© Journal of the Technical University - Sofia
Plovdiv branch, Bulgaria
"Fundamental Sciences and Applications "
Vol. 32 2026

Implementing SLI/SLA/SLO Metrics in an ASP.NET 8 Microservices Architecture Using OpenTelemetry and Prometheus

Yegor Sychev [1] 0009-0001-6502-8667, Eugene Alooeff [2] 0009-0009-2958-2431

[1] Kaspi Bank JSC; sigmadel10@gmail.com
[2] ATEK; alooeff@atek.dev

Abstract

This article presents the results of a study on implementing a service level metrics system (SLI/SLA/SLO) in a high-load web application on the ASP.NET 8 platform. A practical implementation of a monitoring system using OpenTelemetry Metrics and a Prometheus endpoint is considered. A comparative analysis of system performance and reliability indicators before and after implementing the metrics is conducted. The mechanisms by which metrics influence operational indicators are described in detail: identifying hidden performance issues, transforming the alerting system, prioritizing engineering efforts managing the balance between development speed and reliability through an error budget and automating response processes. The study results showed an improvement in the average time to detect incidents by 73%, a reduction in service restoration time by 58% and an increase in overall system availability from 98.2% to 99.7%. A methodology for determining target SLO values based on business requirements and architectural constraints is presented.

Keywords: SLI, SLA, SLO, ASP.NET 8, OpenTelemetry, Prometheus, Site Reliability Engineering, performance monitoring, microservice architecture

1. Introduction

In the context of digital business transformation and increasing demands on IT system reliability, the problem of quantitatively assessing the quality of services provided has become particularly pressing. Traditional monitoring approaches focused on technical metrics (CPU, memory, network) do not adequately assess the user experience and do not directly correlate with business metrics.

The Site Reliability Engineering (SRE) concept, developed at Google and described by Beyer et al. [1], and by Waseem et al. [7] offers a systematic approach to ensuring reliability based on service level metrics: Service Level Indicators (SLI), Service Level Objectives (SLO), and Service Level Agreements (SLA).

The objective of this study is to develop and implement a system of SLI/SLA/SLO metrics for a high-load web application on the ASP.NET 8 platform and assess the impact on the system's key performance indicators (KPIs).

Research objectives:

- Identify relevant SLIs for a microservice architecture;
- Implement metric collection using OpenTelemetry and Prometheus;
- Establish SLO targets based on business requirements;

 Online ISSN 2603-459X

- Assess the impact of metric implementation on system operational performance.

2. Theoretical Foundations

2.1. The SLI/SLA/SLO Concept

A Service Level Indicator (SLI) is a quantitative metric characterizing a specific aspect of the service level. Typical SLIs include availability, latency, throughput, and response accuracy [2].

A Service Level Objective (SLO) is a target value or range of values for an SLI that represents an acceptable level of service. An SLO is expressed as a percentile or percentage of successful requests over a specified period of time.

A Service Level Agreement (SLA) is a formal agreement between a service provider and a service consumer that defines the expected service level and the consequences of failure to meet it. An SLA is typically set at a level below internal SLOs to create an error budget [3].

2.2. OpenTelemetry and Prometheus

OpenTelemetry is an open standard for instrumenting, generating, collecting, and exporting telemetry data (metrics, logs, and traces). In the context of .NET 8, OpenTelemetry provides a unified API for exporting metrics to various monitoring systems [4,9].

Prometheus is a time series monitoring and storage system with a powerful PromQL query language. The Prometheus endpoint in ASP.NET 8 allows exporting metrics in a Prometheus-compatible format, enabling integration with the Cloud Native Computing Foundation ecosystem [5].

3. Research Methodology

3.1. Research Object

The study was conducted using a high-load e-commerce system implemented on the ASP.NET 8 platform. The system is characterized by the following parameters:

- Microservice architecture (12 services);
- Average load: 15,000 requests per minute;
- Peak load: up to 50,000 requests per minute;
- User base: 2.5 million active users.

