MAUN Fen Bil. Dergi., 14, 1, 29-37 Araştırma Makalesi/ Research Article
MAUN J. of Sci., 14, 1, 29-37 DOI : 10.18586/msufbd.1792556

A Research on the Native Cloud Monitoring with the Help of Site Reliability Engineering Insights

Canberk KOÇ[1], Bora UĞURLU[2]✉, Bahadır KARASULU[3]
[1,2,3]Department Of Computer Engineering, Faculty Of Engineering, Canakkale Onsekiz Mart University, Canakkale, Turkey

✉: bugurlu@gmail.com [1]0009-0006-5257-8404 [2]0000-0001-6769-9563 [3]0000-0001-8524-874X

Geliş (Received): 29.09.2025 Düzeltme (Revision): 18.10.2025 Kabul (Accepted): 27.10.2025

ABSTRACT

The demand for scalable and portable applications has accelerated the adoption of distributed systems and cloud computing. To ensure reliability and availability, Site Reliability Engineering (SRE) has become essential. This study compares native cloud monitoring services of Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure within the SRE framework. A PaaS-based sample application was developed using Golang, microservice architecture, API Gateway, and event-driven patterns, supported by relational databases and cloud-native messaging services. Providers' monitoring tools, along with third-party solutions, were integrated to assess system health. Evaluation focused on objective SRE metrics: variety, cost, alert latency, data collection frequency, and retention. Findings show that GCP provides detailed VM-level metrics without additional configuration in API Gateway setups and achieves the highest message-handling capacity in event-driven architectures. The study offers a comparative analysis of monitoring capabilities and insights into platform suitability for reliability-centric deployments.

Keywords: Monitoring Systems, Site Reliability Engineering, Cloud Computing, Software Architecture

Site Güvenilirliği Mühendisliği İçgörülerinden Yararlanarak Yerel Bulut İzleme Üzerine Bir Araştırma

ÖZ

Ölçeklenebilir ve taşınabilir uygulamalara yönelik artan talep, dağıtık sistemler ve bulut bilişimin önemini artırmıştır. Güvenilirlik ve erişilebilirliği sağlamak için Site Reliability Engineering (SRE) yaklaşımı kritik hale gelmiştir. Bu çalışmada, Amazon Web Services (AWS), Google Cloud Platform (GCP) ve Microsoft Azure'un yerel bulut izleme servisleri SRE çerçevesinde karşılaştırılmıştır. Golang tabanlı, mikroservis mimarili, API Gateway ve olay odaklı desenleri kullanan bir PaaS uygulaması geliştirilmiş; ilişkisel veritabanları ve bulut-yerel mesajlaşma servisleriyle desteklenmiştir. İzleme araçları, sistem sağlığını değerlendirmek üzere üçüncü taraf çözümlerle birlikte entegre edilmiştir. Analiz; çeşitlilik, maliyet, uyarı gecikme süresi, veri toplama sıklığı ve saklama süresi ölçütlerine odaklanmıştır. Bulgular, GCP'nin ayrıntılı sanal makine metrikleri sunarak ek yapılandırmaya ihtiyaç duymadığını ve olay odaklı mimarilerde en yüksek mesaj işleme kapasitesine sahip olduğunu göstermektedir. Çalışma, güvenilirlik odaklı dağıtımlar için platformların izleme yeteneklerine ilişkin karşılaştırmalı içgörüler sunmaktadır.

Anahtar Kelimeler: İzleme Sistemleri, Site Güvenirliği Mühendisliği, Bulut Bilişim, Yazılım Mimarisi

INTRODUCTION

Since the early stages of software development, one of the most persistent challenges has been the efficient utilization of computational resources. The rise of distributed systems and, more recently, cloud computing has addressed this demand by offering scalable and portable infrastructure. Reports by the OECD confirm that many institutions are now shifting their operations toward cloud-based environments [1].

Ensuring system reliability has become a critical concern in these distributed environments. The concept of reliability in software engineering has been discussed for decades [2], [3], yet it was Google that formalized the practice through Site Reliability Engineering (SRE) [4]. SRE provides a framework for designing, deploying, and monitoring large-scale systems with an emphasis on stability and performance. It is built upon Service Level Indicators (SLIs), Service Level Objectives (SLOs), and Service Level Agreements (SLAs), three key

components that define measurable targets for reliable services.

In parallel, software architecture has evolved from monolithic structures to modular, distributed, and cloud-native designs. Building on Service-Oriented Architecture (SOA) principles [5], microservice architectures have become predominant due to their scalability, fault isolation, and flexibility [6], [7]. Among these, API Gateway (synchronous) and Event-Driven (asynchronous) patterns are widely adopted for managing communication in modern applications.

Within this context, the present study compares the native cloud monitoring tools of three leading providers—Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure—from an SRE perspective. A PaaS-based sample application was implemented using the Go (Golang) language, supported by relational databases and cloud-native messaging services.

