Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Intelligent Service Level Objective

Maraka Lakshmi Meghana
KL University
Mandadi Rajesh Roshan
2300031524@kluniversity.in

KL University
Paritala Vishnu Vardhan
Kl University

Research Article

Keywords:

Posted Date: November 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11887673/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Intelligent Service Level Objective Generation and Alert Management Using Infrastructure Telemetry

Mandadi Rajesh Roshan, Maraka Lakshmi Meghana, Paritala Vishnu Vardhan, and Dr. S N V J Devi Kosuru

Department of Computer Science and Engineering
Koneru Lakshmaiah University, Andhra Pradesh, India
2300031524@kluniversity.in, 2300030408@kluniversity.in,
2300033135@kluniversity.in, jyotsnadevi@kluniversity.in

***Abstract*—Traditional infrastructure monitoring systems commonly rely on manually defined static thresholds and isolated alerting rules. While these mechanisms detect obvious outages, they frequently generate unnecessary notifications during transient fluctuations and struggle to reflect cumulative service reliability. This paper presents an intelligent Service Level Objective (SLO) generation and alert management framework built on infrastructure telemetry, error budget analysis, and burn-rate alerting. The implemented prototype uses Python, Pandas, NumPy, and Streamlit to simulate and analyze request counts, failed requests, availability, error rates, and latency. Configurable SLO targets are evaluated against computed Service Level Indicators (SLIs) to determine error budget consumption and burn rates, enabling automated classification of normal operation, slow-burn, and fast-burn conditions. Fault injection and automated PASS/FAIL validation are integrated into the workflow to verify alert behavior under simulated service degradation. Results from the local prototype demonstrate automated SLO evaluation at a configurable 99% availability target, real-time visualization of reliability metrics through an interactive dashboard, and structured alert generation during injected faults. Live Amazon Web Services (AWS) integration using CloudWatch, Lambda, and SNS, along with machine learning-based dynamic SLO optimization and multi-service monitoring, are defined as planned extensions for the next development phase.**

***Index Terms*—Site Reliability Engineering, Infrastructure Telemetry, Service Level Indicators, Service Level Objectives, Error Budget, Burn Rate, Alert Management, Streamlit, Cloud Monitoring.**

I. Introduction

Modern software applications and cloud services depend on resilient underlying infrastructure to deliver consistent performance to end users. As system architectures grow more dynamic, engineering teams collect continuous streams of infrastructure telemetry—including request volume, failure counts, availability, error rates, and response latency—to observe operational health. However, collecting telemetry alone does not guarantee timely incident detection or effective reliability management.

Conventional monitoring setups rely heavily on manually configured static thresholds, such as fixed CPU utilization limits or single-point error-rate triggers. When thresholds are configured too tightly, transient load spikes produce excessive alert noise and operator fatigue. Conversely, when thresholds are set too loosely, gradual service degradation may go unnoticed until users experience significant downtime.

Site Reliability Engineering (SRE) introduces a structured, quantitative alternative centered on Service Level Indicators (SLIs), Service Level Objectives (SLOs), and error budgets [1,2]. Rather than alerting on raw infrastructure symptoms in isolation, SLO-based monitoring measures service reliability directly against user-facing targets and tracks how quickly the permissible margin of failure—the error budget—is being consumed [3,4].

This paper describes the design and implementation of a modular prototype that connects infrastructure telemetry analysis, configurable SLO evaluation, error budget and burn-rate computation, automated alert classification, and interactive visualization within a unified local web application. Built with Python, Pandas, NumPy, and Streamlit, the prototype also incorporates fault injection and automated PASS/FAIL validation to test alerting behavior across degraded operating states.

The current implementation establishes the local telemetry simulation, reliability computation engine, rule-based burn-rate alerting, and interactive monitoring dashboard. Integration with live Amazon Web Services (AWS) telemetry and notification services (CloudWatch, Lambda, SNS, EC2, and IAM) as well as machine learning-based dynamic SLO generation are documented as planned extensions rather than completed features.

II. Problem Statement and Objectives

Traditional infrastructure monitoring systems depend on manually tuned thresholds and static alerting rules. Because these static mechanisms do not account for cumulative reliability targets or the rate of budget consumption over time, they often result in delayed failure detection, unnecessary alerts, and inefficient service reliability management.

The problem addressed in this work is the need for an integrated, telemetry-driven workflow that computes SLIs, evaluates configurable SLOs, quantifies error budget burn rates, and generates structured alerts with centralized visual observability.

