Alqalam Journal of Medical and Applied Sciences. 2026;9(2):493-497
https://doi.org/10.54361/ajmas.269228

Original article

Experimental Analysis of Performance and Resource Stability in Cloud-Native Microservices

Abubkr Abdelsadiq*, Gharsa Elmaresh, Salem Abdusalam

Department of Computer Science, Faculty of Science, University of Elmergib, Alkhoms, Libya
Corresponding email. *abubkr.abdelsadiq@elmergib.edu.ly*

Abstract
The widespread adoption of microservices architecture has enabled modern technology organizations to achieve greater system flexibility and scalability. However, this shift introduces significant challenges in managing performance and resource stability due to the system's inherently distributed nature. In this study, we evaluate the dynamic behavior of a microservices system, using the Google Online Boutique project as a representative case. Our methodology involved a series of systematic tests, including baseline, stress, and long-duration soak tests, executed within a Docker container environment. We monitored performance using tools such as JMeter and Docker stats. Our experiments identified a maximum system throughput of 115.5 requests per second. Crucially, we observed that exceeding this threshold triggered a nonlinear degradation in response time, which escalated to 2016ms. The analysis pinpointed a computational bottleneck within the Currency Service. Furthermore, the soak test revealed a 12% increase in memory consumption over time, indicating a potential resource leak. These findings underscore the critical importance of Horizontal Scaling strategies and proactive resource management in cloud-native environments.
Keywords. Microservices Architecture, System Flexibility, Cloud-Native Environment, Baseline Testing.

Introduction

The evolution of software engineering has made the transition from monolithic systems to a microservices architecture a technical imperative for organizations needing to support rapid development cycles [1]. This paradigm decomposes large applications into smaller, independent services that communicate via lightweight protocols such as HTTP or gRPC. This approach is intrinsically linked to the rise of cloud-native development, where systems are designed from the ground up to exploit the full potential of cloud computing platforms [2]. This migration is driven by well-documented advantages, including enhanced modularity, improved fault tolerance, and superior scalability [3]. A critical benefit is the independent deployability of services, which grants development teams greater autonomy and enables the faster iteration and continuous delivery required in modern agile environments [4]. Yet, this architectural shift brings with it new complexities, particularly in performance testing and resource management within highly dynamic cloud infrastructures [5].

Despite their numerous advantages, microservices significantly increase the "surface area of interaction" between components, complicating the diagnosis of performance issues. A delay in a single service can propagate, leading to a backlog of requests in other services and potentially causing Cascading Failure [6]. The core research problem addressed is identifying the "breaking point" for microservices when deployed under limited-resource constraints and the subsequent implications for load-distribution strategies. This challenge is compounded by the need for effective observability tools to explain such failures [7].

The shift to microservices has generated extensive research on performance and reliability. Early work by Newman [1] established the foundational principles of microservice design. Subsequent studies have focused on the practical challenges of performance testing in distributed environments. Eismann [5] highlighted that the dynamic and unstable nature of cloud environments makes the performance testing of microservices non-trivial, a finding that directly motivates the controlled experimental approach used in this study. Research on resource management in microservices is also critical. Luo [8] proposed an efficient resource management system to guarantee Service Level Agreements (SLAs) in shared microservice environments, emphasizing the need for effective resource quotas. Furthermore, the problem of Cascading Failures, where a failure in one service propagates to others, has been a major focus [6]. Sharma [7] and Soldani [6] explored the use of advanced observability tools, such as eBPF and log analysis, to identify the root causes of these failures, underscoring the complexity of debugging in distributed systems. Our work extends this research by providing an empirical analysis of resource stability and performance degradation under controlled stress, specifically identifying the nonlinear degradation point and the impact of a memory leak on the resource consumption of a single service.

The primary objectives of this study were to measure system throughput and latency (response time) under varying load conditions, to identify the most resource-intensive services in terms of CPU and RAM usage and classify them as potential bottlenecks, and to evaluate the reliability and stability of the system when subjected to continuous load over extended periods. Together, these objectives provided a comprehensive framework for assessing both the performance efficiency and operational resilience of the system in diverse workload scenarios.

Received*: 21-12-2025 -* ***Accepted****: 18-02-2026 -* ***Published****: 24-02-2026*

https://doi.org/10.54361/ajmas.269228

Methodology

Technical Environment

All experiments were conducted on a portable workstation to ensure stability and consistency of the physical resources. The documented specifications were as shown in table 1.

