Demystifying Serverless Costs on Public Platforms: Bridging Billing, Architecture, and OS Scheduling

Changyuan Lin
University of British Columbia

Yuanzhi Ma*
Johns Hopkins University

Mohammad Shahrad
University of British Columbia

Abstract

Public cloud serverless platforms have attracted a large user base due to their high scalability, plug-and-play deployment model, and pay-per-use billing. However, compared to virtual machines and container hosting services, modern serverless offerings typically impose higher per-unit time and resource charges. Additionally, billing practices such as wall-clock time allocation-based billing, invocation fees, and usage rounding up can further increase costs.

This work, for the first time, holistically demystifies these costs by conducting an in-depth, top-down characterization and analysis from user-facing billing models, through request serving architectures, and down to operating system scheduling on major public serverless platforms. We quantify, for the first time, how current billing practices inflate billable resources up to 4.35× beyond actual consumption. Also, our analysis reveals previously unreported cost drivers, such as operational patterns of serving architectures that create overheads, details of resource allocation during keep-alive periods, and OS scheduling granularity effects that directly impact both performance and billing. By tracing the sources of costs from billing models down to OS scheduling, we uncover the rationale behind today's expensive serverless billing model and practices and provide insights for designing performant and cost-effective serverless systems.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **General and reference** → **Measurement**; • **Software and its engineering** → **Scheduling**.

Keywords: Serverless Computing, Cloud Computing, Performance Measurements, Billing Models, OS Scheduling

*Conducted the research while at The University of British Columbia.

1 Introduction

Serverless computing has become one of the mainstream cloud computing paradigms, enabling developers to quickly deploy scalable and event-driven applications on the cloud without needing to manage the underlying infrastructure [41, 50]. Major cloud providers offer serverless computing solutions, such as AWS Lambda [93], Google Cloud (GCP) Run functions [23], Azure Functions [12], IBM Cloud Code Engine [31], and Cloudflare Workers [33]. Serverless computing stands out as the purest existing pay-per-use cloud model, offering automated scaling—from zero to thousands of instances in seconds—and fine-grained billing. As a result, it is often advertised as cost-efficient [36, 43, 50, 92, 113].

The widely acknowledged benefits of serverless architectures—such as high scalability, fine-grained pay-per-use billing, freedom from infrastructure management, and seamless integration with other cloud services—are not without associated costs [44, 49, 110]. In terms of the per-unit resource price, serverless offerings are often priced higher than other cloud computing paradigms, such as virtual machines (VMs) and containers running on container hosting platforms. We demonstrate this by comparing the price of AWS Lambda functions, AWS EC2 VMs, and AWS Fargate containers, all configured on identical ARM-based hardware in the *us-east-2* region. We specifically chose ARM due to the diverse and performance-varying nature of AWS's x86 processors, which complicates fair comparisons across services. An AWS Lambda function with 1 vCPU, 1,769 MB of memory, and 512 MB of ephemeral storage costs 2.3034×10^{-5} per second [92], while a compute-optimized EC2 instance (`c6g.medium`) with 1 vCPU, 2 GB memory, and 1 GB storage and an AWS Fargate container with the identical resource allocation as EC2 cost only 9.4753×10^{-6} and 1.1003×10^{-5} per second, which are 41.1% and 47.8% of the AWS Lambda price. The cost of VMs can be further decreased by at least two times if using a burstable instance (e.g., AWS EC2 `t4g.small` flavor). Also, this comparison does not include the invocation fee of AWS Lambda, which is 2×10^{-7} for each request, whereas EC2 instances and Fargate containers do not charge request fees. Additionally, our analysis of billing practices on major serverless platforms uncovers significant over-accounting (§2), showing that users can be charged

for computing resources up to 4.35 times greater than their actual usage.

These observations motivate a fundamental research question: **What makes serverless expensive?** We argue that the root cause of the high unit prices and expensive billing practices in serverless lies in the architecture of modern serverless computing systems. Resource consumption and overhead incurred by the underlying runtimes and control plane for request serving, such as sandbox provisioning, isolation, request dispatch, and keep-alive, translate into higher per-unit charges passed on to serverless users. Additionally, some of our measurements of resource allocation patterns and performance behaviors on major serverless platforms point part of the execution costs and performance fluctuations to the underlying operating system (OS) scheduling mechanisms.

To uncover these costs, a detailed analysis of current billing models together with measurements of the underlying serverless systems is required. Previous studies have characterized major serverless platforms in terms of architecture, performance, and resource management [1, 40, 109, 114]. However, as serverless computing evolves rapidly and more serverless offerings become available, some earlier measurements do not reflect or fully capture the latest billing scheme and operation patterns (e.g., serving architecture and keep-alive behaviors) of the public serverless computing platforms. In this work, we revisit some of the previous measurements and extend some of their performance and overhead characterization to fit modern serverless systems.

We adopt a top-down approach to analyze serverless costs. We start with user-facing billing models and conduct large-scale trace analysis on the billing scheme. Then, we analyze the performance, overhead, and resource allocation patterns of modern serverless request serving architectures. Finally, we investigate the impact of OS scheduling in detail. By tracing sources of costs from the billing model down to kernel scheduling, we provide the first comprehensive decomposition of serverless overhead and reveal the rationale behind current billing practices. For example, our large-scale, trace-based billing model analysis reveals significant bill inflation due to wall-clock allocation-based billing (§2.3), turnaround time billing (§2.4), rounding up of resource usage and execution duration, coarse billing granularity, and high invocation fees (Table 1 and §2.5). Also, we investigate the dual penalty of slowdowns and higher bills stemming from the multi-concurrency model (§3.1), high overheads of the HTTP-based request serving architecture (§3.2), and details of resource allocation during keep-alive (§3.3). Furthermore, we reveal the widespread CPU overallocation issue on public serverless platforms for the first time (§4). Specifically, our main contributions include:

- We conduct a detailed analysis on the billing practices of current major serverless platforms (§2).