The research focuses on evaluating the monitoring capabilities of each provider based on objective SRE-oriented metrics such as metric diversity, cost, alert latency, data collection frequency, and data retention period. The findings aim to provide actionable insights for reliability-centric deployments and to highlight how native observability mechanisms contribute to the operational goals of Site Reliability Engineering.

The following part of the study is presented as follows: In the second part, the previous studies are discussed. In the third part, the research method and materials are discussed. In the fourth part, the research findings are discussed. Finally, in the fifth part, the conclusion and recommendations are included.

RELATED WORKS

The evolution of monitoring in cloud computing has paralleled the increasing complexity and dynamism of distributed systems. Early work primarily focuses on establishing taxonomies and fundamental challenges. Aceto et al. [8], Ward and Barker [9] categorized monitoring tools based on architectural layers, data acquisition methods, and performance metrics, emphasizing that monitoring would remain a central concern in future cloud environments. Moreover, user trust and perceived reliability have long been recognized as essential factors in sustaining engagement within digital systems, as reflected in studies on customer loyalty in e-commerce [10].

Subsequent studies expanded this foundation by exploring reliability and automation through Site Reliability Engineering (SRE). Dave [11] connected SRE practices with operational efficiency and AIOps integration, while Kumar et al. [12] examined how monitoring architecture affects system performance. Fatema et al. [13] and Birje & Bulla [14] offered comprehensive classifications of commercial and open-source monitoring tools, such as Datadog, New Relic, CloudWatch, and Azure Monitor. These surveys improved the understanding of tool diversity but largely remained descriptive rather than comparative in terms of objective performance metrics.

Other works have provided provider-based comparisons. For example, Saraswat and Tripathi [15] and Noviani et al. [16] compared AWS, GCP, and Azure, but their evaluations were limited to general parameters such as latency, elasticity, or cost, without explicitly aligning them to SRE-driven indicators (SLIs, SLOs, SLAs). Similarly, Tamburri et al. [17] and Calero & Aguado [18] addressed observability challenges but did not consider how native metrics affect reliability quantification. From an architectural perspective, studies by Tanadechopon and Kasemsontitum [19] and Aksakalli et al. [6] highlighted the efficiency of microservice and event-driven designs. However, these works did not integrate cloud-native monitoring results into a unified reliability assessment framework.

In summary, most prior research has focused on categorization (what tools exist and what they measure) rather than evaluation (how effectively they measure reliability). Furthermore, no study to date has combined multi-provider deployment with microservice-level architectures while interpreting findings through the SRE analytical lens.

To address these gaps, the present study differentiates itself by integrating two fundamental communication patterns—API Gateway (synchronous) and Event-Driven (asynchronous)—into a unified evaluation framework across AWS, GCP, and Azure. It relies solely on native monitoring tools to maintain methodological parity among providers, avoiding any external instrumentation or third-party bias. The findings are interpreted through quantifiable SRE metrics such as alert latency, data collection frequency, and retention duration, thereby transforming monitoring results into measurable indicators of reliability. Through this analytical approach, the study establishes a concrete link between platform observability and SRE-driven performance evaluation, offering a novel benchmark for understanding how native monitoring ecosystems contribute to reliability engineering in multi-cloud environments.

METHODS and MATERIALS

In our study, two widely recognized microservice patterns from the literature, the API Gateway pattern and the Event-Driven pattern, were implemented across three major cloud computing providers: AWS, GCP and Microsoft Azure. These patterns were subjected to load using a dedicated load testing tool and custom scripts, simulating stress conditions. The metrics generated by the patterns underload on each provider were analyzed within the framework of SRE, and their implications are discussed in the following section of the study.

Table 1 presents the details of the implementations of the selected architectures across the three cloud service providers. It includes the deployment regions, the monitoring tools used, the messaging services employed, the databases selected, as well as the types of virtual machines and virtual networks utilized in each

environment. The geographic placement of infrastructure plays a critical role in the design of the systems, alongside the selection of cloud services. Variations in latency between regions used for comparison could lead to inconsistent or misleading results. To minimize this risk, Frankfurt—the closest common service region to Turkey supported by all three providers—was selected as the deployment location for all experiments.

