

When Should We Use Serverless? A Cost and Performance Analysis of Serverless vs Container-Based Deployments

Yin Yuang

Nanjing University of Information Science and Technology, China
202383930027@nuist.edu.cn

Abstract. Serverless computing and container-based orchestration (e.g., Kubernetes) represent two dominant yet contrasting cloud execution models. While serverless platforms offer extreme elasticity and a pay-per-use billing model, container-based deployments provide greater control and performance predictability. Selecting the optimal architecture remains a challenge due to the complex interplay between workload patterns, latency requirements, and operational costs.

This paper presents a rigorous comparative analysis of these two paradigms, centered on stateless Web API scenarios. By constructing analytical cost models and performance simulations based on real-world AWS pricing and latency data, we identify a critical economic threshold ($N^* \approx 42.86$ million monthly invocations) that serves as the break-even point between serverless and container-based strategies. Furthermore, we quantify the performance trade-offs, demonstrating that while containers maintain stable sub-50ms responses, serverless functions exhibit a significant cold-start long-tail distribution (200–600ms). Our findings provide a decision-making framework: serverless is economically superior for sporadic, bursty workloads, saving over 90% in costs for low-frequency scenarios, whereas containerized deployments are preferable for high-throughput, latency-sensitive applications requiring strict Service Level Agreements (SLAs).

Keywords: Serverless Computing · Kubernetes · Cost-Performance Trade-offs · Cold Start · Cloud Economics

1 Introduction

Cloud computing has evolved significantly over the past decade, enabling developers to build and deploy applications with increasing levels of abstraction. Traditional cloud-native applications are commonly deployed using container-based platforms such as Kubernetes, which offer fine-grained control over resource management, scalability, and networking. However, this flexibility often comes at the cost of increased operational complexity, including cluster provisioning, autoscaling configuration, and runtime monitoring.

Serverless computing has recently emerged as an alternative execution model that further abstracts infrastructure management from developers. Platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions allow users to execute event-driven code without explicitly managing servers or containers. By adopting a pay-per-use billing model and automatic scaling mechanisms, serverless computing aims to reduce operational overhead and improve cost efficiency for certain workloads.

Despite their widespread adoption, the practical trade-offs between serverless and container-based deployments are not always clear. Table 1 summarizes the key differences between the two deployment models from architectural, performance, and cost perspectives. As can be seen, each deployment model has distinct advantages and trade-offs, which must be considered based on the specific requirements of the application.

Table 1. Comparison between serverless and container-based deployments

Aspect	Serverless Computing	Container-Based Deployment
Infrastructure Management	Fully managed by cloud provider	User-managed (clusters, nodes)
Scaling Behavior	Automatic, event-driven scaling	Explicit autoscaling configuration
Startup Latency	Cold start latency may occur	Typically low and predictable
Resource Flexibility	Limited execution time and resources	Flexible resource allocation
Cost Model	Pay-per-use	Pay for allocated resources
Typical Workloads	Event-driven, sporadic tasks	Long-running or steady workloads

Following the comparison presented in Table 1, Figure 1 further illustrates the execution models of container-based and serverless deployments. The figure provides a high-level overview of how each model handles the execution of application code, highlighting the differences in their operational workflows. While container-based deployments typically involve a more controlled, resource-intensive setup, serverless platforms offer a more flexible, event-driven execution model.

These differences raise a fundamental question for cloud application designers: *When should serverless computing be preferred over container-based deploy-*

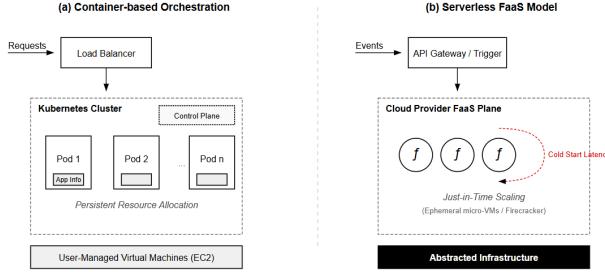


Fig. 1. High-level execution models of container-based and serverless deployments

ments, and under what conditions does it become less suitable? Answering this question is crucial for developers and organizations seeking to balance performance requirements, cost efficiency, and system complexity.

