Performance Evaluation of Serverless Computing Infrastructure: Insights from Open-Source Frameworks

Antonio Sousa, David Beserra and Jean Araujo

Abstract Serverless computing has gained widespread adoption due to its simplified management and lightweight design, particularly when paired with container orchestration systems like Kubernetes. By enabling developers to focus on application logic without managing underlying infrastructure, serverless computing offers advantages such as runtime-based billing in millisecond units, reducing operational costs and appealing to enterprises. This study evaluates resource utilization across 24 combinations of Ubuntu and Debian operating systems with Docker and Podman container platforms under varied workloads. Results indicate that Ubuntu with Docker achieves superior efficiency in CPU and RAM usage compared to other configurations. This experimental analysis provides practical insights into hardware resource management for serverless deployments and highlights opportunities for improving infrastructure in diverse scenarios.

1 Introduction

Monolithic software architecture consolidates all functions into a single module or application, placing all system functionalities in the same environment. This design faces challenges in scalability, maintenance, and updates, often requiring specialized teams for both application and hardware maintenance. Such constraints in-

Antonio Sousa
Departamento de Computação, Universidade Federal de Sergipe, São Cristóvão, Brazil, e-mail: antoniocar@academico.ufs.br

David Beserra
École Pour l'Informatique et les Technologies Avancées (EPITA), Paris, France, e-mail: david.beserra@epita.fr

Jean Araujo
Instituto de Telecomunicações, Universidade de Aveiro, Aveiro, Portugal, e-mail: jean.araujo@ua.pt

crease system complexity, compromise stability, and inflate operational costs. To overcome these challenges, companies like Amazon, Netflix, and Uber have transitioned to microservices architecture [6], where small, independent applications perform specific tasks. This ecosystem of modular services improves scalability, resource utilization, and adaptability, enabling better allocation of physical and human resources.

Building on the advantages of microservices and the emergence of container orchestration technologies, serverless computing has gained traction for its simplicity and resource efficiency. Serverless computing allows users to write functions in any language while the underlying server management is handled by cloud service providers [7]. This model reduces developer overhead and introduces cost-efficient billing based on resource usage, making it increasingly attractive for enterprise applications.

Serverless computing is categorized into Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS) models [8]. BaaS delivers services via APIs, while FaaS provides stateless and event-driven functions. Applications range from machine learning [15] to IoT [16] and big data analysis [2]. By 2025, an estimated 50% of global enterprises will adopt serverless computing, leveraging advancements in microservices and cloud computing [11].

Despite its benefits, serverless computing presents challenges for cloud providers, including resource provisioning, latency management, and maintaining cost-efficiency. Issues such as high cold start latency, communication overhead, and monitoring inefficiencies have been identified in Function-as-a-Service (FaaS) platforms [12, 5]. To address these issues, providers must optimize factors such as latency, throughput, and resource overhead [12, 5, 1].

This study evaluates open-source solutions for serverless application infrastructure, focusing on hardware resource utilization under various scenarios. The findings aim to provide actionable insights into improving the performance of serverless deployments.

The remainder of this paper is structured as follows: Section 2 reviews related work, summarizing previous research on serverless computing infrastructures. Section 3 describes the methodology employed in this study. Section 4 details the experimental design, including test environments and scenarios analyzed. Section 5 presents and evaluates the results. Finally, Section 6 concludes with a summary of findings and future research directions.

2 Related Works

Several studies address the challenges and benefits of serverless computing but lack practical and reproducible insights into open-source frameworks. This work fills that gap, evaluating resource utilization and performance across multiple open-source serverless solutions.

Li et al. [7] review resource management techniques and future directions such as edge integration but omit practical evaluations using open-source frameworks. Our study addresses this through empirical testing with tools like OpenFaaS and Kubeless.

Hassan et al. [3] highlight serverless benefits, including cost savings and scalability, but rely on theoretical analysis. Our research complements this by implementing and evaluating practical serverless infrastructure setups.

Scheuner et al. [13] analyze Function-as-a-Service (FaaS) performance in private clouds, focusing on cold start latency and resource scaling. However, they overlook open-source alternatives, which our study explores through tools like OpenWhisk and Knative.