Table 1. Infrastructure Preferences

Cloud Service Provider	Amazon (AWS)	Google (GCP)	Microsoft Azure
Region	Frankfurt	Frankfurt	Frankfurt
Tracing Tool	CloudWatch	GCP Monitoring	Azure Monitor
Messaging	SQS	PUB/SUB	Service Bus
Database	RDS for Postgresql	Cloud SQL for Postgresql	SQL Database for Postgresql
Virtual Machine	EC2	GCE	Azure VM
Virtual Network	AWS VPC	GCP VPC	Azure Virtual Network
Operating System	Ubuntu 22.04 LTS	Ubuntu 22.04 LTS	Ubuntu 22.04 LTS
CPU Count	1 VCPU	1 VCPU	1 VCPU
Memory	2 GB	2 GB	2 GB

As noted in prior research, monitoring is one of the most critical components for ensuring the reliability, resilience, and availability of increasingly complex and expanding cloud environments. In this study, the Platform as a Service (PaaS) monitoring solutions provided by the selected cloud platforms—AWS CloudWatch, GCP Cloud Monitoring, and Azure Monitor (Microsoft Azure)—were chosen for evaluation. These provider-specific tools were selected with the aim of enabling a comparative analysis, offering readers insights that can inform cloud service provider selection based on monitoring capabilities. In addition, cloud-native services such as databases and messaging tools, used in the experimental setup, are inherently designed and managed by the providers to ensure consistency and reliability. During experiments, sample web addresses are represented in the format X.X.X.X for abstraction. The X terms are shown as selected values from zero to 255. The database tables used in the tests are initially empty and are populated with data during the experiments.

Microservice-Based Communication Architectures

This study examines two fundamental communication patterns commonly used in such systems: (1) the synchronous API Gateway and (2) the asynchronous Event-Driven approach. Their implementation and behaviour across different cloud providers are analysed and assessed their architectural characteristics and performance implications.

The API Gateway functions as a centralized intermediary between clients such as web browsers or mobile applications and backend components. They also play a crucial role in microservices, serverless architecture. In cloud environments the API Gateway serves as a critical entry point for managing, routing, and securing requests. It receives incoming client requests, determines the appropriate backend service to handle each request, and manages the end-to-end flow of data. This may include request validation, routing, response aggregation. API Gateway reduces communication complexity, improves performance, and enables consistent policy enforcement across diverse services. As a result, it plays a vital role in streamlining interactions among distributed components in modern cloud-native applications.

As illustrated in Figure 1, the first architectural pattern (the API Gateway) implemented in this study. It relies on RESTful communication. The API Gateway supports four core use-case scenarios. In the user creation scenario, client requests are transmitted through the API Gateway to the user service, where the data is validated and then stored in the database. In the user retrieval scenario, the user service queries the database based on specific parameters and returns the results to the client via the gateway. For payment creation, the API Gateway routes transaction-related data to the payment service, which validates the input and persists the payment information in the database. Similarly, in the payment retrieval scenario, the system responds to client queries by fetching historical transaction data and delivering it back through the gateway.

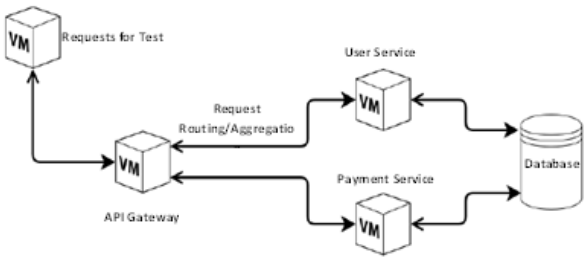


Figure 1. API Gateway Architectural View [20]

In contrast, the Event-Driven Pattern represents a microservice architecture in which components communicate by emitting and responding to events rather than making direct service calls. This asynchronous communication model promotes loose coupling between services, thereby enhancing scalability, flexibility, and modularity. In an event-driven architecture, four core elements are central to its operation. An event signifies a meaningful change or action within the system. A producer generates and emits these events, while one or more consumers subscribe to and act upon them. The event broker serves as an intermediary that routes, filters, and delivers events between producers and consumers. This structure not only decouples service interactions but

also enables scalable processing pipelines and supports advanced features such as event replay and fault tolerance.

In the implementation, all system endpoints documented and visualized using the Swagger tool in alignment with the OpenAPI specification. As shown in Figure 2, the system architecture consists of three core components which are named the Visual Service, the Messaging Service, and the Visual Processing Unit.

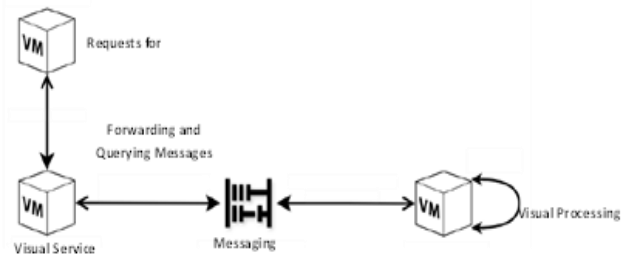


Figure 2. Event Driven Pattern Architectural View [20]

Visual Service receives incoming test requests and operates as the event producer. When it receives a request, it transmits the relevant data to the Messaging Service, which serves as the event broker. This intermediary is responsible for delivering the event to the Visual Processing Unit. The processing unit then executes the necessary computational logic, such as data transformation or analysis, and returns the result along the same communication pathway. This design illustrates the concrete application of event-driven architecture principles within a cloud-based microservice environment.