Table 1. Technical Specifications of the Experimental Workstation

Component	Specification
Device	HP ProBook 450 G3
Processor	Intel Core i7 (6th Gen) @ 2.5 GHz
RAM	8 GB DDR4
Virtual Environment	Docker Desktop with WSL2 (Windows Subsystem for Linux)

Note: The WSL2 engine was used to ensure that the performance characteristics closely resembled those of real production environments.

System Architecture

The Google Online Boutique project, an e-commerce demonstration application, was selected as the System Under Test (SUT). It comprises multiple services written in various languages (Go, Node.js, and Python). Eight core services were deployed: Frontend, Catalog, Currency, Cart, Redis, Checkout, Shipping, and Payment. To maintain experimental focus and resource stability on the host machine, Service Stubbing was employed to redirect secondary services (e.g., emails and advertisements), thereby reducing the extraneous load.

Test Protocol

Apache JMeter [9] was employed to simulate user load through a series of structured scenarios designed to evaluate system performance under varying conditions. In the baseline load scenario, ten simulated users were introduced to establish performance metrics under normal operating conditions. For the stress load scenario, the number of users was gradually increased to 50, 150, and 250 in order to monitor system behavior and identify the saturation point at which performance degradation became evident. Finally, a soak test was conducted by maintaining a constant load of 80 users for a duration of 20 minutes, allowing assessment of long-term memory stability and system reliability under sustained demand.

Results

Request Performance Analysis

The results demonstrate a distinct variation in the system behavior corresponding to an increase in the number of concurrent users. Performance testing in cloud-native environments is crucial for evaluating responsiveness and scalability [5]. The analysis indicated that the system reached the saturation phase at 150 users. The subsequent load increase did not yield a higher throughput; instead, it led to request accumulation in waiting queues (queuing delay), resulting in a significant, non-linear degradation of the response time. This behavior is consistent with the principles of queuing theory, where the system utilization approaches 100%, causing the waiting time to increase exponentially. The nonlinear response time degradation (from 1118ms to 2016ms) clearly demonstrates the system's transition from a stable, underutilized state to a resource-contended, saturated state, which is a critical finding for capacity planning. As shown in Table 2, throughput increases steadily across load levels, peaking at 115.5 requests/second at 150 concurrent users, before declining slightly to 113.4 at 250 users. Simultaneously, the latency (response time) begins its sharp, exponential ascent at this peak point. The latency increase, which reached 2016ms, marks the transition from stable performance to the saturation phase, where the processor is unable to process incoming requests at the rate of arrival.

Table 2. System Performance Metrics Under Varying Load Conditions

Concurrent Users	Throughput (req/sec)	Latency (ms)	Observation
10	83.4	117	Excellent performance (Baseline)
50	102.7	384	Moderate load; throughput increasing steadily
150	115.5 (Peak)	1118	Saturation point reached
250	113.4 (Decrease)	2016	Non-linear degradation

Resource Profiling

Resource consumption was monitored using the Docker Stats tool during the peak-load scenario (250 users). As shown in Table 3, the Currency Service is clearly the most resource-intensive component at 63%, followed by the Frontend Service at 48%. This distribution confirms the hypothesis that the computational operations within the Currency Service are the primary performance drivers, making it the prime candidate for scaling to ensure system stability under stress.

Received*: 21-12-2025 -* ***Accepted****: 18-02-2026 -* ***Published****: 24-02-2026*

Table 3. CPU Resource Consumption per Service at Peak Load (250 Users)

Service	CPU Consumption (%)	Primary Cause
Currency Service	63%	Intensive computational operations for currency conversion.
Frontend Service	48%	Processing HTTP traffic and merging data from other services.
Catalog Service	8.3%	Stable, indicating efficient data retrieval.

Stability Analysis

During the 20-minute soak test, the system maintained operational stability with an Error Rate of 0.00%**.** However, a steady, linear growth in memory consumption was observed in the Currency Service, increasing from 56.06 MiB to 62.8 MiB, a 12% increase over a short duration. This growth pattern suggests a potential limitation in memory management within the Node.js container, which may necessitate strategies such as periodic restarts or code-level optimization to mitigate the long-term resource exhaustion. This finding is particularly relevant to the operational stability of cloud-native applications, as even a small persistent memory leak can lead to resource exhaustion and eventual service failure in long-running environments.