Liu et al. [9] improve serverless function performance through cold start latency reduction but concentrate on function-level optimization. Our research shifts the focus to infrastructure-level evaluations with open-source tools.

Lin et al. [8] model serverless workflows to balance cost and performance, yet limit their scope to private clouds. Our study broadens this by testing multiple platforms and providing insights into diverse configurations.

Wen et al. [14] introduce Super Flow for serverless testing but focus narrowly on specific setups. In contrast, our study compares multiple open-source alternatives, offering a comprehensive view of serverless performance across different scenarios.

In summary, while these studies underscore the importance of serverless performance and cost-efficiency, most lack practical, open-source evaluations. Our research bridges this gap, offering reproducible insights and actionable guidelines for deploying serverless solutions with open-source tools.

3 Research Goals and Methodology

This study aims to achieve the following goals:

- Analyze the performance characteristics of different serverless infrastructure configurations under varying workloads;
- Evaluate the impact of container orchestrators, operating systems, and HTTP methods on resource utilization and response times;
- Develop a set of guidelines for configuring optimal serverless computing environments.

In order to do that we used an adapted version of the methodology proposed in [10], structured to systematically execute and analyze experiments on serverless computing infrastructure. The methodology is organized into three main phases: **Literature Review**, **Pre-Evaluation**, and **Assessment**, each comprising defined activities that align with the research goals.

Figure 1 presents the methodology flowchart, illustrating the sequence of steps and decisions. Rectangles represent process steps, diamonds indicate branching paths, and dashed rectangles detail associated actions such as parameter definition

and testing. The methodology ensures coherence and reproducibility, enabling structured execution and robust analysis.

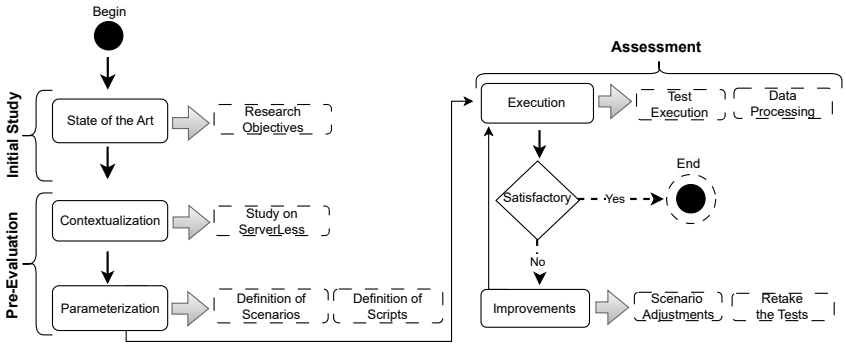
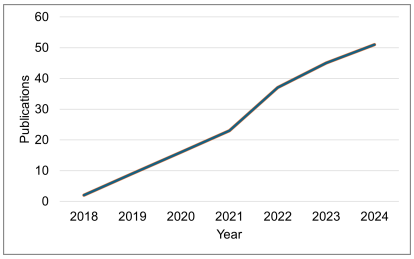
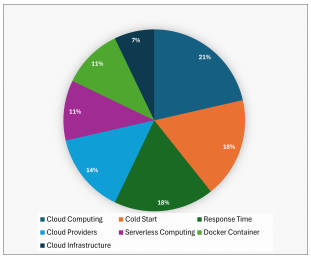


Fig. 1 Methodology

The **Literature Review** phase establishes the foundation for the research by analyzing recent scientific contributions in the field of serverless computing. A total of 187 articles were initially identified (see Figure 2). After applying inclusion and exclusion criteria, 23 articles remained, with a final selection of 16 articles deemed relevant based on a quality cutoff score of 2.5. This phase focused on identifying key concepts, tools, and performance issues related to serverless computing. Key findings highlight a growing interest in serverless computing, particularly in addressing performance issues such as response time and cold starts. Additionally, the predominance of topics such as Cloud Computing (21%), Cold Start (18%), and Response Time (18%) reflects the research community's emphasis on performance optimization.