Load Testing and Monitoring

To evaluate system performance under realistic operational stress, load testing was performed using the Vegeta tool, executed on a dedicated virtual machine configured as a load generator. This testing environment was designed to replicate concurrent client activity and to measure how each cloud-native architecture—API Gateway and Event-Driven—responded under varying workloads.

The procedure followed a multi-threaded algorithm in Figure 3, inspired by the load generation methodology described in Koç [20]. In this algorithm, the total number of requests was divided among threads to enable controlled concurrency. Specifically, two primary threads were used to simulate parallel interactions: the first sent half of the total requests, while the second sent one-quarter, continuing until each completed its assigned task. This design provided a reproducible mechanism for generating distributed load and assessing system responsiveness under concurrent conditions.

To ensure progressive and measurable workload escalation, requests were applied in three successive stages of 100, 250, and 500 requests per second (RPS) over a continuous one-hour testing period. Each stage was executed sequentially, with the request rate gradually increased to simulate real-world load growth. The concurrency level—defined as the number of simultaneous request streams—was dynamically maintained by the thread scheduler to preserve fairness across endpoints.

```
1  Begin
2  Input the total number of requests (request_count)
3  Start Thread 1
4     Set i ← 0
5     while i ≤ (request_count / 2) do
6           Send request to server
7           i ← i + 1
8      End while
9  Start Thread 2
10    Set i ← 0
11    while i ≤ (request_count / 4) do
12          Send request to server
13          i ← i + 1
14     End while
15 Wait for both threads to complete
16 End
```

Figure 3 Load Test Algorithm

During execution, the Vegeta load testing tool collected detailed performance metrics, including latency (response time), throughput (requests per second), and error rate (ratio of failed to total requests). Across all test runs, the observed error rates showed significant variation among cloud providers — 27% for AWS, 88.6% for GCP, and 60.5% for Azure — reflecting different levels of reliability and responsiveness under load. Despite these differences, the experiments provided consistent insights into system behaviour under stress. In addition, network throughput, CPU and memory utilization, and database access rates were continuously monitored using the native cloud monitoring tools to correlate performance metrics with underlying resource consumption.

This structured testing approach ensured methodological consistency and reproducibility between architectural patterns and cloud providers. By explicitly defining the concurrency model, request rate, duration, and acceptable error threshold, the load testing setup aligns with SRE-driven evaluation principles and provides a reliable analytical foundation for interpreting the experimental results.

Objective Evaluation and Metrics

The performance metrics that were evaluated throughout the study are presented in this section. The metrics are organized into dedicated subsections based on key system components, which are virtual machines, databases, and messaging services. Each subsection includes a description of the relevant metrics along with the formulas used for their calculation.

To ensure a fair and meaningful comparison between the API Gateway and Event-Driven architectures, the evaluation criteria were defined based on principles derived from Site Reliability Engineering (SRE) and cloud-native observability. These criteria—CPU

utilization, memory usage, storage usage, network throughput, database operations, and messaging throughput—were selected because they directly correspond to the measurable indicators of service reliability, scalability, and performance efficiency in distributed systems.

In the context of SRE, CPU and memory utilization reflect system stability and resource efficiency under concurrent workloads, while storage usage and network throughput capture the scalability and responsiveness of cloud-based deployments. Similarly, database read/write operations and messaging throughput are central to evaluating how effectively each architecture manages inter-service communication and data persistence.

Unlike synthetic benchmarks that focus solely on latency or throughput, these metrics collectively represent the operational dimensions most affected by architectural design choices. The API Gateway pattern emphasizes synchronous request–response interactions, which are sensitive to CPU and network constraints, whereas the Event-Driven pattern relies on asynchronous message exchange, making messaging throughput and queue latency more critical.

By grounding the evaluation criteria in SRE-aligned reliability dimensions—availability, latency, throughput, and resource utilization—this study ensures that the comparison between architectural patterns reflects not only raw performance but also their operational resilience and scalability potential under realistic cloud workloads.

The formulations were derived through a combination of comprehensive literature review and an analysis of official documentation from major cloud service providers. However, given the proprietary nature of most cloud infrastructures and the limited availability of provider-defined formulas, many of the calculations had to be inferred indirectly. As a result, the metric definitions used in this study represent a synthesis of best practices and reference-based interpretations grounded in publicly available sources. To ensure consistency and comparability across platforms, the time resolution for all metrics was standardized to 1 second, in line with the monitoring granularity supported by native cloud monitoring tools. For virtual machines, key resource-level metrics such as CPU utilization, memory usage, storage usage, and network throughput were employed to evaluate system performance and operational load.