- We analyze and quantify the overhead of modern serverless systems from several new aspects, including the concurrency model, request serving architectures, and resource allocation behaviors during keep-alive (§3).
- We characterize and reveal the impact of OS scheduling granularity on major public serverless platforms (§4).
- We demystify the serverless billing practice through these new characterization results and analyses, and discuss implications (labeled with I) for designing future performant and cost-efficient serverless systems.

We have made our artifact publicly available[1].

2 Serverless Billing Models and Practices

Pay-per-use is the common billing practice on serverless platforms. Billing models are the most direct determinants of the serverless cost as they convert billable resources (i.e., computing resources that are being billed for cloud users) into monetary charges that users immediately perceive. Billing models vary across platforms. In this section, we systematically deconstruct these billing practices to reveal how they shape the cost of serverless, reveal the underlying reasons for relevant billing practices, and discuss implications.

2.1 Overview of Serverless Billing Models

Table 1 summarizes the pay-per-use billing model on major serverless platforms listed in recent market reports [41, 77]. While definitions of billable resources, wall-clock time, and pricing vary across different serverless platforms, most public serverless platforms bill a function invocation based on four factors: (1) billable wall-clock duration, (2) resource allocation amount and/or actual resource consumption over billable duration, (3) billing granularity and/or minimum billing cutoffs, and (4) a fixed fee associated with each invocation, which can be generally modeled as:

$$\text{Cost} = \sum_{r \in R_{\text{ALLOC}}} \left\lceil \frac{ALLOC(r)}{G_r} \right\rceil \times G_r \times \left\lceil \frac{T}{G_T} \right\rceil \times G_T \times C_r$$
$$+ \sum_{r \in R_{\text{USG}}} \left\lceil \frac{USG(r)}{G_r} \right\rceil \times G_r \times C_r + C_0 \tag{1}$$

where T is the billable wall-clock time (e.g., wall-clock execution duration, turnaround time including initialization duration, or function instance lifespan), R_{ALLOC} is the set of billable computing resources that follow allocation-based billing (e.g., vCPUs, memory, GPU, and storage), $ALLOC(r)$ defines the allocation amount of billable resources r over T, R_{USG} is the set of billable resources to which consumption-based billing applies (e.g., network bandwidths and consumed CPU time of Cloudflare Workers), $USG(r)$ defines the absolute usage amount of billable resources r over T, G_r and G_T define the billing granularity of resource r and wall-clock time T for rounding up or minimum billing cutoff (e.g., 128 MB and

---

Serverless Platform	Billable Time	Billable Resources*	Billing Granularity/Cutoffs	Control Knobs and Steps
AWS Lambda [89, 92, 93]	Wall-Clock Turnaround Time**	Allocated Memory	1 ms	Memory 1 MB (CPU proportionally allocated)
Google Cloud Run (Request-Based Billing) [22, 23, 28]	Wall-Clock Turnaround Time	Allocated Memory and CPU	100 ms	Memory 1 MB CPU 0.01 vCPUs (1st Gen)/1 vCPU (2nd Gen)
Google Cloud Run (Instance-Based Billing)*** [22, 23, 28]	Wall-Clock Instance Time	Allocated Memory and CPU	100 ms	Memory 1 MB CPU 1 vCPU
Azure Functions Consumption Plan [11–13]	Wall-Clock Execution Time	Consumed Memory	1 ms (min cutoff 100 ms) 128 MB	N/A (Fixed resource size of 1.5 GB memory and 1 vCPU)
Azure Functions Premium Plan*** [10, 12, 13]	Wall-Clock Instance Time	Allocated Memory and CPU	1 month (minimum monthly cost applies)	CPU and Memory (Fixed Combos)
Azure Functions Flex Consumption Plan [8, 11, 12]	Wall-Clock Execution Time	Allocated Memory	100 ms (min cutoff 1 s)	Memory (Either 2 GB or 4 GB) (CPU proportionally allocated)
IBM Cloud Code Engine Function [31, 45]	Wall-Clock Turnaround Time	Allocated Memory and CPU	100 ms	Memory (Fixed Combos) CPU (Fixed Combos)
Huawei Cloud Function Graph [29]	Wall-Clock Execution Time	Allocated Memory	1 ms	Memory (Fixed CPU-Memory Combos)
Alibaba Cloud Function Compute [17–19]	Wall-Clock Execution Time	Allocated Memory and CPU	1 ms	Memory 64 MB CPU 0.05 vCPUs
Oracle Cloud Functions [32]	Wall-Clock Execution Time	Allocated Memory	Not Documented Publicly	Memory (Fixed Combos)
Vercel Functions [106]	Wall-Clock Execution Time	Allocated Memory	Not Documented Publicly	Memory 1 MB (CPU proportionally allocated)
Cloudflare Workers [33]	Consumed CPU Time	Consumed CPU	1 ms	N/A (Fixed resource size of 128 MB memory)

*This table and related analysis in §2 focus on the most basic billable computing resources (i.e., CPU and memory). Other billable resources (e.g., storage, GPUs, and network bandwidths) may apply in practice. **AWS bills wall-clock turnaround time that includes initialization duration starting August 2025 [48]. ***Instance-based billing applies, where platforms charge for resource allocation over the function runtime instance lifespan regardless of requests.

Table 1. The billable models of major public serverless platforms. The notion of billable time, billable resources, and billing granularity varies across different serverless platforms (as of 2025-05-15).

100 ms), C_r is the per-unit price of resource r, and C_0 is the fixed invocation fee.

Depending on whether to use the whole function instance lifespan as the billable time, the billing model can generally be categorized into request-based billing (e.g., platforms other than the two listed with instance-based billing in Table 1) and instance-based billing (e.g., Azure Functions Premium Plan and Google Cloud Run with instance-based billing in Table 1). In request-based billing, each request is charged separately based on its execution duration (or turnaround time) and/or allocated/consumed resources during the billable period, while instance-based billing usually charges for provisioned resources on always-ready/scaled-out instances (resource allocation over instance lifespan) regardless of requests. On most platforms, users can enable instance-based billing by changing the billing setting or configuring provisioned concurrency, minimum instances, or scale-down delay for their functions [22, 90]. The fixed invocation fee (C_0) is usually not applied under instance-based billing.

