thread activity, all of which directly impact application responsiveness and stability. Table 2 presents is a detailed overview of essential JVM metrics to consider, including their monitoring granularity and suggested thresholds.

Table 2: JVM Metrics for Web Services

Title	Purpose	Granularity	Threshold
Thread count	Track number of JVM threads	Average	Depends on size of heap memory
Heap Memory (Free, Used, Max, Total)	Track heap memory usage.	Average	Free memory > 15% Used memory <= 85%
Non heap committed memory	Track non-heap committed memory that is allocated for use but has not been used yet.	Average	Should remain stable; significant increases may indicate leaks
GC Count	Number of GC events over the last minute	Average (per minute)	Ideally < 10 per minute; frequent spikes may indicate pressure
GC Duration	Average, Max, 90Pct of GC durations over the last minute	Average, Max, 90pct	Average < 200 ms, Max < 1 s, 90th percentile < 500 ms
Last Garbage collection duration	Track if garbage collection is happening often to reclaim memory	Average	< 5 seconds
Collection Count	Number of GC events since the JVM started	Average	No fixed threshold; use for trend analysis
Collection Time	Cumulative duration of GC events since the JVM started	Average	Ideally < **5%** of total uptime

D. Upstream/Downstream Metrics

This section outlines critical metrics for monitoring both upstream and downstream applications that interact with your service. Understanding the performance and reliability of these dependencies is essential for diagnosing issues, managing SLAs, and maintaining overall system stability.

Table 3 presents upstream and downstream metrics a web service should ideally track for being production ready.

Table 3: Upstream and Downstream Metrics.

Title	Purpose	Granularity	Threshold
Downstream Latency/Response Time	Track latency for any downstream applications.	P95, P99, Average	Depends on the SLA with downstream application.
Downstream Error Rate	Track error rate for any downstream applications.	P95, P99, Average	P99 < 1%, P95 < 0.05%, P50 < 0.01%
Downstream Availability	Monitor availability of downstream	P95, P99, Average	P99 > 99.9%
Upstream Error Rate	Track error rate for any upstream applications that are tightly coupled with your application.	P95, P99, Average	P99 < 1%, P95 < 0.05%, P50 < 0.01%
Upstream Availability	Monitor availability of upstream as an application might see fluctuating load in case an upstream is flaky.	P95, P99, Average	P99 > 99.9%

E. Database Metrics

This section details important metrics for monitoring database connectivity and performance, focusing on MySQL and CosmosDB databases. These metrics provide insights into connection usage, latency, replication health, and connection pool status, all of which are critical for maintaining database responsiveness and reliability in production environments.

Table 4

Title	Purpose	Granularity	Threshold
Mysql Connection Count	Track Parallel Connection Counts	P99, P95, P50, Average	< 80-90% of max allowed connections
Mysql Connection Latency - Read	Track Overall connection latency for reads	P99, P95, P50, Average	P99 < 0.1s, P95 < 0.05s, P50 < 0.01s
Mysql Connection Latency - Write	Track Overall connection latency for writes	P99, P95, P50, Average	< 1s
Mysql Connection pool thread count	Track Mysql connection pool usage	P99, P95, P50, Average	Typically < 100 threads
Mysql Master Slave Replication Lag	Track Replication lag	Average	P99 < 10s, P95 < 5s, P50 < 1s
CosmosDB Connection Count	Track Parallel Connection Counts	P99, P95, P50, Average	< 80-90% of max allowed connections
CosmosDB Connection Latency - Reads	Overall database latency for reads	P99, P95, P50, Average	P99 < 0.1s, P95 < 0.05s, P50 < 0.01s
CosmosDB Connection Latency - Writes	Overall database latency for writes	P99, P95, P50, Average	P99 < 10s, P95 < 5s, P50 < 1s
CosmosDB Connection pool thread count	Database connection pool usage	P99, P95, P50, Average	typically < 100 threads
CosmosDB Replication lag	Replication lag	Average	P99 < 0.1s, P95 < 0.05s, P50 < 0.01s

Table 5:

Symptom	Causes
Overall P99 Latency > 1 minute	High CPU, High Disk, Service Throttling, DB connection timeout
Throughput < X number of queries	Connection timeouts, High thread count
Error Rate > 10%	Code issues, Service timeouts (5xx)