Figure 1. below shows the results of the continuous stability test. Despite the stable user load of 80, the memory consumption exhibits a linear upward trend. This pattern indicates the presence of unstable memory behavior, potentially a Memory Leak, within the container. This is a crucial finding that strongly recommends improving the service lifecycle management in long-running environments.

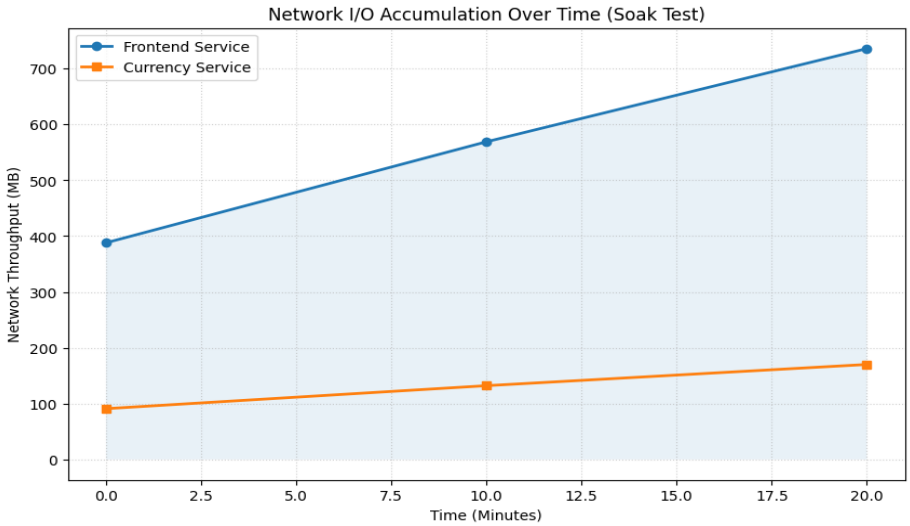

Figure 1. Memory usage trend for the Currency Service during a 20-minute soak test

Discussion

The experimental results reveal several important insights into the performance and resource behavior of cloud-native microservices under varying load conditions. The identification of a clear saturation threshold at 150 concurrent users, corresponding to a peak throughput of 115.5 req/sec, is consistent with the principles of queuing theory and prior empirical work on microservice performance boundaries [5]. Once the system exceeded this saturation point, throughput ceased to grow meaningfully while response latency escalated nonlinearly from 1118ms to 2016ms, indicating a transition from a linear to an exponential performance degradation regime. This behavior underscores the critical importance of proactive capacity planning and the early identification of saturation thresholds in cloud-native deployments.

The resource profiling results (Table 3) highlight a pronounced imbalance in CPU utilization across services. The Currency Service consumed 63% of CPU at peak load due to its computationally intensive floating-point operations for currency conversion, far exceeding all other services. This finding aligns with known challenges in stateless computation-heavy microservices and suggests that the system-level bottleneck is localized rather than distributed. The Frontend Service, at 48%, acts as an aggregation layer and represents a secondary pressure point. These observations are consistent with findings by Luo [8], who emphasized the need for service-level resource quotas to prevent single services from monopolizing shared infrastructure. The highly uneven resource distribution observed in this study further reinforces the argument that horizontal autoscaling policies should be service-specific rather than cluster-wide.

The soak test results reveal a steady, linear 12% increase in memory consumption within the Currency Service over a 20-minute observation window, despite maintaining a constant load of 80 users and an error rate of 0.00%. This pattern is characteristic of a managed memory leak, possibly attributable to the Node.js garbage collection cycle failing to reclaim short-lived objects generated during currency conversion loops.

Received*: 21-12-2025 -* ***Accepted****: 18-02-2026 -* ***Published****: 24-02-2026*

Such a leak, if left unaddressed, would eventually lead to out-of-memory errors and service failure in long-duration production environments. This finding is particularly significant because the system appeared stable at the application level (zero errors) while degrading internally at the resource level, a scenario that is difficult to detect without continuous memory monitoring. This reinforces the importance of implementing observability tools at the container level, as highlighted by Sharma [7], to detect resource-level anomalies that are invisible to conventional application-layer monitoring.