The objectives of this work are:

- Collect and analyze infrastructure telemetry, including request counts, error rates, availability, and latency.
- Develop an intelligent mechanism for generating and evaluating Service Level Objectives (SLOs).
- Calculate error budgets and burn rates to assess cumulative service reliability.
- Automatically detect service degradation and generate fast-

burn and slow-burn alerts.
- Develop an interactive dashboard for visualizing infrastructure performance and service health.
- Reduce unnecessary alerts and enable proactive infrastructure management.

III. Related Work

Site Reliability Engineering principles formalized by Beyer et al. [1] establish SLIs, SLOs, and error budgets as foundational instruments for balancing feature velocity against operational stability. Rather than pursuing impractical 100% reliability targets, error budgets quantify the acceptable level of unreliability within a compliance window and provide an objective signal for operational intervention.

Building on these foundations, *The Site Reliability Workbook* [2] provides practical guidance for translating service metrics into actionable reliability policies. Thurgood and Ferguson [4] detail structured methods for selecting meaningful SLIs and defining realistic SLO targets aligned with user expectations. Complementing this, Thurgood et al. [3] examine the trade-offs of naive error-rate alerting—showing that simple threshold alerts can exhibit low precision and high alert volume—and introduce burn-rate alerting to trigger notifications only when the rate of error budget consumption threatens the SLO.

Recent research has extended SLO and error-budget governance into automated operations and AIOps domains. Solanke [5] demonstrated that burn-rate alerting over verified service outcomes substantially outperforms conventional endpoint threshold monitoring across injected failure scenarios while avoiding false alarms. Their findings reinforce the operational value of separating raw metric collection from budget-aware alerting logic.

In cloud environments, managed observability and event-driven services such as Amazon CloudWatch [6], AWS Lambda [7], and Amazon Simple Notification Service (SNS) [8] provide scalable primitives for collecting telemetry, executing serverless evaluation logic, and routing notifications. On the analytical and prototyping side, Python [9] together with Pandas [11], NumPy [12], and Streamlit [10] enables rapid construction of data-driven reliability dashboards.

The present work synthesizes these established SRE concepts and Python analytics tools into a single, self-contained prototype for telemetry analysis, SLO evaluation, burn-rate alert classification, and fault-injection validation, while outlining a clear architectural path toward live AWS deployment.

IV. Research Gap and Contributions

While SRE literature defines strong theoretical frameworks for SLIs, SLOs, and burn-rate alerting, many small-scale or academic infrastructure setups still depend on ad-hoc scripts or disconnected monitoring tools. Bridging telemetry ingestion, reliability math, alert classification, fault testing, and visual monitoring into a cohesive, testable workflow requires dedicated architectural integration.

This work addresses that integration need through the following contributions:

1. A modular Python-based reliability analysis pipeline that processes simulated infrastructure telemetry (request counts, failures, availability, error rates, and latency) using Pandas and NumPy.
2. Configurable SLO generation and automated SLI, error budget, and burn-rate computation demonstrated at a 99% availability target.
3. Rule-based burn-rate alert management capable of distinguishing normal operation, slow-burn degradation, and fast-burn incidents.
4. An interactive Streamlit dashboard providing centralized visualization of telemetry metrics, SLO compliance, burn rates, and alert states.
5. Integrated fault injection and automated PASS/FAIL validation to verify system behavior under simulated degradation, alongside a proposed AWS cloud architecture for future production deployment.

The contribution is an applied integration of SRE reliability methodologies and interactive Python visualization for proactive monitoring. It does not claim to introduce a new time-series database or replace enterprise observability platforms.

V. Framework Architecture

The framework adopts a layered, modular architecture that decouples telemetry generation, data processing, reliability evaluation, alerting, visualization, and validation. Separating these responsibilities allows the core analytical engine to operate locally on simulated telemetry today while supporting future migration to cloud-native AWS data sources.