F. Alerting

A. Cause and Symptom Relationship

Metrics can be effectively categorized into causes and symptoms. Symptoms represent the observable behaviors or issues a service exhibit as a result of various internal or external factors, which are the causes. For instance, elevated disk or CPU utilization (cause) can lead to performance degradation, such as reduced throughput or increased latency (symptom). Here are some examples illustrating the mapping between symptoms and their potential causes

In use cases where a new service is being developed or being refactored, it is important to follow the below stated practices:

1. **Alert on Symptoms First:** Focus alerting on symptoms that directly impact the service's behavior or user experience.
2. **Use Metrics to Identify Causes:** Utilize cause metrics to diagnose and pinpoint the underlying issues driving the symptoms.
3. **Document Symptom-to-Cause Mapping:** Clearly define and document the relationship between symptoms and their causes within the alert metadata, such as the notes or incident tracking URL fields. This helps responders quickly understand potential root causes when an alert fires.

A. Alert Categorizations

Every web service should organize alerts into various severity levels which should be well understand and have a clear incident response plan which is understandable by each engineer in the team. Industry standard is to follow priority-based alerting to ensure the impact and urgency of the issue. A sample classification is:

1. **Priority 0 (P0): Critical Alerts**: An engineer would typically wake up for these alerts if they are off business hours as it indicates severe incidents such as a service is unavailable on more than 50% of the hosts causing a significant capacity shortage. These alerts need to be catered to as soon as possible to prevent client errors.
2. **Priority 1 (P1): High Priority Alerts:** These alerts are

important however might not be as critical as a service being down. The response time for P1's is generally in hours while P0's are generally minutes. An example in this category is a CPU spike that might happen during peak traffic. In this scenario an engineer needs to be aware of the load and vigilant however an immediate action might not be necessary. These alerts if unattended may (and should) result into a P0 alert being triggered if not resolved within a certain time period.

3. **Priority 2 (P2): Informational and Low Priority Alerts**: This category includes all other alerts that do not pose an immediate threat to service availability or performance. P2 alerts serve as early warnings or informational signals that can be addressed during routine maintenance or as part of ongoing service improvements.

By defining clear severity levels and corresponding response plans, teams can prioritize their efforts effectively, ensuring critical incidents receive immediate attention while less urgent issues are tracked and resolved in due course.

C. Alert Coverage

Alerts should focus on key areas to ensure timely detection of issues to ensure comprehensive coverage while reducing noise and focusing on impactful issues.

1) **Application Metrics (Symptoms)**: Alert on symptoms of failure like high latency, error rates, or low throughput, as detailed in the Monitoring and Reporting section. Avoid alerting on all causes; focus on symptoms that impact service behavior.
2) **System Metrics**: Monitor and alert on critical system-level metrics such as CPU, memory, disk usage, and network performance to catch resource issues before they affect the application.
3) **Upstream and Downstream Dependencies**: Track latency, error rates, and availability of upstream and downstream services. Early alerts on these dependencies help prevent cascading failures.

D. ITR (Incident Response) standards

Every alert added to a monitoring system and include a clear and detailed incident response plan covering the following elements:

1) **Alert Description**: A precise explanation of what triggers the alert. For example: "This alert fires when the number of 5xx errors returned by Service X remains high for 'm' consecutive minutes. These errors typically occur when the service is overloaded (e.g., returning 502 errors) or experiencing internal crashes."
2) **Source Code Location**: Identify the specific code path or module where the alert originates.
3) **Debugging Instructions**: Steps to investigate the issue, including:

a) Relevant logs to review
b) Key metrics to check for confirming symptoms and identifying causes
c) Upstream and downstream metrics to examine

4) **Escalation Guidelines**: Clear criteria for escalating the issue if it remains unresolved after 'x' minutes, including who to notify and how.

G. High Availability

High Availability (HA) is a critical requirement for all services deploying on the cloud. The overall deployment strategy leverages services configured for fault tolerance across multiple cloud fault domains. Services onboarding to the cloud should prioritize building resiliency into their code. From an operational perspective, HA involves several key aspects:

A. **Automated Recovery and Restarts**: The system should automatically recover from unplanned issues such as crashes or disk exhaustion without manual intervention.
B. **Seamless Traffic Failover**: Traffic should automatically and transparently switch to healthy instances if any instance starts returning errors. (Note: Validation is needed to confirm if services support this capability fully.)
C. **Staged Rollouts**: Deployments should be rolled out incrementally to reduce risk. Services support rolling upgrades by updating a limited percentage of instances at a time (e.g., 20% of instances per batch), preventing complete downtime during deployment.
D. **Over-Provisioned Capacity**: Services should allocate at least 20% more capacity than the estimated requirement to handle unexpected spikes or failures. For services written in Python or Java, performance testing should cover:

1. **Load/Stress Tests:** Identify service limits, such as the maximum queries per second it can handle at acceptable resource usage.
2. **Longevity Tests:** Detect regressions like memory leaks or performance degradation over time.
3. **Capacity Estimation:** Use test results to guide capacity planning.