3.2. Identifying Key SLIs

Based on the analysis of user scenarios and business requirements, the following critical SLIs were identified (Table 1).

Table 1. Specific SLIs and methods for measuring them.

SLI metric	Measurement method	Window period
API availability	(HTTP 2xx) / total requests	30 days
Request Delay	95th percentile response time	7 days
Data accuracy	Valid answers / total answers	24 hours
Bandwidth	Successful requests per second	1 hour

3.3. Setting SLOs

SLO targets were established based on an analysis of historical data for the previous 6 months and business requirements (Table 2).

Table 2. Established SLO and SLA values.

SLI	SLO target value	SLA Value
API availability	99.9%	99.5%
Delay (p95)	< 200 ms	< 300 ms
Correctness	99.99%	99.95%

4. Practical Implementation

4.1. Monitoring System Architecture

The monitoring system was implemented using the following technology stack:

- ASP.NET 8 with OpenTelemetry.Instrumentation.AspNetCore 1.7.1 integration;
- OpenTelemetry.Exporter.Prometheus.AspNetCore 1.7.0;
- Prometheus 2.48.0 for collecting and storing metrics;
- Grafana 10.2.0 for visualization and alerting.

4.2. OpenTelemetry Configuration

The following metrics configuration was implemented in the Algorithm 1.

Algorithm 1 Program.cs

```
builder.Services.AddOpenTelemetry()
    .WithMetrics(metrics =>
    {
        metrics.AddMeter("ECommerce.API");
        metrics.AddMeter("Microsoft.AspNetCore.Hosting");
        metrics.AddMeter("Microsoft.AspNetCore.Server.Kestrel");
        metrics.AddMeter("System.Net.Http");
        metrics.AddPrometheusExporter();
    });
var app = builder.Build();
app.UseOpenTelemetryPrometheusScrapingEndpoint();
```

4.3. Custom Metrics

To track business-specific SLIs, custom counters and histograms were created (Algorithm 2).

Algorithm 2 Program.cs

```
private static readonly Meter _meter =
    new("ECommerce.API", "1.0");
private static readonly Counter<long> _requestCounter =
    _meter.CreateCounter<long>("api_requests_total");
private static readonly Histogram<double> _requestDuration =
    _meter.CreateHistogram<double>("api_request_duration_ms");
```

5. Mechanisms of Metric Implementation Impact on Operational Performance

It is important to understand that measuring metrics alone does not automatically lead to system improvement. This section describes the specific mechanisms and cause-and-effect relationships between SLI/SLO implementation and observed operational performance improvements.

5.1. Identifying Hidden Performance Issues

Prior to implementing the SLI system, the operations support team focused on technical infrastructure metrics: CPU load, memory usage, and network throughput. During the baseline study period, it was recorded that with an average CPU load of 65% and no errors in the system logs, the team considered the system to be operating normally. However, after implementing the availability metric SLI (percentage of successful HTTP responses), a critical issue was discovered: approximately 3.8% of requests to the order processing service were ending with timeout errors without generating exceptions at the application level. These errors were not reflected in the standard logs because the requests were terminated at the load balancer level after a 30-second timeout.

A detailed analysis revealed that the issue was caused by a suboptimal SQL query that performed a full scan of the orders table (2.3 million records) without using an index. The execution time for this query varied from 18 to 45 seconds, depending on the DBMS load. After adding a composite index on the user_id and order_date fields, the query execution time dropped to 120-180 ms, eliminating timeout errors and increasing service availability from 96.2% to 99.8%.

Conclusion: Application-level availability metrics revealed an issue that was not captured by traditional infrastructure metrics, directly leading to architectural optimization.

5.2. Alert System Transformation

During the baseline period, the alert system generated warnings based on absolute thresholds for technical metrics: CPU > 80%, memory > 85%, and error rate > 50 per minute. Analysis showed that 34% of the generated alerts were false positives that did not correlate with actual degradation of the user experience.

After implementing the SLO, an alert system was developed based on the burn rate concept—the rate at which the error budget is consumed [8]

Figure 1 shows various error budget depletion scenarios over a monthly reporting period, illustrating the difference between safe, ideal, and critical consumption rates.