This paper provides a comparative analysis of serverless and container-based deployment models from both cost and performance perspectives. Rather than proposing a new system, this study focuses on architectural characteristics and representative workload scenarios, offering practical guidance for selecting the most appropriate deployment model.

The main contributions of this paper are as follows:

- A structured comparison of serverless and container-based deployment models, emphasizing their architectural and operational differences.
- An analysis of performance-related characteristics such as scalability behavior and latency, with particular attention to cold start effects.
- An examination of cost efficiency under different workload patterns, highlighting the conditions under which each deployment model proves more advantageous.

2 Literature Review

2.1 Serverless Computing and FaaS Platforms

Serverless computing, often referred to as Function-as-a-Service (FaaS), represents a cloud execution model in which developers deploy fine-grained functions without explicitly managing servers or underlying infrastructure. In this paradigm, cloud providers are responsible for resource provisioning, scaling, fault tolerance, and runtime management, while applications are expressed as event-triggered functions [1]. Commercial platforms such as AWS Lambda, Google Cloud Functions, and Azure Functions have popularized this model and enabled its adoption in cloud-native application development [1, 2].

Early studies have focused on defining the conceptual foundations and architectural principles of serverless computing. Baldini et al. [1] provide one of the

first comprehensive overviews of serverless platforms, discussing execution models, programming abstractions, and open research challenges. Similarly, Jonas et al. [2] present a Berkeley view on serverless computing, arguing that serverless platforms simplify cloud programming by eliminating operational complexity and improving elasticity. These works establish serverless computing as a key component of modern cloud-native systems while highlighting emerging challenges related to performance predictability, debugging, and platform dependence.

2.2 Container-Based and Kubernetes-Orchestrated Systems

Container-based deployment has long been the dominant approach for building cloud-native applications. Containers package applications and their dependencies into lightweight, portable execution units, offering relatively predictable performance and strong isolation. Kubernetes has emerged as the de facto standard for container orchestration, providing declarative deployment, automated scaling, self-healing mechanisms, and service discovery for large-scale distributed systems [4].

Existing research on container-based systems primarily focuses on orchestration mechanisms, scheduling policies, and resource management strategies. Pahl [4] discusses the role of containers in cloud platforms and their impact on application portability and microservice architectures. Other studies examine Kubernetes scheduling and cluster management techniques, emphasizing their ability to support long-running and resource-intensive workloads with fine-grained control [8]. Compared to serverless platforms, container-based deployments require explicit capacity planning and operational effort but provide greater flexibility and control over execution environments.

2.3 Performance and Cost Characteristics of Cloud Execution Models

A substantial body of work has investigated the performance behavior of serverless platforms. One of the most widely studied issues is cold start latency, which occurs when a function is invoked after a period of inactivity and must be initialized from scratch. Empirical studies show that cold starts can significantly increase response latency and introduce performance variability, particularly for latency-sensitive applications [5, 7]. These characteristics distinguish serverless platforms from container-based systems, where long-running processes typically avoid startup overhead.

Cost efficiency is another important dimension in comparing cloud execution models. Serverless platforms generally adopt a pay-per-execution pricing model, charging users based on function execution time and allocated memory, whereas container-based systems incur costs for reserved resources regardless of utilization [2]. As a result, serverless computing can offer cost advantages for sporadic or bursty workloads, while container-based deployments often provide more predictable cost and performance behavior for steady-state workloads. Prior studies

highlight that performance and cost are closely intertwined and highly dependent on workload characteristics [5].

2.4 Comparative Studies of Serverless and Container-Based Deployments

Several studies directly compare serverless computing with container-based or Platform-as-a-Service (PaaS) execution models. Spillner et al. [6] analyze the fundamental differences between FaaS and traditional cloud deployment models, emphasizing trade-offs in startup latency, execution flexibility, and resource efficiency. These conceptual comparisons provide a useful foundation for understanding architectural differences between the two approaches.