(a) Publications by year



(b) Publications by subject

Fig. 2 Literature Review

The **Pre-Evaluation** phase defined the experimental setup and tools to be used, encompassing the contextualization and parameterization of the study. The infras-

tructure for experiments was configured on a cloud server with specifications including 4 virtual CPUs, 8 GB RAM, and 200 GB of disk space. The server utilized Ubuntu Server 22.04.3 LTS and Debian 12.5 as the operating systems, while Podman v3.4.4 and Docker v2.27.0 were employed as container orchestrators. Additionally, Kubernetes v1.28 and Knative v1.12.3 were used as serverless frameworks, ensuring compatibility and reliability for benchmarking containerized workloads. Experimental scenarios were parameterized with workloads categorized as low, medium, and high, simulating up to 100, 500, and 1000 simultaneous users, respectively. Resource usage metrics such as CPU, disk I/O, network, and response time were monitored using Linux-based tools including `pidstat`, `pidof`, and `/proc/meminfo`. Each experiment lasted 60 minutes and tested combinations of operating systems, orchestrators, HTTP methods, and workloads. These scenarios were designed to provide insights into optimizing server infrastructure to enhance user experience and avoid service interruptions.

The **Assessment** phase encompassed the execution of experiments, data processing, and iterative refinement of the setup. During the test execution stage, scripts were executed for each scenario, generating data on resource utilization and performance metrics. This stage was carefully controlled to ensure consistent and comprehensive data collection. The data processing stage involved exporting collected data for visual representation using Gnuplot, along with the calculation of statistical measures such as standard deviation and confidence intervals to validate the results. The refinement phase focused on making iterative adjustments to experimental parameters, scripts, or hardware configurations, aiming to enhance the study's accuracy and alignment with its objectives without compromising reproducibility, cost-efficiency, or time constraints.

With the parameters defined, the next step is to develop the scripts that will be used to obtain the experiment data and execute stress tools. Two scripts were used, one to generate stress load and another to collect data. Algorithm 1 demonstrates the hardware resource data capture script, and Algorithm 2 demonstrates the workload generation script.

4 Experimental Design

This study aims to evaluate the resource utilization of serverless applications under various configurations to propose infrastructure setups that maximize resource efficiency while maintaining performance. A total of 24 scenarios were tested, combining different operating systems, container orchestrators, and workloads. The experiments focused on key metrics such as RAM, CPU, Disk I/O, and Network usage to identify configurations that balance performance and efficiency effectively.

The testing environment was hosted by the public cloud provider Infonet, which utilizes VMware virtualization technology. A virtual machine (VM) with 4 vCPUs running at 2.2 GHz, 8 GB of DDR5 ECC RAM, a 200 GB SSD disk, and a 1 Gb network card was used. The VM was configured with Ubuntu Server 22.04.3 LTS

Algorithm 1 Capture Script

```
Start
SET Network Interfce
SET Container ID
SET Count
CREATE and initialize log files
SET count = 0
while While count ≤ 3600 do
    FETCH memory usage (used, free, shared, buffered, available)
    FETCH swap usage (used)
    FETCH disk usage for "sda2"
    FETCH CPU stats (user, system, IO wait, idle) using mpstat
    FETCH initial network received and transmitted bytes
    WAIT for 1 second
    FETCH updated network received and transmitted bytes
    CALCULATE received and transmitted differences
    FETCH stats for container
    FETCH current date and time
    WRITE system metrics to logs
    INCREMENT count by 1
end
```

Algorithm 2 Load Script

```
Start  while While ≤ 3600 do
    hey -z 1h -c 100 -cpus 12 -m GET "url" ⩾ log.csv
end
```

and Debian 12.5 as the operating systems. These systems were selected due to their open-source nature, security, and stability.

For container orchestration, Docker v2.27.0 and Podman v3.4.4 were used to set up a Kubernetes cluster comprising a control plane and two workers. These clusters ran a serverless HTTP application capable of processing GET and POST requests. Kubernetes v1.28, an open-source orchestration platform, provided an automated and scalable environment for managing containers. Additionally, the Knative framework v1.12.3 extended Kubernetes capabilities by simplifying and automating the deployment and maintenance of serverless applications. Knative employed Kubernetes Custom Resource Definitions (CRDs) to integrate seamlessly with native Kubernetes resources, supporting the execution of Functions as a Service (FaaS) workloads.