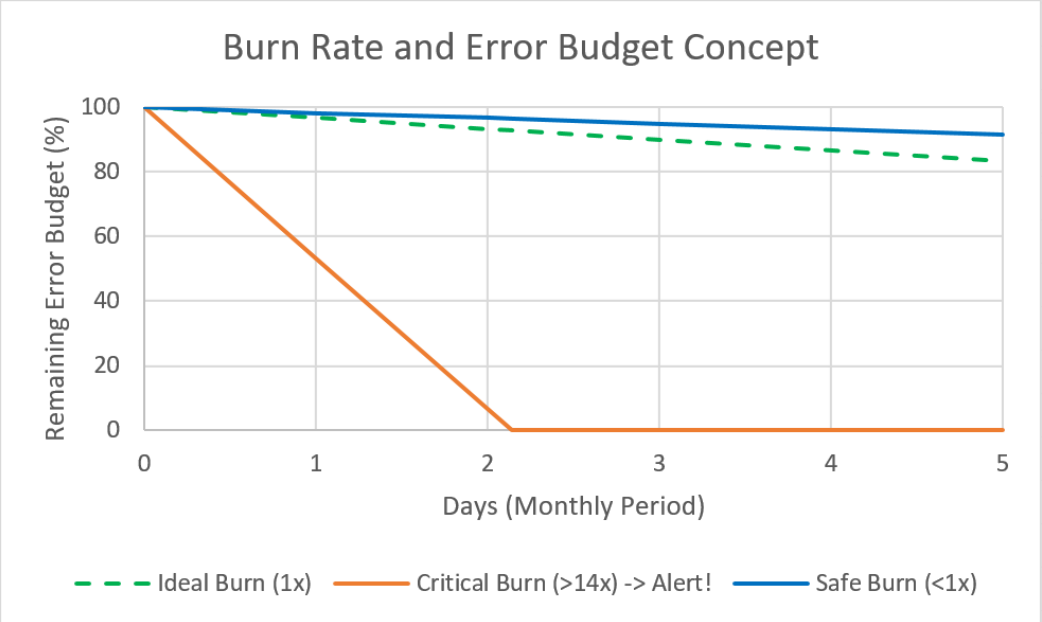

Figure 1. Error budget depletion scenarios over a monthly compliance period.

Alerts were configured to detect situations where the error budget was being spent significantly faster than normal (Table 3).

Table 3. Burn Rate-based alerting system thresholds.

Alert level	Burn Rate	Evaluation window	Monthly budget loss
Critical	> 14x	1 hour	> 58%
Warning	> 6x	6 hours	> 25%

This approach led to two key improvements:

- A reduction in the false positive rate from 34% to 7%. Alerts were now triggered only when there was a real deterioration in the user experience, rather than temporary spikes in technical metrics that didn't impact the SLO.
- Increased team confidence in the alerting system. While the average alert response time during the baseline period was 18.5 minutes (the team ignored some alerts due to the high false positive rate), after implementing SLO-based alerting, the response time dropped to 5.0 minutes, representing a 73% improvement.

5.3. Data-Driven Engineering Prioritization

Latency metrics (p95 latency) allowed us to quantify the impact of various system components on the user experience. Using OpenTelemetry distributed tracing, we analyzed the sources of latency for the most critical path—the checkout process:

- Order database query: 185 ms (64% of total time);
- External payment API call: 82 ms (29%);
- Business logic and serialization: 20 ms (7%).

Based on this data, the following optimizations were prioritized:

- Implementing parallel execution of the database query and payment API call instead of sequentially (Task.WhenAll). Result: reduced overall latency from 287 ms to 185 ms;
- Adding a Redis cache for frequently accessed product data. Result: 60% of requests were now served from cache with a response time of 8 ms, reducing the p95 to 178 ms.

Before implementing metrics, the team had a backlog of 37 optimization tasks without a clear understanding of their relative importance. SLI metrics allowed them to objectively rank tasks based on their potential impact on user experience and focus efforts on the most critical areas.