Figure 1 illustrates the CPU and memory prices on major serverless platforms presented in Table 1, which shows that the per-unit resource prices are often very similar across platforms. Following the serverless versus non-serverless cost comparison discussed in §1, this consistency in high per-unit resource prices indicates that **(I1) the high price of serverless computing is not the result of any single provider's billing strategy** (AWS already offers some of the lowest per-unit resource prices).
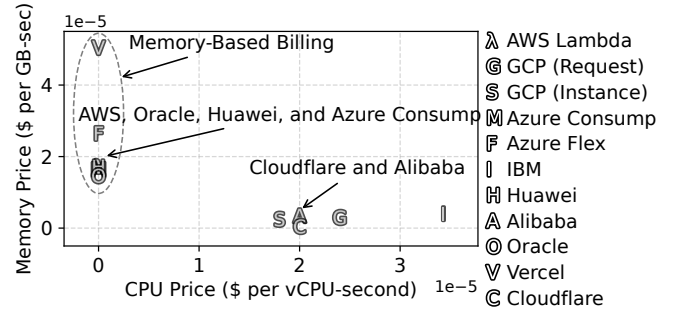

Figure 1. Resource (i.e., vCPU and memory) prices on major serverless platforms discussed in Table 1. The per-unit vCPUs and memory prices are generally similar across serverless platforms (as of 2025-05-15).

2.2 Coupled Control Knobs and Billable Resources

(I2) **Control knobs and billable resources are tied closely, but allocated resources are always billed directly or indirectly:** The billable resources in serverless billing models are usually mainly defined by the available control knobs. Public serverless platforms usually bill the computing resources they expose to users as control knobs. For example, AWS Lambda, Vercel Functions, and the Azure Functions Flexible Consumption allocate vCPUs in proportion to the allocated memory size. Some platforms, such as Huawei Function Compute, Azure Functions Consumption, and Oracle Functions, offer only a fixed set of memory sizes or fixed vCPU–memory pairs, rather than fine-grained configurations (e.g., per-MB memory configuration). In these cases, billing often appears to be based solely on memory

allocation/usage, but the cost of CPU is embedded implicitly within the memory price. For instance, an AWS Lambda function with 1,769 MB of memory (which corresponds to 1 vCPU [89]) incurs a charge of 2.8792×10^{-5} per second, while a GCP function (first generation with request-based billing) provisioned with 1 vCPU and 1,769 MB of memory costs 2.8319×10^{-5} per second. The price per GB-second of memory on the platforms that mainly expose and bill memory control knobs usually closely matches what one would pay for memory and CPU on platforms that allow separate CPU allocations. Also, the ratio between the unit prices of CPU (in vCPU-seconds) and memory (in GB-seconds) on GCP, AWS Fargate (a container hosting platform that bills CPU and memory separately) [91], and IBM Cloud Code Engine (function workloads) lies in a narrow range between 9 and 9.64, indicating a broad industry consensus on the relative value of vCPU versus memory.

On serverless platforms where vCPU and memory settings are relatively decoupled, CPU and memory are typically billed as two separate resources. However, even these platforms impose limits on how finely resources can be tuned. For example, Alibaba Cloud requires the ratio of vCPU to memory (in GB) to remain between 1:1 and 1:4, with step sizes of 0.05 vCPUs and 64 MB of memory [19]. Similarly, GCP imposes a minimum CPU allocation on the configured memory size (e.g., allocated memory of 512 MB must be configured with at least 0.333 vCPUs) [25]. These constraints on resource control knobs usually reflect an underlying function placement challenge: highly unbalanced CPU-to-memory combinations can fragment the resource capacity on host servers, potentially leading to higher deployment costs; e.g., through decreased deployment density [16, 68], or higher scheduling delay waiting for placement [53, 84].

2.3 Inflation of Billable Resources

(I3) **Billable resources are greatly inflated under wall-clock time allocation-based billing:** Understanding the amount of billable resources under different billing models is critical to evaluating the cost of serverless platforms. To measure how much billable resources users pay on public serverless platforms, we analyze 558.74 million[2] requests from the Huawei serverless trace (Huawei Public request tables) [51, 53] and compute the billable vCPU time and the billable memory resources of each request under the billing models presented in Table 1. To avoid distortions from differences in per-unit prices on different platforms (however, they are mostly similar as discussed in §2.1), we report raw billable resources rather than cost in dollars. Figure 2 shows the distribution of these billable vCPU times and memories across requests under several representative billing models and resource allocation patterns, including proportional
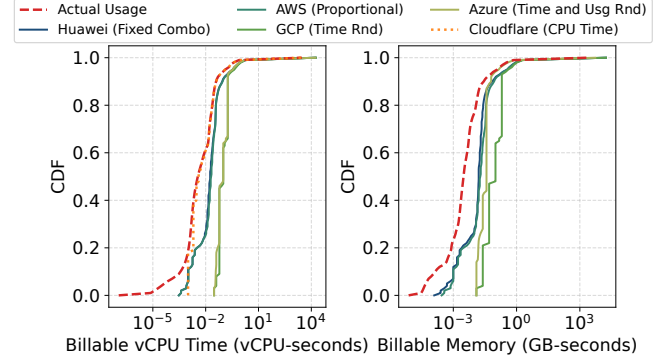
---

[2]The Huawei trace contains over 947.97 million requests. We exclude the requests reporting zero CPU usage or missing valid pod IDs/flavors.

Figure 2. Billable resources under different billing models. The billable resources can be multiple times higher than actual consumption on major serverless platforms.

vCPU allocation (AWS Lambda), fixed vCPU-memory combinations (Huawei), wall-clock duration and resource usage rounding (GCP and Azure), and usage-based billing (Cloudflare Workers). As discussed in §2.2, CPU pricing is usually embedded for platforms with memory-based billing. Therefore, we include billable vCPU time for AWS.