Docker and Podman, chosen as container orchestrators, differ in their use of container daemons; Docker relies on a centralized daemon, while Podman interacts directly with runC, that in theory increase its performance. This architectural distinction can influence overhead and potentially affect performance under varying workloads.

The experimental scenarios assessed various combinations of operating systems and container orchestrators under low, medium, and high workloads. Low workloads

simulated 100 users accessing the application simultaneously, medium workloads simulated 500 users, and high workloads simulated 1000 users. To avoid interference between tests, the server was restarted between scenarios. Each scenario was run for 60 minutes using Hey software, an open-source load-testing tool, to simulate HTTP GET and POST requests. After configuring Docker or Podman, a Kubernetes cluster with a control plane and two workers was established, followed by deploying a web application through Knative for testing.

This experimental design adheres to the ACM Empirical Standards for benchmarking by employing clearly defined metrics (CPU, RAM, Disk I/O, Network usage, and response times), consistent experimental conditions (sequential execution with server restarts), and reproducible methodologies (open-source scripts and detailed documentation). Statistical analysis, including confidence intervals and percentiles, ensures robustness in performance evaluation. The benchmarking scenarios and workloads align with established best practices in performance testing.

The data collection was automated using scripts, ensuring uniformity and reliability across all scenarios. Statistical measures, such as confidence intervals and standard deviation, were calculated to validate the results, providing a robust foundation for the conclusions.

Table 1 outlines the scenarios tested, detailing the specific combinations of operating systems, container orchestrators, HTTP methods, and workloads. This structured approach aims to propose effective configurations for future serverless infrastructure design, offering practical insights for service providers and developers.

Table 1 Experimental Scenarios

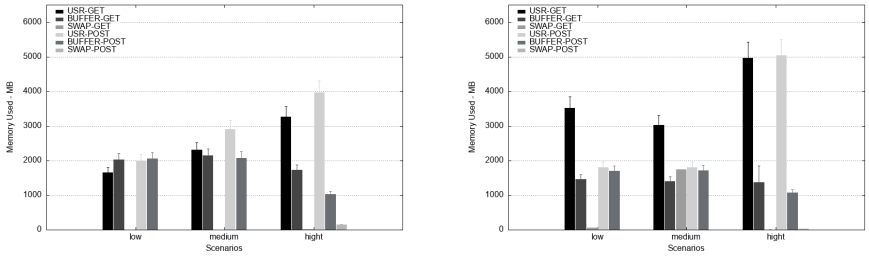
OS	ORCHESTRATOR	METHOD	WORKLOADS
Ubuntu	Docker	Get	low/medium/high
Ubuntu	Podman	Get	low/medium/high
Debian	Docker	Get	low/medium/high
Debian	Podman	Get	low/medium/high
Ubuntu	Docker	Post	low/medium/high
Ubuntu	Podman	Post	low/medium/high
Debian	Docker	Post	low/medium/high
Debian	Podman	Post	low/medium/high

5 Results and Discussion

The data collected from the 24 executed scenarios allowed for an in-depth examination of CPU, RAM, Disk I/O, Network consumption, and Response Time. Confidence Intervals (CI) at a 95% level were calculated to gauge the reliability of the results, while the Design of Experiments (DoE) approach helped to identify relationships between variables and optimize scenario selection.

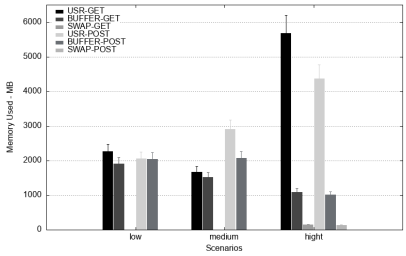
5.1 Memory and CPU usage

The memory usage analysis (Figure 3) reveals distinct patterns. For instance, Ubuntu with Podman in the GET method under low load consumed more than 3000 MB of RAM, surpassing other configurations. In contrast, Ubuntu with Docker consistently showed lower RAM usage around 2000 MB, suggesting that this combination manages memory more efficiently across both GET and POST methods. At medium loads, Debian with Podman exhibited higher memory consumption, while Debian with Docker remained comparatively lower. Under high load conditions, Debian with Docker demonstrated greater RAM consumption for the GET method, whereas Debian with Podman showed elevated values for POST requests.