As seen from Equation 1, CPU Utilization (%) estimates the percentage of time the CPU is actively processing instructions rather than waiting on I/O. It is particularly important in identifying resource saturation under high concurrency [21].

$$\text{CPU Util.}(\%) = (1 - p^n) \times 100 \quad (1)$$

In Equation 1, p term represents the probability that a process is waiting for I/O operations, while n term denotes the number of processes currently residing in memory. CPU utilization quantifies the degree to which a virtual machine's processing resources are actively engaged during system operation. In this study, CPU usage was monitored under both the API Gateway and Event-Driven architectures in order to evaluate the management of computational load by each communication pattern. This metric was a key performance efficiency indicator and instrumental in identifying potential processing bottlenecks. It also contributes to broader assessments conducted within the SRE framework.

Memory utilization measures the proportion of a virtual machine's allocated memory that is actively in use (In Equation 2). It provides critical insight into system efficiency and resource management. Memory usage was observed across both architectural patterns under identical workload conditions, enabling a comparative analysis of memory demands. Fluctuations in memory utilization were analysed to assess each architecture's reliability and scalability.

$$\text{Memory Util.}(\%) = \left(\frac{\text{Used Memory}}{\text{Total Memory}}\right) \times 100 \quad (2)$$

Storage utilization refers to the percentage of allocated disk space that is actively in use during system operation, as seen in Equation 3. It was calculated as the ratio of used storage to the total available storage. This metric was used to evaluate how the API Gateway and Event-Driven architectures handle data persistence and storage efficiency under load conditions. The findings contributed to a broader assessment of each architecture's implications for scalability, resource management, and long-term system reliability within the context of SRE.

$$\text{Storage Util.}(\%) = \left(\frac{\text{Used Storage}}{\text{Total Storage}}\right) \times 100 \quad (3)$$

Incoming Data Rate and Outgoing Data Rate represent the volume of data the system receives and transmits per unit of time, respectively. Both metrics were calculated by dividing the total amount of data transferred by the duration of the observation period, as defined in Equations 4 and 5. These metrics were employed to evaluate the network throughput of each architectural pattern. Such measurements are essential for assessing system responsiveness and scalability in cloud-native environments.

$$\text{Incoming Data Rate} = \frac{\text{Total Incoming Data}}{\text{Elapsed Time}} \quad (4)$$

$$\text{Outgoing Data Rate} = \frac{\text{Total Outgoing Data}}{\text{Elapsed Time}} \quad (5)$$

In addition to virtual machine-level metrics, monitoring messaging services is crucial for understanding the behaviour of event-driven architectures and ensuring reliable message exchange between distributed components. Messaging systems are fundamental to decoupling microservices and enabling asynchronous

communication. Their performance must be evaluated using dedicated metrics.

The monitoring of messaging services focused on three core indicators, which are the number of messages sent, messages received, and the total number of messages processed. These metrics were chosen to reflect the throughput and responsiveness of the messaging infrastructure. They offer insight into how effectively each architectural model manages inter-service communication under load.

Three key throughput metrics were defined to assess messaging performance within the API Gateway and Event-Driven architectures. These are Message Sent per Unit Time, Message Received per Unit Time and Message Deletion per Unit Time, which are given in Equation 6,7 and 8, respectively. These metrics provide a detailed view of the operational efficiency of the communication layer and the intensity of message flow. The formulas used to calculate these metrics are presented below.

$$\text{Msg. Sent Rate} = \frac{\text{Total Num. of Sent Msg(s)}}{\text{Elapsed Time}} \quad (6)$$

$$\text{Msg. Rec. Rate} = \frac{\text{Total Num. of Received Msg(s)}}{\text{Elapsed Time}} \quad (7)$$

$$\text{Msg. Del. Rate} = \frac{\text{Total Num. of Deleted Msg(s)}}{\text{Elapsed Time}} \quad (8)$$

The formulations presented above enabled a quantitative evaluation of event flow throughput and served as an objective foundation for comparing the messaging load managed by the API Gateway and Event-Driven architectures under equivalent testing conditions.

In addition, monitoring database services is vital for assessing the responsiveness and operational demands placed on backend data layers within cloud-based systems. As shown in Equations 9 and 10, two key metrics were used in this context. They were the number of read operations and the number of write operations. These metrics reflect how frequently the database is accessed for querying or updating data by various application components. By analysing these metrics, the study aimed to gain insights into database usage intensity, typical access patterns, and potential performance bottlenecks associated with data retrieval and storage. The corresponding formulas used to calculate these metrics are presented below (Equations 9 and 10).

$$\text{Read Op. Rate} = \frac{\text{Total Num.of Reads}}{\text{Elapsed Time}} \quad (9)$$

$$\text{Write Op. Rate} = \frac{\text{Total Num. of Writes}}{\text{Elapsed Time}} \quad (10)$$

These metrics hold particular significance within the context of SRE, as they are directly linked to database performance, latency, and overall system reliability. Elevated read or write rates may signal intensive data access patterns, the need for query optimization, or early indications of database saturation. By continuous monitoring, it becomes possible to ensure that backend services remain responsive and efficient under varying load conditions.