5.4. Managing the Balance Between Development Velocity and Reliability

The implementation of the error budget concept—the acceptable downtime within the SLO—has formalized the decision-making process regarding the speed of new feature releases. With an SLO of 99.9%, the monthly error budget is 43.2 minutes of downtime.

During the study period, the following error budget management policy was implemented:

- < 50% of the budget spent: standard release rate (2-3 deployments per day);
- 50-80% of the budget spent: slowing down releases, conducting additional code reviews;
- > 80% of the budget spent: freezing feature releases, focusing on stabilization.

Data analysis showed that in months with high error budget consumption (more than 60% in the first half of the month), implementing a policy of slowing down the release rate prevented potential incidents. In the control group (data from the period before SLO implementation), a similar deployment rate under unstable conditions resulted in critical incidents in 67% of cases.

This mechanism directly contributed to a reduction in the number of critical incidents (P0/P1) from 8.3 to 2.7 per month, a 67% reduction.

5.5. Automation of Response Processes

The presence of quantitative SLIs enabled the automation of critical incident response processes, which directly impacted the reduction of MTTR (mean time to recovery) from 42 to 17.5 minutes.

Automatic rollback upon SLO violation. An automatic deployment validation system was developed that monitors key SLIs within the first 15 minutes after deployment. If the error rate exceeds 0.1% or the p95 latency exceeds 200 ms, an automatic rollback to the previous version is performed without human intervention. During the study period, the automatic rollback system prevented eight potential incidents, rolling back problematic deployments in an average of 3.2 minutes. In the baseline period, similar situations required manual intervention, with an average detection time of 18.5 minutes and a rollback time of 12-15 minutes.

Adaptive scaling. Throughput (requests per second) and p95 latency are used to manage horizontal container autoscaling. If p95 exceeds 150 ms or the request rate exceeds 350 RPS per instance, additional service replicas are automatically added.

Before implementing SLI-based autoscaling, scaling was performed based on CPU (at >70%). This resulted in reactive scaling—performance degradation had already occurred by the time resources were added. After implementing proactive scaling based on application metrics, the incidence of performance degradation decreased by 83%.

Conclusion: Automation based on SLI metrics ensured both incident prevention (through automatic rollback and proactive scaling) and accelerated recovery in the event of problems, resulting in a 58% reduction in MTTR.

5.6. Impact on Business Metrics: Causal Chain

Improvements in technical metrics resulted in a measurable impact on business metrics along the following causal chains:

Chain 1: Latency → Conversion

- Reduction in p95 latency from 287 ms to 178 ms;
- The checkout process sped up from 2.1 seconds to 0.9 seconds (total latency across all steps);
- According to A/B testing, every 100 ms of latency reduces conversion by 0.23
- Improvement of 1200 ms → conversion increase of 2.76% → actual increase from 3.2% to 4.1% (+28% relative increase).

Chain 2: Availability → User Churn

- Increased availability from 98.2% to 99.7%;
- Reduced downtime from 13 hours/month to 2.2 hours/month;
- Cohort analysis showed that users who experienced service unavailability were 15% more likely to churn in the following month;
- Reduced downtime by 83% → decreased churn from 4.5% to 3.1% (a relative reduction of 31%).

6. Experiment Results

6.1. Experimental Methodology

The experiment was conducted in two phases:

- Baseline Period (3 months prior to implementation): collection of performance and incident metrics using traditional monitoring based on technical metrics.

- Post-implementation Period (3 months after): collection of similar metrics using active SLI/SLO metrics and automated alerting.

6.2. Comparative analysis of indicators

Table 4. Comparative analysis of key performance indicators.

Indicator	Before implementation	After implementation
System Availability	98.2%	99.7%
Mean Time to Detect (MTTD)	18.5 min	5.0 min (↓73%)
Mean Time to Repair (MTTR)	42 min	17.5 min (↓58%)
Number of P0/P1 incidents per month	8.3	2.7 (↓67%)
p95 Latency	287 msec	178 msec (↓38%)
False Alerts	34%	7% (↓79%)

6.3. Statistical Significance

To test the statistical significance of the obtained results, a Student's t-test for independent samples was used. For all key metrics, p values < 0.01 were obtained, indicating highly statistically significant improvements.