A. Architecture Components

The nine core architectural layers are organized as follows:

- **Infrastructure Telemetry Layer:** Generates simulated infrastructure metrics including total request counts, failed requests, availability, error rates, and response latency.
- **Data Processing Layer:** Uses Python, Pandas, and NumPy to structure, aggregate, and analyze telemetry records in memory.
- **SLI Evaluation Layer:** Computes quantitative Service Level Indicators that reflect current service availability and performance.
- **SLO Generation Layer:** Establishes configurable Service Level Objectives aligned with service reliability requirements.
- **Error Budget Analysis Layer:** Computes the permissible failure margin and tracks cumulative error budget consumption.
- **Burn Rate Analysis Layer:** Evaluates how rapidly the service is consuming its allocated error budget relative to the target window.
- **Alert Management Layer:** Classifies system health into normal operation, slow-burn, or fast-burn states and generates corresponding alerts.
- **Visualization Layer:** Renders real-time charts, SLO compliance status, burn rates, and active alerts via an interactive Streamlit web dashboard.

- **Testing and Validation Layer:** Injects controlled failure scenarios and executes automated PASS/FAIL validation checks.

B. Proposed Cloud Extension

Figure 1 illustrates both the implemented local architecture and the planned cloud-native target architecture. In the proposed AWS deployment, Amazon CloudWatch [6] will ingest live infrastructure telemetry, AWS Lambda [7] will execute serverless metric processing and SLO evaluation, Amazon SNS [8] will dispatch automated alert notifications, AWS EC2 will host the web dashboard, and AWS IAM will govern access control and security permissions.

VI. System Workflow

The implemented workflow executes end-to-end whenever telemetry data is generated or fault conditions are injected through the local Streamlit interface. The operational sequence consists of nine structured stages:

1. **Infrastructure Telemetry Generation:** Produce simulated request counts, failure counts, availability, error rates, and latency streams.
2. **Data Collection and Processing:** Load telemetry into in-memory Pandas DataFrames and perform numerical aggregation with NumPy.
3. **SLI Calculation:** Derive quantitative service availability and error-rate indicators from processed telemetry.
4. **SLO Generation and Evaluation:** Apply the user-configured SLO target (e.g., 99% availability) and assess compliance.
5. **Error Budget Calculation:** Compute the permissible error allowance and remaining budget capacity.
6. **Burn Rate Analysis:** Calculate the rate of error budget consumption to detect rapid or sustained reliability loss.
7. **Automated Alert Generation:** Classify current conditions into normal operation, slow-burn, or fast-burn alert states.
8. **Interactive Dashboard Visualization:** Render time-series charts, compliance indicators, burn-rate metrics, and alert panels in Streamlit.
9. **Testing and Validation:** Execute fault injection scenarios and report automated PASS/FAIL validation outcomes.

Figure 2 summarizes this end-to-end validation and alert-management pipeline.

VII. Implementation

A. Simulated Telemetry and Data Processing

Because live cloud infrastructure instrumentation is planned for a subsequent phase, the prototype implements a configurable telemetry simulator in Python [9]. The simulator generates time-series records capturing total requests (N_{total}), failed requests (N_{failed}), availability percentage, error rate, and request latency.

All telemetry records are stored and processed entirely in memory without an external database. Pandas [11] handles structured tabular indexing and aggregation, while NumPy [12] executes vectorized numerical and statistical computations. This lightweight architecture enables rapid local iteration and responsive dashboard updates on standard desktop hardware.

B. SLI and SLO Computation

From the processed telemetry, the SLI Evaluation Layer computes service availability and error rate over the observed evaluation interval as:

$$\text{SLI}_{\text{avail}} = \frac{N_{\text{total}} - N_{\text{failed}}}{N_{\text{total}}} \times 100\%, \tag{1}$$

$$E_{\text{obs}} = \frac{N_{\text{failed}}}{N_{\text{total}}}, \tag{2}$$

where E_{obs} denotes the observed error rate. The SLO Generation Layer exposes configurable reliability targets directly in the dashboard, allowing operators to specify availability goals such as SLO $= 99\%$ (or 0.99 in fractional form) according to service criticality [4].

C. Error Budget and Burn-Rate Analysis

Following Google SRE methodology [2, 3], the Error Budget (EB) represents the maximum allowable fraction of failed requests within an evaluation window:

$$\text{EB} = 1 - \text{SLO}. \tag{3}$$

For the demonstrated 99% availability target (SLO $= 0.99$), the permissible error budget is EB $= 0.01$ (1% failure allowance).

To measure how quickly the service is depleting its error budget, the Burn Rate (BR) relates the observed error rate to the allocated error budget:

$$\text{BR} = \frac{E_{\text{obs}}}{\text{EB}}. \tag{4}$$

A burn rate of BR $= 1$ consumes the entire error budget across the nominal window, whereas values substantially greater than 1 indicate accelerated reliability erosion requiring timely intervention [3].