The gap between billed and actual resource usage quantifies the degree of inflated billable resources on current serverless platforms. Our analysis reveals that, under current models, billable vCPU time exceeds actual CPU usage by a factor of 1.01× (Cloudflare) up to 3.63× (GCP) on average, and billable memory exceeds real memory use by 1.57× (Azure) up to 4.35× (GCP) on average, among which usage-based billing (Cloudflare billable CPU and Azure billable memory) shows the lowest inflation. While differences in unit pricing shift the curve horizontally, these ratios remain the same (we do not compare absolute costs across platforms). Also, when mapping Huawei's reported vCPU and memory allocations to AWS, we choose the larger of the two values to match its proportional vCPU allocation, which makes AWS billable resources slightly higher than Huawei.

One of the major driving factors of inflated billable resources is allocation-based wall-clock time billing. Even AWS, with one of the finest billing granularities (i.e., 1 ms), charges billable vCPUs and memory, 2.49× and 2.72× higher than actual consumption on average. Functions rarely consume their full resource allocation [52, 75], and wall-clock time includes periods when functions hold resources but remain idle or use little (e.g., blocking on remote API calls). Figure 3 illustrates resource usage relative to allocations. More than 65% of requests use less than 50% of the allotted CPU, and around 76% of requests use less than half of the allotted memory. The scatter plot of CPU and memory utilization shows a Pearson correlation of 0.552 and a Spearman correlation of 0.565, which is slightly smaller than the value reported on serverless traces of Huawei private cloud (i.e., 0.6) in 2023 [52]. The moderate Pearson correlation suggests that the linear relationship is not strong, indicating that decoupling vCPU
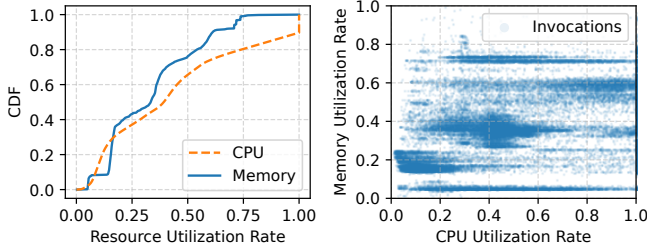
Figure 3. Resource utilization rate distributions and their correlations. Huawei serverless traces [51, 53] show that serverless functions usually have low utilization of allocated resources. Billable resources are further inflated by inflexible resource control knobs as no strong linear relationships between CPU and memory utilization rates exist.

and memory configurations is crucial for reducing inflated costs. Inflexible allocations (e.g., proportional/linear) force developers to over-provision one resource to satisfy another bottleneck [15, 107].

Although usage-based billing offers the lowest inflation in billable resources, it currently faces limitations and has not gained widespread adoption across providers. Cloudflare Workers is the only platform we studied that bills only on actual CPU time and aligns well with the actual CPU usage. However, it caps code artifacts at 10 MB and memory at 128 MB. This is mainly designed for small, short, single-threaded JavaScript or WebAssembly (Wasm) tasks (under 1–2 ms) running on the V8 engine within their content delivery network (CDN) [4, 34, 35], rather than general serverless workloads. Note that we do not argue that billing solely on the absolute amount of consumed resources is the only valid model in serverless computing. This is because resource allocation tied to wall-clock time, particularly for resources like memory (which is risky to overcommit [102]), significantly impacts function scheduling and deployment density. An ideal pay-per-use billing model is one that tracks real usage, exhibiting a perfect positive correlation statistically. There have been recent studies that aimed to move towards this ideal direction through dynamic or cooperative scheduling, or new billing models [16, 73, 81, 116].

2.4 Turnaround Time Billing and Cold Starts

(I4) **Billing on wall-clock turnaround time has become a common practice to compensate for the initialization phase:** The serverless runtime sandbox lifecycle typically consists of initialization (cold start), request execution, keep-alive, and shutdown (e.g., `SIGTERM` handling) [72, 94]. Depending on whether the initialization duration is included, serverless providers usually define billable wall-clock time as either execution time or turnaround time (i.e., execution time plus initialization) in their request-based, pay-per-use billing models. Besides billing based on execution time and
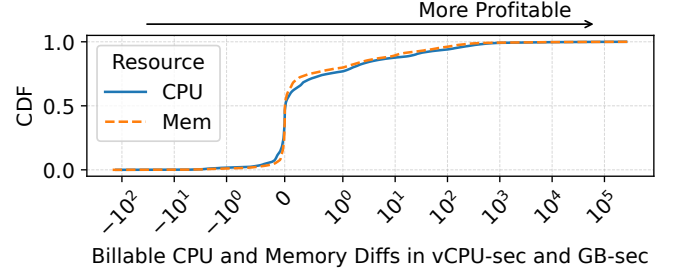

Figure 4. The differences between the billable CPU and memory resources consumed during request execution and those during initialization.

turnaround time, users may customize provisioned concurrency, minimum instances, or scale-down delay, and pay for the whole runtime instance lifespan (i.e., instance time) on most platforms. Such instance time billing can further increase billable resources under bursty traffic patterns since scale-down-to-zero is delayed or disabled, and instance idle time is billed. We observe that billing based on turnaround time has become increasingly popular on major platforms: GCP and IBM explicitly state that they charge for the turnaround time [28, 45], while AWS recently updated its billing model to include the initialization phase (cold start delay) starting August 2025 [48].

Our analysis of cold start resource usage helps explain why providers favor turnaround time billing. We analyze 388,955 traceable cold starts from the Huawei serverless traces [51, 53]. For each cold start, we consider the duration spent on the initialization of the runtime sandbox and the resource allocation. We compute the difference between the billable resources (measured in wall-clock resource allocations) consumed during cold start and the sum of billable resources used by all subsequent requests within the sandbox. A negative difference means the cold start alone consumed more billable resources than all later requests combined. Figure 4 quantifies, for the first time, such relative resource cost of cold starts compared to subsequent executions on production systems, which shows that 42.1% of cold starts produce a zero or negative difference. In other words, under a billing model based on purely execution duration of requests, providers would charge less (or the same) for request execution than the actual cost of the initialization phase in about 42.1% of cold start cases.