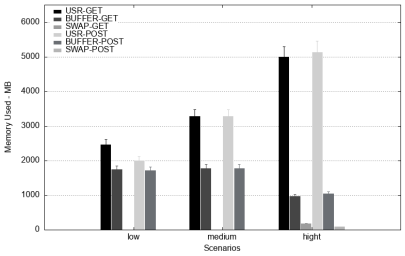


(a) RAM - Ubuntu x Docker

(b) RAM - Ubuntu x Podman



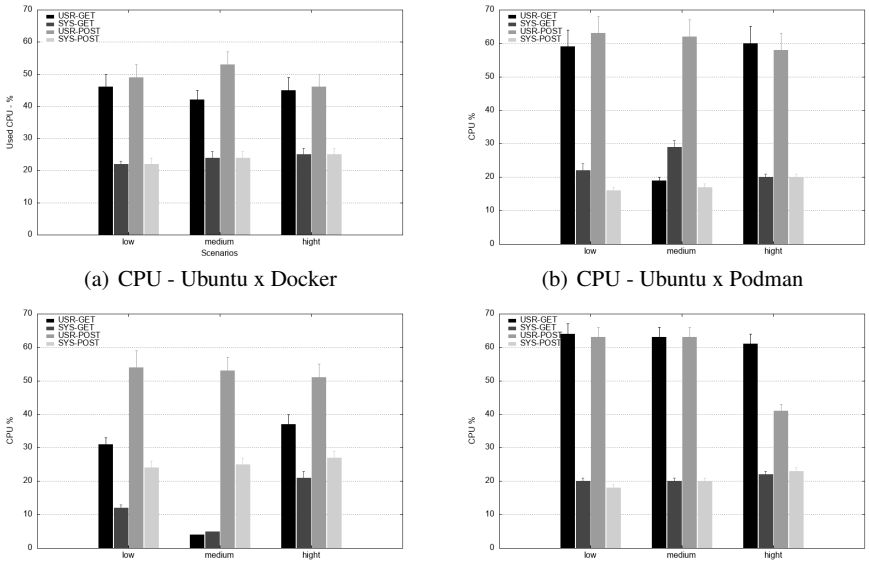
(c) RAM - Debian x Docker



(d) RAM - Debian x Podman

Fig. 3 Memory usage

CPU usage measurements (Figure 4) indicated that Debian combined with Docker often achieved lower CPU consumption, particularly at moderate loads, whereas Ubuntu with Podman sometimes showed CPU utilization above 60% even under lower load conditions. The POST method generally demanded more CPU than the GET method, frequently nearing or exceeding 50% utilization. These observations confirm that container runtime and workload type significantly influence resource usage.



(a) CPU - Ubuntu x Docker

(b) CPU - Ubuntu x Podman

(c) CPU - Debian x Docker

(d) CPU - Debian x Podman

Fig. 4 CPU usage

5.2 Processes and Response Time

The analysis of background processes such as Kubelet, Containerd, Conmon, and others indicated minimal CPU and RAM consumption (see Tables 2 and 3). Although some SWAP memory usage appeared during peak loads, its magnitude remained low, suggesting that performance was not critically impacted. This stable behavior aligns well with the expectations of serverless architectures, where automatic scaling and resource provisioning should prevent severe resource shortages.