D. Alert Classification and Interactive Dashboard

The Alert Management Layer evaluates computed burn rates and SLO compliance to classify infrastructure health into three operational states, summarized in Table I. Separating fast-burn and slow-burn conditions enables operators to distinguish acute incidents from gradual degradation without reacting to isolated, low-impact noise.

The Visualization Layer is implemented using Streamlit [10] and native Streamlit charts. Running on a local server (`localhost:8501`), the interface provides real-time controls for configuring SLO targets, injecting simulated faults, and inspecting availability trends, latency behavior, error budget consumption, burn rates, and active alert statuses in one centralized view.

VIII. Experimental Setup and Evaluation

A. Implementation Environment

The prototype was developed and executed locally using the software stack and platform configuration listed in Table II.

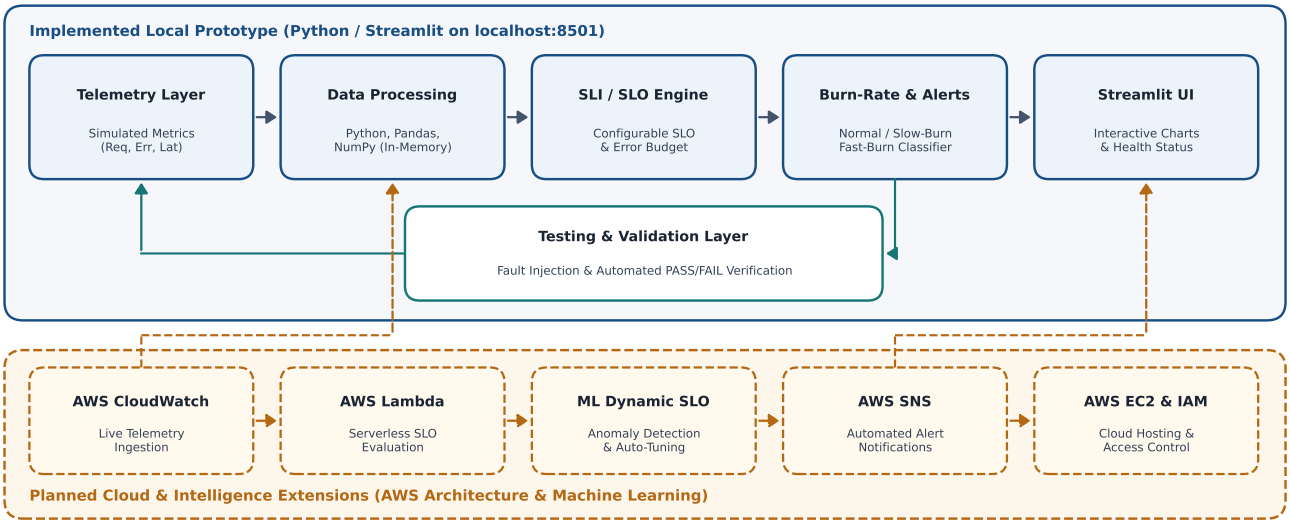

Fig. 1. Proposed modular and cloud-extended architecture of the intelligent SLO generation and alert management framework. The upper solid region represents the completed local prototype built with Python, Pandas, NumPy, and Streamlit; the lower dashed region represents planned AWS and machine learning extensions.