EXPERIMENTAL RESULTS

This section presents the experimental results obtained from evaluating the two architectural patterns introduced earlier. API Gateway and Event-Driven architectures were deployed across three major cloud service providers and tested using the Vegeta load testing tool, following the procedure described in Figure 3. To simulate varying levels of system load, test requests were generated at increasing rates of 100, 250, and 500 requests per second, applied in successive intervals over a one-hour testing window. System performance under these conditions was assessed using the set of objective metrics defined in previous section, including CPU utilization, memory usage, network throughput, and service responsiveness. The results presented in this section provide a detailed view of how each architectural pattern behaves under different levels of operational stress.

Experimental Findings on the API Gateway Architecture

In this section, the performance and resource utilization of virtual machines and databases—as observed across the four key scenarios defined under the API Gateway pattern (user creation, user retrieval, payment creation, and payment retrieval)—will be presented and analysed in detail.

Table 2 compares CPU usage and data transfer across AWS, GCP, and Azure for four common API Gateway operations. AWS shows low CPU usage and moderate data rates, reflecting lightweight workloads. GCP has higher CPU usage but modest data flow, indicating more intensive processing. Azure stands out with very low CPU usage but the highest data throughput. Each provider demonstrates different strengths depending on the computer and network demands.

These findings reveal that variations in network throughput among providers mainly stem from how each platform allocates and manages virtual network resources. GCP's moderate data flow combined with higher CPU usage indicates a more computation-intensive handling of synchronous requests, likely due to tighter coupling between gateway routing and backend validation. In contrast, Azure's higher inbound and outbound rates reflect optimized bandwidth management and packet scheduling, enabling efficient handling of high-volume transactions. AWS exhibits a more conservative throughput pattern, which may result from its built-in throttling and load-balancing policies aimed at stabilizing latency during burst traffic.

Overall, these distinctions show that network performance depends not only on bandwidth capacity but also on how each provider orchestrates routing, caching, and load balancing, all of which directly influence service reliability and scalability under sustained workloads.

Table 2. Scenario-Based Resource CPU Utilization Metric for API Gateway Pattern Across Cloud Providers

Provider	Scenario	CPU Usage Avg (%)	Inbound Data Avg	Outbound Data Avg
AWS	User Creation	4.5	0.11 MiB/s	0.10 MiB/s
	User Retrieval	4.35	0.05 MiB/s	0.05 MiB/s
	Payment Creation	4.9	0.10 MiB/s	0.20 MiB/s
	Payment Retrieval	4.35	0.06 MiB/s	0.07 MiB/s
GCP	User Creation	22.5	3 KiB/s	6.85 KiB/s
	User Retrieval	24.6	2.7 KiB/s	7.5 KiB/s
	Payment Creation	18.5	2.5 KiB/s	7 KiB/s
	Payment Retrieval	26.5	4 KiB/s	11.5 KiB/s
Azure	User Creation	0.75	478 KiB/s	266 KiB/s
	User Retrieval	1.1	483 KiB/s	244 KiB/s
	Payment Creation	1	268 KiB/s	269 KiB/s
	Payment Retrieval	1	176 KiB/s	236 KiB/s

Table 3 highlights memory and storage usage across cloud providers for API Gateway scenarios. GCP is the only platform providing detailed metrics, with memory usage ranging from 13.5% to 19% and consistent storage usage at 20%, indicating strong default observability. AWS and Azure, in contrast, show no data for these metrics, likely due to the absence of built-in monitoring for memory and disk at the PaaS level. Despite this, Azure maintains the highest data throughput, suggesting efficient handling of network operations. Overall, GCP stands out for transparency in resource metrics, while AWS and Azure may require additional configuration to access similar insights.

These observations suggest that visibility gaps in AWS and Azure could influence how effectively performance anomalies are detected and resolved in production systems. Without default access to fine-grained memory or disk metrics, reliability engineers may depend more on external agents or aggregated dashboards, which could delay alerting or reduce the precision of root-cause analysis. Conversely, GCP's default exposure of such data reduces observability overhead and supports more immediate SRE-driven responses. Expanding native metric accessibility across providers would therefore enhance transparency and provide a more consistent reliability baseline in multi-cloud deployments.

Table 3 Scenario-Based Resource Memory and Storage Utilization Metrics for API Gateway Pattern for GCP

Provider	Scenario	Memory Usage Avg (%)	Storage Usage Avg (%)
GCP	User Creation	13.5	20
	User Retrieval	18.5	20
	Payment Creation	19	20
	Payment Retrieval	17	20

Experimental Findings on the Event-Driven Architecture

The Event-Driven Pattern (as illustrated in Figure 2) was deployed across the three major cloud platforms—AWS, GCP, and Microsoft Azure—to evaluate their performance in asynchronous communication environments. The architecture supported two primary operations: the generation of visual content triggered by event publication and the retrieval of that content by client consumers. Each operation followed a decoupled message flow mediated through cloud-native queuing or notification services, ensuring independence between producer and consumer components.