Table 2 Resources utilization-Proccess Kubelet

Scenario	Method	CPU (%)	RAM (MB)	SWAP (MB)	VSZ (MB)
Debian x Docker	GET	7	1.1	2.8	2258.5
Debian x Docker	POST	3	1.4	5.5	2173.2
Debian x Podman	GET	4	1	5.6	2481.8
Debian x Podman	POST	3	2	3.8	2481.7
Ubuntu x Docker	GET	2	1	1	2407.2
Ubuntu x Docker	POST	1	1	2.1	2555.6
Ubuntu x Podman	GET	4.95	1	23.12	1104.1
Ubuntu x Podman	POST	2	1	27.1	2704.2

Response time data (Figure 4) shows that Ubuntu paired with Podman occasionally exceeded a 2-second threshold, which can degrade user experience and po-

Table 3 Resources utilization-Proccess Containerd

Scenario	Method	CPU (%)	RAM (MB)	SWAP (MB)	VSZ (MB)
Debian x Docker	GET	2	1	7.9	2173.2
Debian x Docker	POST	2	1	15.1	3082.6
Debian x Podman	GET	4	1	4.5	3428.4
Debian x Podman	POST	3	1	6.8	3469.2
Ubuntu x Docker	GET	1	1	3.6	2921.9
Ubuntu x Docker	POST	2	1	6.7	2615.9
Ubuntu x Podman	GET	3	1,5	11.50	3384.1
Ubuntu x Podman	POST	2	1	5.6	5568.0

tentially push users toward alternative solutions. Conversely, Ubuntu with Docker maintained sub-second response times, providing a more responsive environment. Debian scenarios fell between these extremes, generally sustaining acceptable response times but not consistently matching Ubuntu with Docker's lower latency.

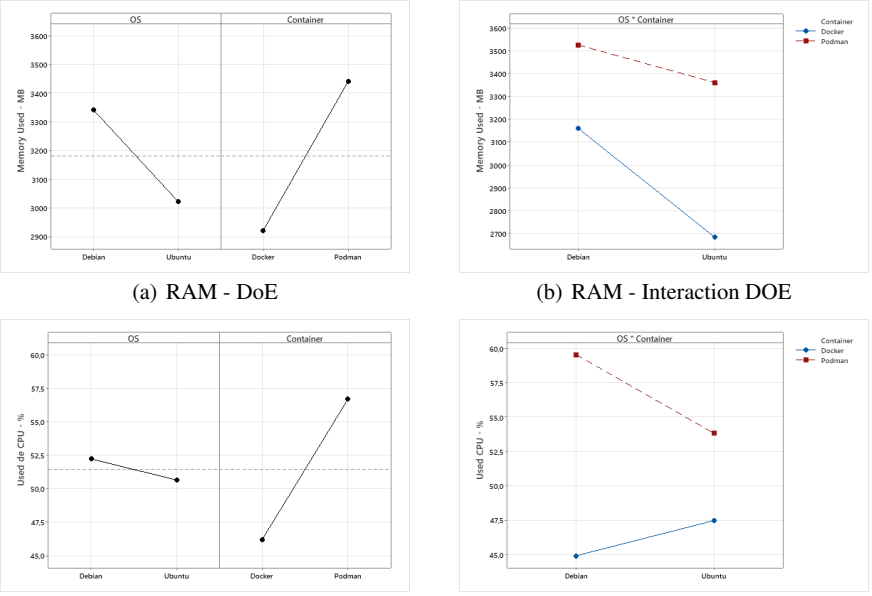
Table 4 Response Time

Scenario	Method	Time (sec)
Debian x Docker	Get	1.033
Debian x Docker	Post	0.930
Debian x Podman	Get	0.414
Debian x Podman	Post	0.448
Ubuntu x Docker	Get	0.320
Ubuntu x Docker	Post	0.209
Ubuntu x Podman	Get	2.730
Ubuntu x Podman	Post	1.238

5.3 DoE Analysis

Using the Design of Experiments (DoE) [4] analysis, the data were analyzed to identify patterns, determine the best configurations and model the relationship between the collected data.

Figure 5(a) shows that Ubuntu and Docker had lower RAM consumption than Debian and Podman. Figure 5(b) shows the interaction graph between the Operating Systems and Container Orchestrators, demonstrating that the Ubuntu x Docker interaction has lower RAM consumption. Figures 5(c) and 5(d) show the CPU consumption with the Debian and Docker interaction presenting the lowest resource consumption, despite a small difference between Ubuntu and Docker.