E. **Load Balancer Integration**: Deploy services behind a load balancer that routes traffic to healthy instances. Although the current setup may be limited to a single availability zone, properly configured services ensure fault tolerance across multiple fault domains.

Runbook standards

To ensure consistent and effective incident management, every service deployed on the cloud must maintain a comprehensive runbook detailing the resolution procedures for any alerts triggered. Operational runbooks and incident response plans (IRPs) should be precise, actionable, and adhere to a standardized structure as outlined below.

1. **Service Architecture and Overview**: A succinct description of the service, its objectives, and high-level design, providing essential context for responders
2. **Service Ecosystem**: Documentation of upstream and downstream dependencies that may influence or be influenced by the service's health.
3. **Monitoring and Alerting Links**: Direct references to relevant dashboards and monitoring tools to facilitate rapid diagnosis.
4. **Service Checklist**: A summary of key service attributes including: Service name, Runbook hyperlink, Primary purpose and functionality
5. **Issue-to-Resolution Mapping:** For each alert, at minimum include Description, Resolution and Point of Contact.
6. **Escalation Procedures**: Defined criteria and steps for

escalating unresolved issues, including escalation points and timeframes.

7. **Communication and Reporting Channels**: Establish communication protocols such as email aliases or ticketing queues to streamline issue reporting and tracking.

Centralized Logging

Every cloud service should provide centralized access to its access, application, and error logs. This can be achieved through:

1. **Centralized Log Aggregation**: Multiple companies generally would either develop propriatory logging framwork or complire frameworks such as Splunk or Elasticserach for their workloads. It is improtant to create logs in such as way that they have standard metadata like application name, log line, log file, a request id for effective debugging and searching
2. **Distributed Tracing**: Tracing is an easy and a hard approach at the same time because it is easy to use a tracing solution like Jaeger, or OpenTelemetry however the logs should have metadata assocaiated with unique identifies for tracking specific operations and requests as they flow from service to service.
3. **Structured Logging**: Use structured logs with metadata (e.g., request IDs, timestamps) to enable quick correlation and troubleshooting.
4. **Access Control and Retention**: Protect log data with proper permissions and set retention policies based on compliance and cost.

Conclusion

This paper has outlined a comprehensive framework for defining and enhancing operability in web services, focusing on principles, metrics, and best practices critical to achieving high availability and robust disaster recovery. By identifying key system-level and JVM-specific metrics, alongside upstream and downstream dependency monitoring, the framework enables timely detection and diagnosis of service degradations. The integration of centralized logging and distributed tracing strengthens visibility and root cause analysis capabilities across distributed services. We outline the importance of having exhaustive and standard runbooks. Together, these practices support resilient, scalable, and highly available web services, ultimately reducing Mean Time to Detect (MTTD) and Mean Time to Recover (MTTR) and improving overall service reliability and operability in cloud environments.

References

1. Beyer C, Jones J, Petoff J, Murphy NR. Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media; 2016. Available from: https://sre.google/sre-book/service-level-objectives/ [Accessed 2025 May 18].
2. Nygard MT. Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf; 2007.
3. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications; 2018.
4. OpenTelemetry Contributors. OpenTelemetry: Observability for Cloud-Native Software [Internet]. Available from: https://opentelemetry.io/ [Accessed 2025 May 18].
5. DigitalOcean. Cloud Metrics: The 8 Most Important Metrics to Monitor [Internet]. DigitalOcean; 2022 Oct 20. Available from: https://www.digitalocean.com/resources/articles/cloud-metrics.
6. Breyter M, Rojas C. Reliability Engineering in the Cloud: Strategies and Practices for AI-Powered Cloud-Based Systems. 1st ed. Hoboken, NJ: Addison-Wesley Professional; 2025