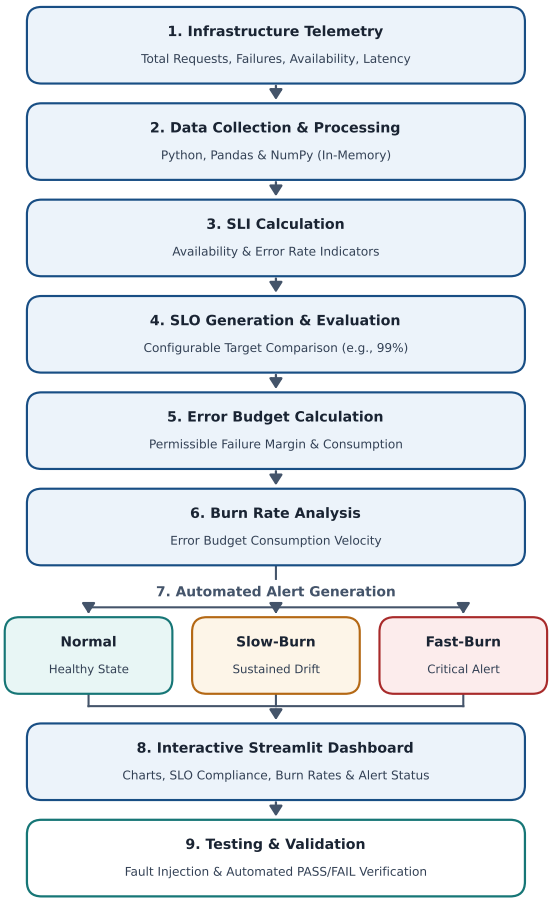

Fig. 2. End-to-end telemetry processing, SLO evaluation, burn-rate alert classification, and validation workflow implemented in the local prototype.

TABLE I
OPERATIONAL ALERT STATES IN THE PROTOTYPE

State	Observed Condition	System Action
Normal Operation	Observed error rate stays within budget; burn rate remains nominal.	Maintain healthy status on dashboard; suppress unnecessary alerts.
Slow-Burn Alert	Sustained moderate budget consumption indicating gradual service drift.	Flag slow-burn warning for proactive investigation before budget exhaustion.
Fast-Burn Alert	Rapid error budget depletion caused by severe or acute failure spikes.	Trigger high-priority fast-burn alert highlighting immediate SLO risk.

B. *Evaluation Procedure*

Validation focused on functional verification of the telemetry processing, SLO computation, burn-rate alerting, and dashboard rendering pipeline on the local Windows execution environment. The following evaluation steps were performed:

1. Launch the Streamlit monitoring application locally via Windows Command Prompt (`localhost:8501`).
2. Generate baseline simulated infrastructure telemetry covering request counts, failed requests, availability, error rates, and latency.
3. Configure the SLO availability target at 99% and verify automated SLI, error budget, and burn-rate calculations.
4. Inject controlled fault scenarios representing gradual degra-

TABLE II
IMPLEMENTATION ENVIRONMENT AND PROPOSED CLOUD STACK

Component	Technology or Platform
Programming Language	Python 3.x
Web Framework	Streamlit
Data Processing	Pandas
Numerical Computing	NumPy
Data Visualization	Streamlit Charts
Operating System	Microsoft Windows
Execution Tool	Windows Command Prompt (CMD)
Data Source	Simulated infrastructure telemetry
Data Storage	In-memory processing (no external DB)
Hosting Environment	Local Streamlit server (`localhost:8501`)
Testing Methodology	Fault injection and PASS/FAIL validation
Proposed Cloud Stack	AWS (CloudWatch, EC2, Lambda, SNS, IAM)

TABLE III
SUMMARY OF RECORDED PROTOTYPE VALIDATION

Evaluation Item	Observed Outcome
Local Application Execution	Successful on Windows via CMD (`localhost:8501`)
Configurable SLO Target	Demonstrated at 99% availability target
Error Budget & Burn Rate	Automated calculation verified in memory
Alert Classification	Normal, slow-burn, and fast-burn states identified
Fault Injection Testing	Automated PASS/FAIL validation executed
Dashboard Visualization	Centralized metrics, charts, and alerts rendered
Production Alert Benchmarks	Not yet independently quantified (planned phase)

dation and severe failure spikes.

5. Verify automated state classification across normal operation, slow-burn, and fast-burn conditions.
6. Execute automated PASS/FAIL validation checks and inspect visual updates on the interactive dashboard.

C. Observed Results

Table III summarizes the verified functional outcomes of the prototype evaluation. The local system successfully computed SLIs, evaluated the configurable 99% SLO target, calculated error budget consumption and burn rates, and differentiated normal operation, slow-burn, and fast-burn states under injected fault scenarios.

These observations confirm correct end-to-end execution of the prototype logic on simulated telemetry. They do not yet constitute an independent quantitative benchmark of alert-volume reduction or mean time to detect (MTTD) in a live production cloud environment.

IX. DISCUSSION

Integrating telemetry analysis, SLO compliance evaluation, and burn-rate alerting inside a single interactive workflow demonstrates a practical shift away from rigid static threshold monitoring. Expressing reliability through error budgets provides clear operational context: instead of reacting to every momentary latency or error fluctuation, teams can prioritize conditions that genuinely threaten the agreed SLO [3].