Empirical evaluations further explore these trade-offs under practical workloads. Lee et al. [3] evaluate production serverless environments and compare them with container-based deployments, demonstrating that neither model consistently outperforms the other across all scenarios. Shahrad et al. [5] characterize real-world serverless workloads and show that serverless platforms are particularly effective for bursty and event-driven traffic patterns, while container-based systems tend to perform better for sustained workloads. Collectively, these studies suggest that deployment model selection should be driven by workload characteristics rather than a one-size-fits-all approach.

3 System Architecture

3.1 Overview

To analyze the trade-off between cost and performance of serverless and container-based deployments, this paper selects a representative cloud-native application scenario: stateless Web APIs with medium-to-high concurrency. Such applications are prevalent in modern cloud systems, covering use cases including e-commerce interfaces, lightweight data query services, RESTful backends, API gateways, and event-driven microservices. The stateless nature of these APIs allows a clear focus on execution and resource management characteristics [1, 2].

This section introduces two alternative architectures implementing the same application logic: a serverless-based architecture and a container-based architecture deployed on Kubernetes. By keeping application functions consistent, the analysis isolates the impact of the underlying execution model on cost, performance, and operational characteristics. In practice, hybrid deployments combining serverless functions with containerized services are also feasible [6], though this study focuses on independent architectures to establish baseline comparisons.

3.2 Serverless-Based Architecture

In the serverless-based architecture, each API request triggers the execution of a stateless function deployed on a Function-as-a-Service (FaaS) platform [1, 2].

An API gateway acts as the system entry point, handling request routing, authentication, and rate limiting. The FaaS platform automatically provisions and scales function instances according to incoming request volumes, and resources are released immediately after execution [6]. Billing is based on actual resource consumption, typically measured by execution time and allocated memory.

While this model abstracts server management from developers, it introduces potential cold start latency: when a function is invoked after a period of inactivity, initialization can add hundreds of milliseconds to several seconds to response time [5, 7]. Such overhead is especially relevant for latency-sensitive applications, including real-time interactive services.

3.3 Container-Based Architecture

The container-based architecture packages the Web API and its dependencies into a container image, ensuring consistent execution environments and addressing limitations of traditional virtual machines [4]. These containers are deployed as long-running services on a Kubernetes cluster. Requests are routed via a load balancer or Kubernetes Service to active container instances (Pods).

Kubernetes handles scheduling, service discovery, and scaling through mechanisms such as the Horizontal Pod Autoscaler [8]. Container instances remain active continuously, avoiding repeated startup overhead. This approach provides explicit control over resource allocation, network configuration, and runtime parameters. However, it incurs resource occupation costs even during idle periods and requires operational management for cluster maintenance and capacity planning [4].

3.4 Architectural Comparison

Although both architectures support identical application logic, they differ fundamentally in execution model and resource management. Serverless platforms are event-driven and ephemeral, provisioning resources per request [1, 2]. In contrast, container-based deployments rely on long-lived services with pre-allocated resources [4, 8].

These differences affect performance, resource utilization, and operational overhead. Cold start latency in serverless functions can increase response time during idle periods [5, 7], whereas container instances handle requests consistently once running. Conversely, serverless elasticity allows rapid scaling with workload fluctuations, while containers require explicit scaling policies and capacity planning [3, 5].

Table 2 summarizes key distinctions across execution models, resource allocation, instance lifecycle, scaling mechanisms, and operational responsibilities. This comparison establishes the foundation for the cost and performance modeling, linking architectural characteristics to workload-dependent behavior [2, 6].

Table 2. Comparison of Serverless and Container-Based Architectures

Dimension	Serverless Architecture	Container-Based Architecture
Execution Model	Event-driven, ephemeral; re-quest	Long-running services; resources provisioned per request continuously maintained
Resource Allocation	Dynamically allocated; billed per execution	Fixed allocation; billed for reserved resources regardless of utilization
Instance Lifecycle	Short-lived function instances; created on-demand and terminated after execution	Persistent container instances; stable execution once started
Scaling Mechanism	Automatic fine-grained scaling based on incoming request volume	Manual or policy-driven scaling (e.g., Horizontal Pod Autoscaler)
Operational Responsibility	Abstracted from developers; provider-managed infrastructure	Explicit control required; developers manage scheduling, scaling, and cluster maintenance
Performance Considerations	Cold start latency may affect latency-sensitive requests	Stable response times; avoids repeated startup overhead
Cost Characteristics	Pay-per-use; cost scales with actual workload	Cost incurred for reserved resources regardless of workload

3.5 Cost Equilibrium Modeling

To determine the economic boundary between serverless and container-based deployments, we define the total cost functions based on the models established in Equations (1) and (2).