To avoid this revenue gap, it is a natural choice for providers to include initialization delay in the billed duration, i.e., to bill on turnaround time, which also captures variation in initialization delays (and wall-clock-based resource usage) across functions with different language runtimes and dependency requirements. Also, providers may impose additional billing components to offset cold start costs, such as a fixed per-invocation fee and minimum billing cutoffs for billable time and resources, which we further discuss in §2.5. Additionally, the results also show that a small portion of functions exhibit
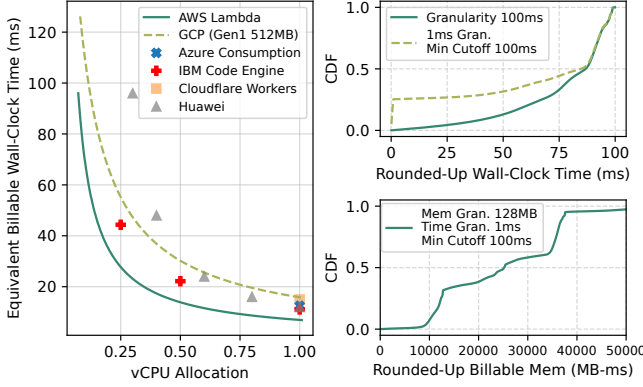
Figure 5. The equivalent billable wall-clock time of invocation fees (left) (as of 2025-05-15) and the rounded-up amount of billable wall-clock time and memory (right).

a long tail of negative resource differences, indicating that turnaround-time billing can substantially increase the cost of these functions.

2.5 High Invocation Fee and Expensive Rounding Up

Major serverless platforms charge a fixed fee per invocation, typically between 1.5×10^{-7} and 6×10^{-7} per request [17, 92, 106]. Although these amounts seem small, they can add up disproportionately when functions run for very short durations or use minimal resources. For example, on AWS Lambda, a fixed invocation fee of 2×10^{-7} is equivalent to 96 ms of billable wall-clock time for a function with the default 128 MB memory configuration, which exceeds the average execution durations reported in the Huawei traces (i.e., 58.19 ms) [53]. Figure 5-left shows how invocation fees convert to equivalent billable wall-clock time across different platforms. Besides invocation fees, several platforms apply coarse billing granularity (minimum billing increments) or cutoffs.

The charts on the right in Figure 5 present the inflated billable time and memory usage under different billing granularities for 527.05 million requests with execution times of at least 1 ms in Huawei traces [51, 53]. For a 100 ms billing granularity (e.g., GCP and IBM), the average rounded-up wall-clock time is 77.12 ms, while for a 1 ms granularity with a 100 ms minimum cutoff (e.g., Azure Consumption), the average rounded-up wall-clock time is 61.35 ms. When billing memory with a 128 MB granularity (e.g., Azure Consumption), the average rounded-up billable memory is 2.67×10^{-2} GB-seconds. These values are on the same order of magnitude as the average execution durations and billable memory amounts reported in the studied serverless traces (58.19 ms and 2.75×10^{-2} GB-seconds).

Therefore, our analysis shows that *(I5)* **invocation fees are high, and together with billing granularity, they can cause disproportionate costs for short, small function invocations**. These extra costs may not be explained

only as a way to offset resource usage during the initialization phase (cold start), since providers that bill for turnaround time still charge the invocation fee. They may further be linked to overheads in the serving architecture and OS scheduling, which we analyze further in §3 and §4.

3 Hidden Cost of Serverless Serving Architecture

Serverless computing platforms usually abstract away low-level infrastructure details. However, the overhead of the underlying serving layer directly affects how providers schedule and run serverless workloads and at what cost. These are passed on to the users as the billing model and pricing parameters. Therefore, studying serverless serving architectures is key to understanding serverless computing costs. In this section, we analyze the serverless runtime of major serverless platforms, run benchmarks on major platforms to quantify the costs that remain hidden in the serving layer, and discuss their implications on cost.

3.1 Cost Implications of the Concurrency Model

Serverless platforms vary in how they map concurrent requests to sandboxes. Depending on the maximum number of concurrent requests allowed per sandbox, there are two common serving models: the single-concurrency model, in which the concurrency limit is strictly one (i.e., no intra-runtime concurrency), and the multi-concurrency model, where the concurrency limit can be greater than one.

In the single-concurrency model (e.g., AWS Lambda and Cloudflare Workers), each sandbox accepts only one request at a time. When a request arrives, the serverless platform allocates a new runtime sandbox or reuses a warm, idle sandbox for the request. As there is no resource competition (e.g., CPU and memory) among concurrent requests, this can help keep the execution duration consistent even under high load.

Under the multi-concurrency model, multiple requests can enter and be executed within the same sandbox concurrently, if the user code supports concurrency. Platforms using this model usually allow users to set the maximum concurrency per sandbox and concurrency-based scaling policies. However, *(I6)* **if the extra control knob on concurrency is not configured properly, multi-concurrency can degrade function performance while increasing cost** since resource contention (e.g., CPU, memory, and cache) slows down all concurrent requests and increases billable wall-clock time (e.g., execution time) under request-based billing. For example, running two CPU-bound requests, each requiring 1 s of CPU time, together in a sandbox with 1 vCPU doubles the execution duration of each request to 2 s, thus doubling billable resources as well. In practice, such slowdowns stemming from resource contention are often worse due to context switch overhead and cache misses [96].