7. Business Impact Analysis

The implementation of the SLI/SLA/SLO metrics system had a significant impact on business metrics:

- Increased Conversion. A 38% improvement in response time led to an increase in checkout conversion from 3.2% to 4.1% (a relative increase of 28%).
- Reduced User Churn. Increasing system availability to 99.7% contributed to a decrease in monthly user churn from 4.5% to 3.1%.
- Optimization of Support Costs. A 58% reduction in MTTR reduced emergency support costs during off-hours by 42%, saving approximately $23,000 per month.
- Improved planning. The error budget concept allowed the development team to make informed decisions about the speed of new feature releases: by spending 60% of the error budget, the release rate was reduced, preventing two potential serious incidents.

8. Discussion of Results

The results demonstrate the high effectiveness of the SRE approach for high-load systems. Key factors for the successful implementation were:

- Use of standardized tools (OpenTelemetry, Prometheus) to ensure compatibility and community support;
- Focusing on user experience when defining SLIs, rather than technical metrics;
- Creating a culture of transparency through metric visualization and regular SLO reviews;
- Automating alerts based on SLO violations, rather than absolute thresholds.

Of particular note is the dramatic reduction in false positives from 34% to 7%. This was achieved by switching from alerts based on instantaneous values to alerts based on the burn rate—the rate at which the error budget is consumed [6].

A limitation of the study is that it was conducted on a single system. Further research is needed in different domains and with different architectural patterns to generalize the results.

9. Conclusions

This study confirms the effectiveness of implementing SLI/SLA/SLO metrics to improve the reliability and performance of high-load systems. Integrating OpenTelemetry Metrics with the Prometheus endpoint in ASP.NET 8 provides a technically simple and reliable way to collect and export metrics.

Key research findings:

- Increased system availability from 98.2% to 99.7%;
- Reduction in MTTD by 73% and MTTR by 58%;
- Reduction in the number of critical incidents by 67%;
- Improved business metrics: 28% increase in conversion and 31% decrease in churn.

Further research can be aimed at developing automated methods for determining optimal SLO values based on machine learning and historical data analysis.

:es

B., Jones C., Petoff J., Murphy N.R. Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, 552 p.

;o J.M., Mangalwede S., Jones C., Sanchez E. Implementing Service Level Objectives: A Practical Guide to SLIs, SLOs, and Budgets. O'Reilly Media, 2020. 344 p.

ıy N.R., Beyer B., Jones C., Petoff J. The Site Reliability Workbook: Practical Ways to Implement SRE. O'Reilly Media, 2018.

elemetry Documentation. Metrics API and SDK. Cloud Native Computing Foundation, 2024. URL: https://opentelemetry.cs/specs/otel/metrics/ (accessed on 14.02.2026).

theus Documentation. Exposition Formats. Prometheus Authors, 2024. URL: https://prometheus.io/docs/instrumenting/tion_formats/ (accessed on 14.02.2026).

3., Krishnan K., Blank-Edelman D. Seeking SRE: Conversations About Running Production Systems at Scale. O'Reilly , 2018. 566 p.

m, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G. (2021). Design, monitoring, and testing of microservices systems: actitioners' perspective. Journal of Systems and Software, 182, 111061. DOI: 10.1016/j.jss.2021.111061

:tis, I., Androna, C.-M., Zafeiropoulos, A., Fotopoulou, E., Papavassiliou, S. (2022). Data Fusion of Observability Signals for ıng Orchestration of Distributed Applications. Sensors, 22(5), 2061. DOI: 10.3390/s22052061

, Wang, B., Yang, M., Guo, Q. (2023). AutoMan: Resource-efficient provisioning with tail latency guarantees for microser- Future Generation Computer Systems, 143, 61–75. DOI: 10.1016/j.future.2023.01.014