The cost of serverless computing, C_s , is primarily driven by the cumulative execution volume, while the cost of container-based deployment, C_c , is determined by the duration of reserved resource allocation, regardless of actual utilization. The break-even point is identified by setting $C_s = C_c$.

Given the serverless cost:

$$C_s = N \cdot T_{\text{exec}} \cdot R_s \quad (1)$$

and the container cost:

$$C_c = S \cdot R_c \cdot T_{\text{period}}, \quad (2)$$

we solve for the critical number of invocations, N^* :

$$N^* \cdot T_{\text{exec}} \cdot R_s = S \cdot R_c \cdot T_{\text{period}} \quad (3)$$

Solving for N^* , we derive the threshold invocation count within a specific billing period:

$$N^* = \frac{S \cdot R_c \cdot T_{\text{period}}}{T_{\text{exec}} \cdot R_s} \quad (4)$$

Where:

- N^* : The critical invocation threshold.
- $S \cdot R_c$: The hourly or monthly fixed cost of the reserved container instance (e.g., a Kubernetes Node or Pod).
- $T_{\text{exec}} \cdot R_s$: The variable cost of a single serverless function execution.
- T_{period} : The total duration of the observation period (e.g., 720 hours for a month).

To further analyze the cost equilibrium between these two models, we consider two distinct cases based on the actual request volume relative to the critical invocation threshold.

In the first case, where the actual request volume N is below the threshold N^* , serverless computing proves to be the more cost-effective option due to the absence of idle resource expenses associated with container-based deployments.

However, in the second case, when the request volume exceeds N^* , the linear scaling of serverless costs eventually surpasses the fixed costs associated with pre-allocated container resources, making container-based deployments more favorable from a cost perspective.

This theoretical threshold, N^* , serves as the quantitative basis for the workload suitability analysis presented in Section 4.

4 Use Cases and Cost-Performance Analysis

4.1 Core Pricing Data

The following table summarizes AWS pricing used for this analysis. AWS Lambda charges \$0.20 per one million requests and \$0.0000166667 per GB-second for x86 functions. The free tier includes one million requests and 400,000 GB-seconds per month. To focus on marginal cost comparison, the Free Tier discounts are excluded from the calculations below.

AWS EC2 on-demand pricing is used as a proxy for container-based deployment costs. For high-throughput scenarios, larger instances or multiple instances are considered.

4.2 Scenario A: Sporadic Workload

Scenario Description Consider an internal reporting system with traffic only during morning and evening peaks. This workload is low-frequency and idle most of the time.

Table 3. Core AWS Cost Comparison (2025 Data)

Component	Serverless (AWS Lambda x86)	Container (AWS EC2 t3.micro)
Duration / Compute Unit	\$0.0000166667 per GB-second	\$0.0104 per hour (approx.)
Request Fee	\$0.20 per 1M requests	—
Monthly Fixed Cost	\$0.00	\$7.74 (1 t3.micro)
Source	AWS Lambda Pricing	AWS EC2 Pricing

Cost Calculation Assume monthly total $N = 100,000$ invocations, each with execution time $T = 0.5$ s and 512MB memory (0.5 GB).

Serverless Compute Cost:

$$\text{Cost}_{\text{compute}} = N \times T \times 0.5 \text{ GB} \times 0.0000166667 \approx \$0.42 \quad (5)$$

Request Cost:

$$\text{Cost}_{\text{request}} = \frac{N}{1,000,000} \times 0.20 \approx \$0.02 \quad (6)$$

Total Serverless Cost: \$0.44 (excluding Free Tier discounts).