Evaluation primarily focused on latency performance and message throughput at the edge components under varying load intensities. The experiments were executed uniformly across all providers to maintain consistency and eliminate configuration-based bias. Metrics were collected from native monitoring tools during sustained load intervals, allowing observation of how each platform manages event queuing, message dispatching, and delivery confirmation, which directly influence end-to-end responsiveness.

The analysis aimed to determine how effectively each platform handles event-based workloads in terms of processing delay, queue depth, and transient congestion, thereby offering a clearer view of reliability and responsiveness from an SRE perspective. The results are summarized in Table 4, providing a comparative overview of message-handling efficiency across providers.

The results in the table reveal notable differences in messaging throughput under the two event-driven scenarios. GCP achieved the highest overall message-handling capacity, especially in the visual retrieval scenario, reaching up to 980 messages per second. This

outcome reflects GCP's shorter sampling intervals and optimized background message polling, which enhance real-time delivery responsiveness. In comparison, AWS showed steady but slightly lower throughput due to conservative queue management and default batching strategies, while Azure maintained intermediate performance, benefiting from efficient message broker integration but exhibiting marginally higher latency under heavy concurrency.

Table 4 Maximum Messaging Throughput by Scenario and Cloud Provider in the Event-Driven Pattern

Provider	Scenario	Max Messages Received (/s)	Max Messages Sent (/s)	Max Messages Deletion (/s)
AWS	Visual Processing	333	500	329
	Visual Retrieval	371	371	371
GCP	Visual Processing	500	276	500
	Visual Retrieval	980	495	976
Azure	Visual Processing	190	550	197
	Visual Retrieval	200	557	200

AWS demonstrated relatively balanced performance, while Azure maintained stable but comparatively lower messaging rates. These variations highlight how messaging infrastructure efficiency and throughput can differ significantly across platforms, which is a critical consideration for the reliability and scalability of event-driven architectures within the SRE framework.

CONCLUSION

This study presented a comparative analysis of the native monitoring capabilities offered by three major cloud service providers—Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure—within the context of Site Reliability Engineering (SRE). To facilitate this comparison, two widely adopted microservice communication patterns, the API Gateway and the Event-Driven architectures, were implemented across all three platforms. Their performance was then evaluated using objective metrics collected exclusively from provider-native monitoring tools, ensuring a consistent and unbiased assessment framework.

The results demonstrated that the number, granularity, and coverage of native metrics vary significantly among providers. From an SRE standpoint, these differences directly influence the ability to define meaningful SLIs, establish measurable SLOs, and enforce effective SLAs. Providers offering broader metric coverage and shorter sampling intervals inherently enable more accurate observability and faster incident detection—both of which are central to SRE-driven operational resilience.

In this context, GCP emerged as the most advantageous platform, outperforming AWS and Azure across multiple technical dimensions that directly affect observability and reliability analysis. One of the key differentiators is GCP's shorter default sampling interval compared to AWS. Its native monitoring service (Cloud Monitoring) uses a one-minute metric resolution by default, whereas AWS CloudWatch typically operates at five-minute intervals; Azure Monitor is also one minute for equivalent data. This finer granularity allows GCP to detect short-lived performance anomalies—such as transient latency spikes or CPU saturation—that would otherwise be invisible in coarser sampling windows.

Additionally, GCP benefits from native agent integration that automatically discovers and registers monitored resources without manual configuration. This "agentless-by-default" design not only simplifies deployment but also ensures uniform metric availability across compute, messaging, and database layers. In contrast, AWS and Azure often rely on optional agents or user-defined instrumentation to achieve comparable metric depth. GCP further distinguishes itself by offering comprehensive metric coverage and native PromQL-based querying and alerting, providing a more flexible and fine-grained environment for SRE practitioners.

While AWS remains advantageous in terms of customization and Azure provides strong interoperability with external observability suites, GCP's combination of finer sampling intervals, richer native metric scope, and automated monitoring integration yields a tangible operational advantage. These technical characteristics collectively explain its superior performance in this study and align closely with the SRE philosophy of maximizing visibility, reducing mean time to detection (MTTD), and enhancing reliability through proactive monitoring.

Overall, this research highlights that cloud-native monitoring depth is not uniform across providers, and that even subtle differences in data collection and instrumentation significantly influence reliability evaluations. Future work may expand this study by incorporating hybrid or multi-cloud orchestration environments, integrating advanced alert correlation mechanisms, and exploring the application of machine learning to predict system reliability trends based on native monitoring data.