Using Python, Pandas, NumPy, and Streamlit keeps the prototype transparent, inspectable, and straightforward to validate through fault injection. Operators and developers can adjust SLO targets interactively and immediately observe how error budgets and burn-rate alerts respond to simulated degradation.

At the same time, the current evaluation is scoped to simulated single-service telemetry and rule-based burn-rate logic running locally. While the prototype demonstrates the structural potential to reduce unnecessary alerts and support proactive failure detection, verifying quantitative alert-reduction percentages and detection latency requires real-world production workloads and live telemetry streams.

X. LIMITATIONS AND FUTURE WORK

The current implementation establishes the local functional prototype and validation workflow, while several planned capabilities remain for subsequent development:

- **Live AWS Telemetry and Alerting Integration:** Connecting the processing pipeline to Amazon CloudWatch [6] for real-time metric ingestion, AWS Lambda [7] for serverless SLO evaluation, and Amazon SNS [8] for automated multi-channel alert delivery.
- **Machine Learning-Based Dynamic SLOs:** Extending rule-based alerting with machine learning models for historical baseline learning, dynamic SLO threshold generation, and telemetry anomaly detection.
- **Multi-Service Monitoring:** Expanding the data model and dashboard to track interconnected microservices simultaneously with per-service error budgets.
- **Cloud Deployment and Production Benchmarking:** Hosting the dashboard on AWS EC2 with IAM access controls, adding persistent telemetry storage, and conducting production-scale evaluations to independently quantify alert-fatigue reduction and failure-detection latency.

XI. CONCLUSION

This paper presented an intelligent Service Level Objective generation and alert management framework based on infrastructure telemetry, error budgets, and burn-rate analysis. Using Python, Pandas, NumPy, and Streamlit, the implemented prototype processes simulated request, error, availability, and latency metrics to evaluate configurable SLO targets, compute error budget consumption, classify normal, slow-burn, and fast-burn conditions, and visualize service health in an interactive dashboard.

Fault injection and automated PASS/FAIL validation confirmed the end-to-end execution of the prototype at a 99% availability target on a local server. The work provides a modular, testable foundation for reliability-driven monitoring. Future development will focus on live AWS CloudWatch, Lambda, and SNS integration, machine learning-based dynamic SLO optimization, multi-service support, and production-scale quantitative evaluation.

ACKNOWLEDGMENT

The authors acknowledge the support and guidance provided by Dr. S N V J Devi Kosuru and the Department of Computer Science and Engineering, Koneru Lakshmaiah University, during the development of this project.

REFERENCES

[1] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly Media, 2016. [Online]. Available: https://research.google/pubs/site-reliability-engineering-how-google-runs-production-systems/

[2] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The Site Reliability Workbook: Practical Ways to Implement SRE*. Sebastopol, CA, USA: O'Reilly Media, 2018.

[3] S. Thurgood, J. Frame, A. Lenton, et al., "Alerting on SLOs," in *The Site Reliability Workbook*, Google SRE. [Online]. Available: https://sre.google/workbook/alerting-on-slos/

[4] S. Thurgood and D. Ferguson, "Implementing SLOs," in *The Site Reliability Workbook*, Google SRE. [Online]. Available: https://sre.google/workbook/implementing-slos/

[5] A. Solanke, "From AIOps to AgentOps: Service-Level Objectives and error-budget-governed autonomy for production AI agents," *World Journal of Advanced Engineering Technology and Sciences*, vol. 21, no. 1, pp. 061–069, 2026, doi: 10.30574/wjaets.2026.21.1.0466.

[6] Amazon Web Services, "Amazon CloudWatch Documentation," [Online]. Available: https://docs.aws.amazon.com/cloudwatch/

[7] Amazon Web Services, "AWS Lambda Documentation," [Online]. Available: https://docs.aws.amazon.com/lambda/

[8] Amazon Web Services, "Amazon Simple Notification Service Documentation," [Online]. Available: https://docs.aws.amazon.com/sns/

[9] Python Software Foundation, "Python 3 Documentation," [Online]. Available: https://docs.python.org/3/

[10] Streamlit Inc., "Streamlit Documentation," [Online]. Available: https://docs.streamlit.io/

[11] The pandas development team, "pandas Documentation," [Online]. Available: https://pandas.pydata.org/docs/

[12] NumPy Developers, "NumPy Documentation," [Online]. Available: https://numpy.org/doc/stable/