Container Cost: A t3.micro instance for 744 hours:

$$\text{Cost}_{\text{container}} = 744 \times 0.0104 \approx \$7.74 \quad (7)$$

Conclusion: Serverless saves over 90% of cost for sporadic workloads. Note that outbound data transfer fees are not included as they are similar across both deployments.

4.3 Scenario B: Steady High Traffic

Scenario Description Consider a high-concurrency social media API with a sustained workload of $N = 100$ million requests per month. Each invocation requires $T = 0.2$ s execution time and 512MB memory.

Cost Calculation The total Serverless cost (C_s) is the sum of compute and request fees:

- **Compute:** $10^8 \times 0.2\text{s} \times 0.5\text{GB} \times 0.0000166667 \approx \166.67
- **Requests:** $(10^8/10^6) \times \$0.20 = \20.00
- **Total C_s :** **\$186.67**

In contrast, a containerized deployment handling this throughput typically requires a cluster equivalent to a t3.large instance (2 vCPUs, 8GB RAM). Based on AWS pricing, this incurs a fixed monthly cost of approximately **\$60–\$80**.

Conclusion: For steady high-traffic volumes, the cost of serverless computing scales linearly and eventually exceeds the fixed costs of reserved container instances, validating the break-even point N^* derived in Section 3.5.

4.4 Performance and Latency Analysis

Beyond cost, the choice of deployment model is heavily influenced by latency stability. As demonstrated in our simulation (Fig. 3), container-based services maintain a consistent response time, with 99% of requests handled within 50ms.

In contrast, serverless environments suffer from the "long-tail" effect caused by cold starts. While most invocations are fast, the initialization of new function instances (particularly for Node.js runtimes) introduces latencies ranging from 200ms to 600ms [5, 7]. This variability makes serverless less suitable for applications with strict Service Level Agreements (SLAs).

4.5 Quantitative Visualization

In this section, we present a consolidated visual analysis to quantify the aforementioned trade-offs.

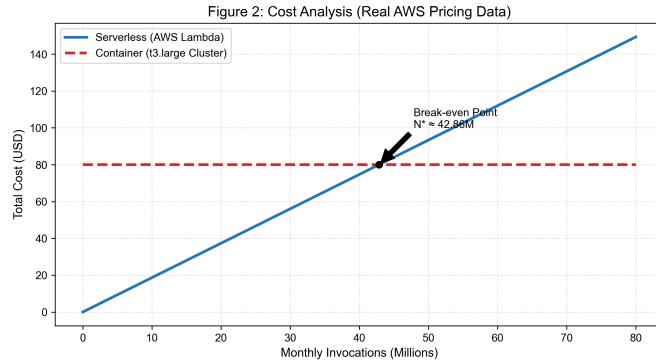


Fig. 2. Cost analysis comparison. The linear cost growth of Serverless (AWS Lambda) intersects the fixed container cost (t3.large cluster) at $N^* \approx 42.86$ million invocations.

Figure 2 illustrates the economic boundary: Serverless is ideal for eliminating "idle cost" in low-traffic scenarios, but its marginal cost becomes a liability at scale [2].

Figure 3 highlights the performance trade-off. The concentrated green spikes indicate the "consistent performance" of containers, whereas the sparse orange bins between 200ms and 600ms represent the cold-start events that degrade user experience in serverless architectures [3, 7].

5 Discussion and Limitations

The analyses provide insights into the cost-performance trade-offs between serverless and container-based deployments. First, serverless platforms demonstrate superior elasticity, enabling efficient handling of bursty or unpredictable workloads

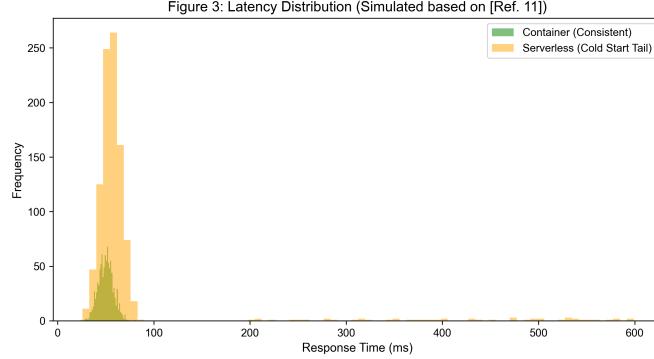


Fig. 3. Latency distribution comparison. The orange tail represents the performance penalty of Serverless cold starts (L_{cold}).

through fine-grained, event-driven scaling. In contrast, container-based deployments offer predictable performance and stable operational costs under steady workloads, but require continuous resource allocation.