REFERENCES

[1] Ker, D. (2021). Measuring cloud services use by businesses (OECD Digital Economy Papers No. 304). OECD Publishing. https://doi.org/10.1787/71a0eb69-en

[2] Musa, J. D. (1980). The measurement and management of software reliability. Proceedings of the IEEE, 68(9), 1131–1143. https://doi.org/10.1109/PROC.1980.11812

[3] Musa, J. D., & Everett, W. W. (1990). Software-reliability engineering: Technology for the 1990s. IEEE Software, 7(6), 36–43. https://doi.org/10.1109/52.60588

[4] Google. (n.d.). What is Site Reliability Engineering (SRE)? https://sre.google

[5] Niknejad, N., Ismail, W., Ghani, I., Nazari, B., & Bahari, M. (2020). Understanding service-oriented architecture (SOA): A systematic literature review and directions for further investigation. Information Systems, 91, 101491. https://doi.org/10.1016/j.is.2019.101491

[6] İ. K. Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdogan, "Deployment and communication patterns in microservice architectures: A systematic literature review," Journal of Systems and Software, vol. 180, p. 111014, 2021. doi: 10.1016/j.jss.2021.111014

[7] Jhingran, S., & Rakesh, N. (2023). Performance analysis of microservices behavior in cloud vs containerized domain based on CPU utilization. International Journal of Recent Innovations and Trends in Computing and Communication, 11, 509–516. https://doi.org/10.17762/ijritcc.v11i6s.6959

[8] Aceto, G., Botta, A., Donato, W., & Pescapè, A. (2013). Cloud monitoring: A survey. Computer Networks, 57, 2093–2115. https://doi.org/10.1016/j.comnet.2013.04.001

[9] Ward, J. & Barker, A. (2014). Observing the clouds: A survey and taxonomy of cloud monitoring. Journal of Cloud Computing, 3, Article 24. https://doi.org/10.1186/s13677-014-0024-2

[10] Gefen, D. (2002). Customer loyalty in e-commerce. Journal of the Association for Information Systems, 3, 27–51. https://doi.org/10.17705/1jais.00022

[11] Dave, D. M. (2023). Impact of site reliability engineering on manufacturing operations: Improving efficiency and reducing downtime. International Journal of Scientific and Research Publications, 10, Article 14312.
https://doi.org/10.29322/IJSRP.13.11.2023.p14312

[12] Kumar, P., Singh, P., Chopra, S., Sarna, J. S., & Rawat, K. (2017). Inspection of cloud computing monitoring tools. In Proceedings of the International Conference on Infocom Technologies and Unmanned Systems (ICTUS) (pp. 361–365). IEEE. https://doi.org/10.1109/ICTUS.2017.8286033

[13] Fatema, K., Emeakaroha, V. C., Healy, P. D., Morrison, J. P., & Lynn, T. (2014). A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. Journal of Parallel and Distributed Computing, 74, 2918–2933. https://doi.org/10.1016/j.jpdc.2014.06.007

[14] Birje, M. N., & Bulla, C. (2019). Commercial and open source cloud monitoring tools: A review. In M. S. Obaidat & I. Awan (Eds.), Proceedings of the International Conference on E-Business and Telecommunications (pp. 480–490). Springer. https://doi.org/10.1007/978-3-030-24322-7_59

[15] Saraswat, M., & Tripathi, R. C. (2020). Cloud computing: Comparison and analysis of cloud service providers—AWS, Microsoft, and Google. In Proceedings of the 9th International Conference on System Modeling & Advancement in Research Trends (SMART) (pp. 281–285). IEEE.

[16] Noviani, E. F., Kembara, B., Pratama, B. A. Y., Sari, D. A. P., Shiddiqi, A. M., & Santoso, B. J. (2022). Performance analysis of AWS and GCP cloud providers. In Proceedings of the IEEE International Conference on Cybernetics and Computational Intelligence (pp. 236–241). IEEE. https://doi.org/10.1109/CyberneticsCom55287.2022.9865484

[17] Tamburri, D., Miglierina, M., & Di Nitto, E. (2020). Cloud applications monitoring: An industrial study. Information and Software Technology, 127, 106376. https://doi.org/10.1016/j.infsof.2020.106376

[18] Calero, J. M. A., & Aguado, J. G. (2015). Comparative analysis of architectures for monitoring cloud computing infrastructures. Future Generation Computer Systems, 47, 16–30.

[19] Tanadechopon, T., & Kasemsontitum, B. (2023). Performance evaluation of programming languages as API services for cloud environments: A comparative study of PHP, Python, Node.js, and Golang. In Proceedings of the 7th International Conference on Information Technology (InCIT) (pp. 293–297). IEEE. https://doi.org/10.1109/InCIT60207.2023.10413079

[20] Koç, C. (2025). The investigation of native cloud monitoring services with the use of the site reliability engineering approach (Master's thesis, Çanakkale Onsekiz Mart University, School of Graduate Studies, Department of Computer Engineering). Çanakkale Onsekiz Mart University.

[21] Tanenbaum, A. S., & Bos, H. J. (2014). Modern operating systems (4th ed.). Pearson.