To show how concurrency models affect performance and cost, we deploy the same compute-intensive function (PyAES from Functionbench [62]) on AWS Lambda and GCP with 1 vCPU allocation. Each request takes about 160 ms of CPU time. We use the default concurrency configurations (i.e., limit of 80) and scaling policy (60% CPU utilization target and concurrency-based scaling [20]) on GCP. At various request rates (in RPS), we send bursts of requests for 120 s to simulate a short traffic spike. The plot on the left in Figure 6 presents the average execution duration reported by providers. AWS maintains a stable execution time at all request rates due to dedicated sandboxes without resource contention. The average execution time (and cost) of the function deployed on GCP rises by up to 9.65× when the request rate is higher than 6 RPS.

Longer traffic logs reveal a key caveat of multi-concurrency models: it takes time to gather scaling metrics to scale instances to match demand. We send a steady traffic of 15 RPS to the GCP function for 20 minutes. The plot on the right in Figure 6 shows the first five minutes (later data remains stable) of execution time and container count reported by GCP. Scaling does not begin until about 40 s. This is likely due to the fact that platforms with the multi-concurrency serving model usually aggregate the scaling metrics over a time window (e.g., 60 s by default in Knative [65]) to avoid oscillation [6]. The execution duration and instance count remain stable after around 90 s, but the average duration is still 1.43 times higher (i.e., 239.29 ms versus 166.78 ms) than that under the RPS of 1 due to resource contention.

Such a dual penalty of slowdowns and higher bills stemming from the multi-concurrency model is particularly concerning, given that a recent characterization study reports that 93.3% of serverless workloads on IBM Cloud Code Engine used Knative's default container concurrency of 100 as the limit [63, 79]. This suggests that most users either do not optimize, or might be unaware of concurrency settings, highlighting the critical need for relevant control knob tuning tools and concurrency optimization guidance from cloud providers.

3.2 Request Serving Architecture Overhead

Figure 7 depicts three common request serving architectures on major serverless platforms. In the runtime API long polling model (e.g., AWS Lambda [95]), the user provides a handler method (non-HTTP) or a binary executable for processing requests. A runtime program (usually offered by providers) runs inside the sandbox and repeatedly polls the runtime API endpoint over HTTP or RPC in a blocking infinite loop. The retrieved request event is processed by the handler, and the result is posted back to the runtime API before the next poll. In the HTTP server model (e.g., Azure, GCP, IBM, and Knative [7, 26, 30, 64]), the function itself runs an HTTP server on a given port, with the user logic
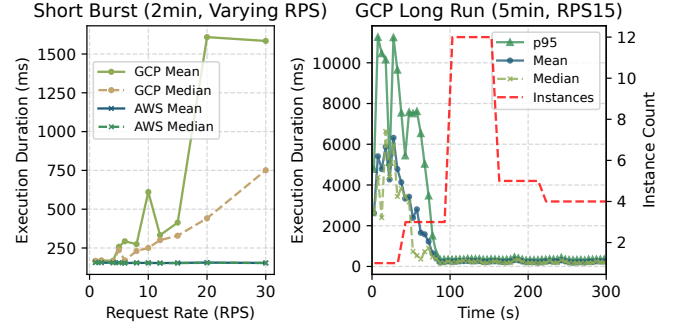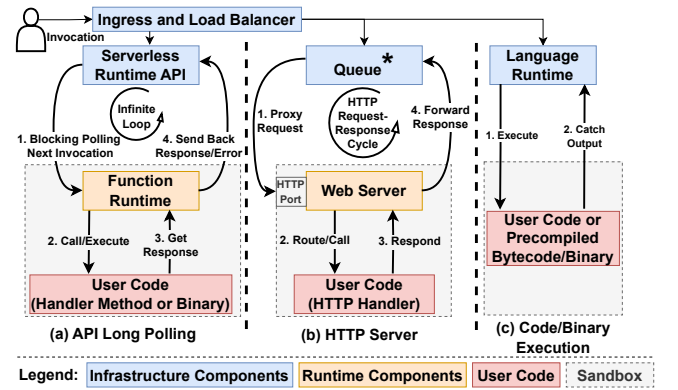

Figure 6. Function execution durations under varying request rates. The multi-concurrency serving model can lead to non-linear slowdowns and increased costs under high concurrency, mainly due to delays in scaling the number of sandboxes to match the incoming request load.

*The queue can be a runtime component as it can be located within the same scaling unit (e.g., pod) or sandbox (e.g., container) and be dedicated to each function instance on some platforms (e.g., GCP, IBM, and Knative).

Figure 7. The three mainstream serverless request serving architectures, including (a) API long polling (e.g., AWS Lambda), (b) HTTP server (e.g., Azure and GCP), and (c) code/binary execution (e.g., Cloudflare Workers).

wrapped in an HTTP handler. The queue (e.g., usually running in a sidecar container) that receives the request from the ingress acts like a reverse proxy and forwards requests to the HTTP server. In the code/binary execution architecture (e.g., Cloudflare Workers), the user uploads a code block or precompiled binary (e.g., Wasm modules [35]). For each request, the language runtime engine (e.g., V8 JavaScript engine) compiles and executes (JIT) or loads and executes the binary, captures the output, and sends back the response.

Each serving architecture has its own benefits. The API polling architecture can avoid exposed ports and simplify the concurrency models by serializing event handling in each sandbox, while the HTTP server model natively supports rich HTTP semantics. Lastly, the code/binary execution architecture generally requires minimal runtime dependencies and artifacts.
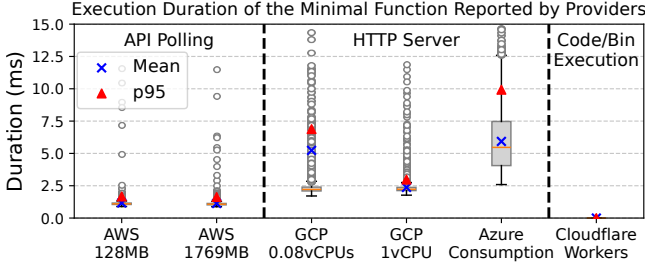
Figure 8. Execution durations of the minimal serverless function across platforms with different serving architectures. The functions with HTTP servers have the highest overhead, while code/binary execution has the smallest.