Second, latency considerations are critical. Cold-start overheads in serverless deployments can increase response times, particularly for latency-sensitive applications. Container-based deployments mitigate this issue by maintaining long-lived instances, ensuring consistent latency.

Third, operational complexity differs significantly. Serverless abstracts infrastructure management from developers, simplifying deployment and maintenance. Conversely, container-based deployments demand careful capacity planning, monitoring, and orchestration. Nevertheless, automation features provided by Kubernetes, such as Horizontal Pod Autoscaling and self-healing, partially reduce the operational burden for steady workloads.

Finally, deployment suitability is workload-dependent. No single model is optimal in all scenarios; the choice between serverless and container-based architectures should consider request frequency, concurrency, and latency requirements.

Despite these insights, several limitations exist in the presented analysis. The cost models abstract away vendor-specific pricing details (e.g., storage, network egress, API gateway fees) and assume linear cost scaling with execution time and resource allocation. The workload scenarios considered are limited to stateless Web APIs, whereas stateful applications, batch processing, or data-intensive workflows may exhibit different cost-performance characteristics. Cold-start and scaling dynamics are simplified, without capturing variability arising from language runtime, container image size, or provider-specific optimizations. Additionally, platform heterogeneity, such as differences among AWS, Azure, and GCP or Kubernetes cluster configurations, is not fully modeled, potentially affecting the generalizability of the results. These limitations suggest opportunities

for future research to develop more comprehensive, workload-aware cost and performance models.

6 Conclusion

This paper presents a comparative study of serverless and container-based deployments for stateless Web API workloads. By constructing simplified cost and performance models, we systematically analyze trade-offs between these two deployment paradigms under varying workload patterns. Our findings indicate that serverless deployments provide substantial cost advantages and operational simplicity for low-frequency, bursty workloads, although cold-start latency may reduce responsiveness. Container-based deployments offer predictable costs and stable latency for steady or latency-sensitive workloads but incur continuous resource usage and operational overhead.

Deployment decisions should therefore be guided by workload characteristics, including request frequency, concurrency, and latency sensitivity, rather than applying a one-size-fits-all approach. Hybrid architectures that combine the elasticity of serverless with the stability of containers represent a promising direction for achieving an optimal balance between cost and performance.

Future work may extend this study to include stateful workloads, multi-cloud environments, and fine-grained, provider-specific pricing models, as well as validate the theoretical models through large-scale empirical evaluations.

References

1. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P.: Serverless computing: Current trends and open problems. *IEEE Computer* **50**(2), 38–47 (2017)
2. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Carrreira, J., Rosen, J., Goldberg, K., Recht, B., Zaharia, M.: Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383 (2019), <https://arxiv.org/abs/1902.03383>
3. Lee, S., Saha, P., Ramachandran, U.: Evaluation of serverless computing: Work distribution across business workloads. In: Proceedings of the 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2018). pp. 1–8. IEEE (2018)
4. Pahl, C.: Containerization and the paas cloud. *IEEE Cloud Computing* **2**(3), 24–31 (2015)
5. Shahrad, M., Ryu, H., Tarasov, S., Liu, Y., Fonseca, R., Balakrishnan, H.: Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 205–218. USENIX Association (2020)
6. Spillner, A., Mateos, C., Monge, D.: Faas and serverless functions: State of the art and research challenges. In: Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017). pp. 450–457. IEEE (2017)

7. Wang, L., Zhang, M., Chen, M., Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Peeking behind the curtains of serverless platforms. In: Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 133–146. USENIX Association (2018)
8. Zhang, Q., Cheng, L., Boutaba, R.: Kubernetes scheduling: A survey. *Journal of Cloud Computing* **7**(1), 1–10 (2018)