Compared to prior work, the controlled experimental methodology adopted in this study fills an important gap. Eismann [5] noted the inherent difficulty of microservice performance testing in real cloud environments due to resource variability; by using a fixed local hardware configuration with Docker and WSL2, this study provides reproducible and interpretable results that are free from external interference. While this limits direct generalizability to production cloud clusters, it enables precise characterization of service-level behavior under defined conditions. Future studies should replicate these experiments in orchestrated environments such as Kubernetes to validate the findings at scale and explore the effectiveness of dynamic resource management strategies such as Horizontal Pod Autoscaling (HPA) and service mesh policies [10].

Recommendations

Based on the experimental findings, several technical recommendations can be proposed to enhance system performance and reliability. First, horizontal autoscaling should be implemented to automatically duplicate instances of the Currency Service when CPU consumption exceeds a threshold of 50%. Automated scaling mechanisms, such as Horizontal Pod Autoscalers (HPAs), are particularly effective in microservice architectures for dynamic resource allocation [10]. Second, caching solutions such as Redis can be employed to store the results of frequent currency conversion operations, thereby reducing computational overhead and improving response times. Third, resource quotas should be established to define maximum memory limits for containers, ensuring that no single service can exhaust the host machine's resources. This practice is essential for optimizing resource management in shared microservice environments [8].

Future Work

The empirical results of this study open several avenues for future research in the following ways. First, the identified memory leak in the Node. JS-based Currency Service warrants a deeper investigation into language-specific memory management within containerized environments. Second, future work should focus on implementing and testing advanced resource management strategies, such as Service Mesh technologies (e.g., Istio or Linkerd), to evaluate their effectiveness in mitigating nonlinear performance degradation and managing resource contention. Finally, exploring the application of Machine Learning (ML) models for predictive autoscaling and QoS-aware resource management, as suggested by Sinan [11], could provide a more proactive approach to maintaining stability in highly dynamic microservice architectures.

Conclusion

This study confirms that although the microservices architecture successfully isolates tasks, it concurrently introduces specific stress points or bottlenecks. The Currency Service was identified as the dominant factor driving CPU consumption and response time degradation, and was severely impacted once the system's processor reached its saturation point.

Conflict of interest. Nil

References

1. Newman S. Building Microservices: Designing Fine-Grained Systems. Sebastopol (CA): O'Reilly Media; 2021.
2. Oyeniran OC, Adewusi AO, Adeleke AG. Microservices architecture in cloud-native applications: design patterns and scalability. Int J Adv Res Sci Eng Technol. 2024;11(1):1-10.
3. Agrawal P. Microservices architecture: a modern approach to cloud-native development. J Comput Sci Technol Stud. 2025;7(1):1-12.
4. RSIS International. Microservices architecture in cloud computing: a software engineering perspective on design, deployment, and management. Int J Res Innov Soc Sci. 2025;9(3):1-10.
5. Eismann S. Microservices: a performance tester's dream or nightmare? In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE); 2020 Apr 20-24; Edmonton, Canada. New York: ACM; 2020.
6. Soldani J, Forti S, Roveroni L. Explaining microservices' cascading failures from their logs. Softw Pract Exp. 2025.
7. Sharma B. Improving microservices observability in cloud-native infrastructure using eBPF [thesis]. West Lafayette (IN): Purdue University; 2023.
8. Luo S. Optimizing resource management for shared microservices. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC); 2024 Nov 15-18; Redmond, WA, USA. New York: ACM; 2024.
9. Apache Software Foundation. Apache JMeter user manual [Internet]. 2025 [cited 2026 Feb 18]. Available from: https://jmeter.apache.org/usermanual/
10. Ahmad H. Resilient auto-scaling of microservice architectures with efficient resource management. arXiv 2501.12345 [Preprint]. 2025 [cited 2026 Feb 18]. Available from: https://arxiv.org/abs/2501.12345

Received: *21-12-2025* - ***Accepted***: *18-02-2026* - ***Published***: *24-02-2026*

11. Sinan M. ML-based and QoS-aware resource management for cloud microservices. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC); 2021 Nov 1-4; Seattle, WA, USA. New York: ACM; 2021.
12. Google Cloud Platform. Microservices Demo: Online Boutique [Internet]. 2020 [cited 2026 Feb 18]. Available from: https://github.com/GoogleCloudPlatform/microservices-demo
13. Merkel D. Docker: lightweight Linux containers for consistent development. Linux J. 2014.
14. Weyuker EJ. Experience with performance testing of software systems: issues, an approach, and case study. IEEE Softw. 2002.

Received*: 21-12-2025 -* ***Accepted****: 18-02-2026 -* ***Published****: 24-02-2026*