To quantify the overhead of these architectures, we deploy a minimal serverless function that simply returns an empty string and status code across major platforms. We measure the execution duration of the minimal function that encapsulates pod-/container-level system software. Figure 8 presents the execution duration of the minimal function reported by the providers, which reflects the latency added by the request serving architecture, such as polling request events, HTTP routing, and sending back responses. Our measurements reveal that *(I7)* **platforms using the HTTP server architecture (i.e., GCP and Azure) usually have notably higher overhead, compared to API polling and code/binary execution architectures**, with an average latency up to 5.93 ms. This added latency can impact short functions or round the billable time up to the next interval.

This is due to that functions with the HTTP server model usually host standard HTTP servers as the upstream of the ingress, queue, and/or load balancer, which add overheads, such as maintaining HTTP listeners, connections, thread pools, and handler routing. Also, requests usually traverse additional middleware (e.g., queues, ingress, and/or service mesh sidecars) across containers and veth devices, adding latencies [21, 27, 40, 117]. The resource configuration may also affect such overhead (i.e., GCP 0.08 vCPUs vs GCP 1 vCPU), as the HTTP request–response cycle and HTTP server involve CPU-bound tasks (e.g., header and payload parsing, encoding, and serialization), and a lower CPU allocation can slow these operations. The AWS Lambda functions that use long polling maintain a stable overhead of around 1.17 ms on average. Cloudflare Workers delivers near-zero latency (falling below the precision limit of 0.01 ms reported by Cloudflare), suggesting the high efficiency of the code/binary execution architecture.

3.3 Keep-Alive Duration and Resource Allocation Patterns

Cold start is one of the main causes of performance degradation in serverless functions, and keep-alive has become a common practice to mitigate the cold start latency [97].
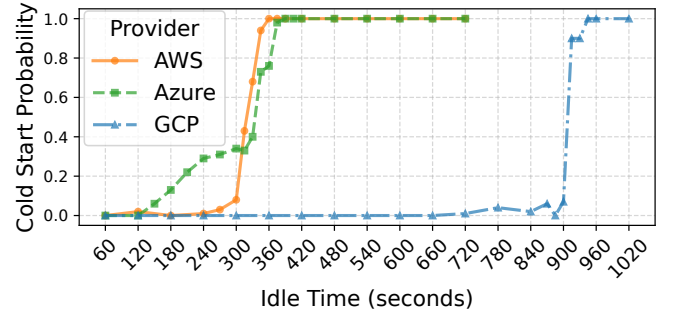

Figure 9. Cold start probabilities versus function idle times. The probabilities of having cold starts increase as the function sandbox idle time becomes longer. The keep-alive durations vary across platforms (as of 2025-05-15).

Major serverless providers keep the user function sandbox active for a period after each request to reduce the chance of cold starts for subsequent invocations. Common keep-alive mechanisms include scale-down delay (e.g., Azure Consumption Plan, GCP, and IBM) [20], container snapshotting [66], code caching (e.g., Cloudflare Workers) [80], and runtime freezing (e.g., AWS Lambda) [94]. Function keep-alive has a direct impact on provider cost, as idle functions can hold active resources (e.g., memory) or reserved capacity (for some schedulers), affecting deployment density. Even techniques that deallocate CPU and memory during the keep-alive phase, such as snapshotting, freezing (e.g., microVM pause), and caching, require CPU time for processing snapshots and cache/storage space. These costs are ultimately passed on to users through per-unit resource pricing or invocation fees.

We deploy serverless functions on major serverless platforms and analyze the keep-alive durations as well as the underlying keep-alive mechanisms. We send requests at different idle intervals to check whether the sandbox is re-created and empirically measure the keep-alive duration. The idle interval is the duration between the end of the previous invocation and the arrival of the next. Figure 9 presents the probability of a cold start as a function of the sandbox idle time, calculated over 100 data points per idle interval. The results show that AWS Lambda keeps the function sandbox alive for up to 300 to 360 s. Azure is likely to use an opportunistic keep-alive strategy, resulting in varying keep-alive durations between 120 s and 360 s. Also, Azure pre-warms the function if the platform detects cold starts occurring at regular intervals (i.e., through idle time histograms) [3, 98]. However, we did not observe such behavior in our experiments, despite regular traffic patterns, as we encountered consistent cold starts at high idle times. This is probably due to the test period being too short for Azure to learn traffic patterns [97]. Besides, Azure may further increase the keep-alive duration for functions with higher traffic and that have been scaled up to multiple instances. We observe a maximum

Serverless Platform	Keep-Alive Phase Behavior	Graceful Shutdown after Keep-Alive
AWS Lambda	Deallocate CPU and memory (Freeze and Resume)	Supported with Lambda Extensions (wait for SIGTERM handling) [87]
GCP Function (Request-Based Billing)	Scale down CPU (to about 0.01 vCPUs)	N/A (kill without SIGTERM)
Azure Function (Consumption)	Run as usual	N/A (kill right after SIGTERM)
Cloudflare Workers	Code/Bytecode Cache	N/A

Table 2. The resource allocation behavior during keep-alive varies across platforms (as of 2025-05-15).

keep-alive duration of around 740 s for the Azure function that has scaled up to 3 instances. In contrast, GCP has the longest keep-alive duration, with the most instances being kept alive for about 900 s. Compared with the data reported in 2018 (e.g., AWS Lambda usually kept functions alive for up to 27 minutes) [109], our observations reveal that **(I8) keep-alive durations on current serverless platforms still vary but have become shorter than previous measurements**, possibly reflecting opportunistic strategies or measures for cost savings.