(a) RAM - DoE

(b) RAM - Interaction DOE

(c) CPU - DOE

(d) CPU - Interaction DOE

Fig. 5 DoE Analysis

6 Final Remarks

The insights gained from evaluating various OS and container orchestrator configurations highlight that subtle architectural choices can significantly influence serverless performance. Ubuntu combined with Docker consistently demonstrated efficient resource usage and low latency, indicating that certain runtime pairings can yield tangible advantages in environments demanding rapid responses and predictable scalability. In contrast, Debian paired with Podman exhibited higher overhead and proved more cumbersome to manage, reflecting how particular combinations may not be well-suited for all operational contexts.

These findings hold broader applicability across diverse scenarios. In latency-sensitive domains such as financial trading, high-performance computing, or edge computing, selecting configurations that optimize response times and resource utilization can directly impact throughput and user satisfaction. Similarly, organizations constrained by cost, compliance requirements, or hardware limitations can leverage this understanding to deploy serverless infrastructures that strike a meaningful balance between performance and practicality.

Future research may delve deeper into alternative frameworks beyond Knative, explore emerging technologies like WebAssembly-based serverless platforms, and consider more complex workloads, including data-intensive or GPU-accelerated tasks. Additionally, examining infrastructure choices in IoT devices, edge environ-

ments, and other specialized contexts can refine these insights further. Such advancements will help guide continuous performance optimization and offer more granular, informed recommendations, ultimately supporting a broader range of serverless applications and operational goals.

References

1. Chard, R., Skluzacek, T.J., Li, Z., Babuji, Y., Woodard, A., Blaiszik, B., Tuecke, S., Foster, I., Chard, K.: Serverless supercomputing: High performance function as a service for science (2019)
2. Enes, J., Expósito, R.R., Touriño, J.: Real-time resource scaling platform for big data workloads on serverless environments. Future Generation Computer Systems **105**, 361–379 (2020)
3. Hassan, H.B., Barakat, S.A., Sarhan, Q.I.: Survey on serverless computing. Journal of Cloud Computing **10**(1), 1–29 (2021)
4. Jankovic, A., Chaudhary, G., Goia, F.: Designing the design of experiments (doe)–an investigation on the influence of different factorial designs on the characterization of complex systems. Energy and Buildings **250**, 111,298 (2021)
5. Jia, Z., Witchel, E.: Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 152–166 (2021)
6. Kaloudis, M.: Evolving software architectures from monolithic systems to resilient microservices: Best practices, challenges and future trends. International Journal of Advanced Computer Science & Applications **15**(9) (2024)
7. Li, Y., Lin, Y., Wang, Y., Ye, K., Xu, C.: Serverless computing: state-of-the-art, challenges and opportunities. IEEE Transactions on Services Computing **16**(2), 1522–1539 (2022)
8. Lin, C., Khazaei, H.: Modeling and optimization of performance and cost of serverless applications. IEEE Transactions on Parallel and Distributed Systems **32**(3), 615–632 (2020)
9. Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H., Jin, X.: Faaslight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Transactions on Software Engineering and Methodology (2023)
10. Melo, P., Gama, L., Dantas, J., Beserra, D., Araujo, J.: Performance evaluation of container management tasks in os-level virtualization platforms. In: 2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 1–6. IEEE (2023)
11. Mohanty, S.K., Premsankar, G., Di Francesco, M., et al.: An evaluation of open source serverless computing frameworks. CloudCom **2018**, 115–120 (2018)
12. Nguyen, H.D., Yang, Z., Chien, A.A.: Motivating high performance serverless workloads. In: Proceedings of the 1st Workshop on High Performance Serverless Computing, pp. 25–32 (2020)
13. Scheuner, J., Leitner, P.: Function-as-a-service performance evaluation: A multivocal literature review. Journal of Systems and Software **170**, 110,708 (2020)
14. Wen, J., Chen, Z., Sarro, F., Liu, X.: Superflow: Performance testing for serverless computing (2023)
15. Yu, M., Jiang, Z., Ng, H.C., Wang, W., Chen, R., Li, B.: Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), pp. 138–148. IEEE (2021)
16. Zhang, M., Krintz, C., Wolski, R.: Edge-adaptable serverless acceleration for machine learning internet of things applications. Software: Practice and Experience **51**(9), 1852–1867 (2021)