To examine resource consumption during keep-alive, we run CPU profiling workloads (i.e., Algorithm 1 discussed in §4) and empirically measure the CPU resources available to sandboxes during the keep-alive phase. Table 2 summarizes the resource allocation patterns of the function sandbox during keep-alive and its graceful shutdown behavior when exiting the keep-alive phase and being terminated across platforms. **(I9) The resource allocation behavior during keep-alive varies across platforms, and so do its performance and cost implications for serverless providers and users.**

AWS freezes the sandbox (e.g., puts the microVM to sleep) and resumes it when a new request arrives [61]. Therefore, no active CPU or memory resources are allocated during keep-alive. On GCP, CPU allocation is dynamically scaled down to around 0.01 vCPUs during keep-alive and is scaled back up to the user-configured level when requests arrive within the keep-alive window. Both AWS and GCP deallocate or scale down resources during keep-alive, which naturally saves cost. In contrast, Azure appears to make no change to CPU and memory allocation during keep-alive, which may explain its shorter, opportunistic keep-alive period that reflects a trade-off between resource use and cold start probabilities. Cloudflare pre-warms functions on receiving the TLS handshake before the connection establishment, which can mask the very short loading and JIT compilation latency (e.g., around 5 ms) in case of cold starts [80].

The keep-alive duration and behaviors can directly affect function performance and user costs. A longer keep-alive period can help reduce user costs on platforms that charge based on turnaround time, which includes the cold start latency. Also, keeping resources and sandboxes active during keep-alive (e.g., Azure and GCP) can enable the execution of background tasks and increase the probability of reusing

long-lived, persistent connections. Deallocating resources (e.g., AWS and Cloudflare) may cause remote servers to close connections when they stop receiving heartbeat packets, adding overhead and cost when connections must be re-established. Furthermore, Azure maintains full resource allocation during keep-alive, enabling user-created background tasks to run outside the request execution window. Since Azure Consumption is billed on memory consumption during request execution, resource consumption during keep-alive is not billed (although background tasks can still affect billable wall-clock time and memory usage during request executions due to resource contention) [118]. This may provide opportunities for users to exploit resource allocation. For example, a short request can start a background task running in another thread or coroutine. The background task can send results to other cloud services (e.g., block storage) or remote endpoints after completion, or a subsequent request can retrieve those results. We have successfully implemented this execution pattern on Azure. By doing so, only one or two brief requests are billed, which can substantially reduce overall cloud costs.

4 Impact of OS Scheduling on Performance and Cost

Serverless has a high degree of co-tenancy on servers compared to traditional VM hosting environments [38, 96]. In this environment, the OS kernel plays a crucial role in enforcing resource isolation and fair allocation across workloads with varying limits from different tenants. Common approaches involve fairness-oriented schedulers (e.g., CFS and EEVDF) and control groups (cgroups) for CPU bandwidth control and resource isolation. We observe that when the execution time of a function, the required CPU time, and the billing granularity all fall within the same range as the OS timer tick, scheduling can significantly impact performance and costs. For the first time, we carefully characterize and understand these effects on public serverless platforms.

4.1 Overallocation on Public Serverless Platforms

We deploy a single-threaded, compute-bound serverless function (PyAES from Functionbench [62]) on AWS Lambda under memory sizes ranging from 128 MB (minimum size) to 1,769 MB, and on GCP (first generation is used due to its support for fractional vCPU allocation) under CPU configurations ranging from 0.08 (minimum size) to 1 vCPU. AWS Lambda allocates vCPUs proportionally to the configured memory size, with 1,769 MB equivalent to 1 vCPU [89, 115], while GCP provides a fine-grained CPU control knob with a 0.01 vCPUs increment [24]. Figure 10 shows the execution duration reported by the serverless platform with 900,000 samples in total under different CPU configurations. We make two main observations based on these real-world execution logs.
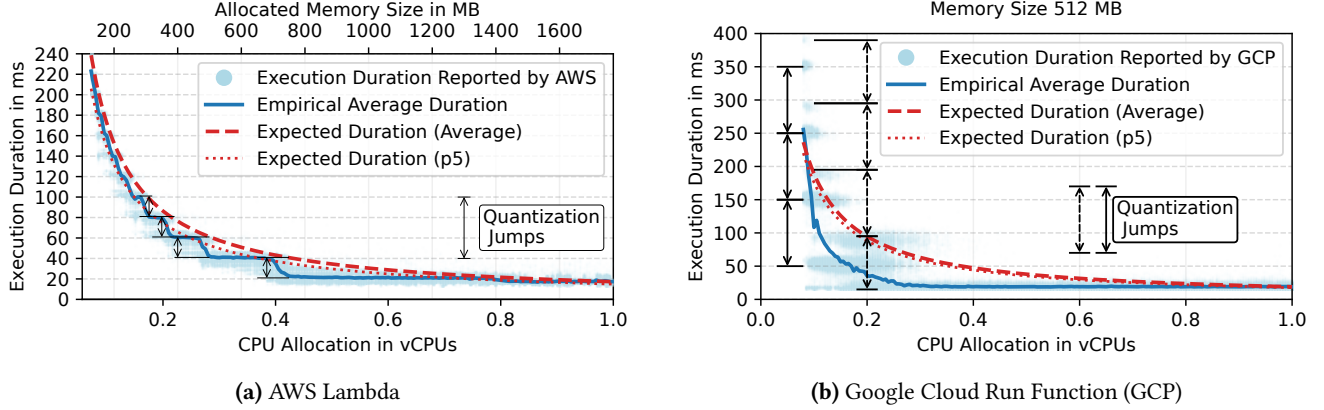
Figure 10. Function execution durations and varying fractional CPU allocations. The difference between the ideal (expected) and actual execution duration shows CPU overallocation for functions hosted on major serverless platforms. GCP logs show two sets of quantization jumps, which may be the cause of CPU scaling down/up when entering/exiting keep-alive phases (§3.3).

First, a single-threaded, CPU-bound workload like PyAES with a fractional vCPU allocation should experience a slow-down of $\frac{1}{\text{vCPUFraction}}$ that follows reciprocal scaling (i.e., half the core allocation, double the execution time). However, the empirical average (solid blue line) is consistently less than the expected average duration (dashed red line) on AWS and GCP (except for a few of the smallest vCPU allocations on GCP). The expected average and expected 5th percentile shown in the figure are based on measurements at full vCPU allocation, scaled proportionally for smaller resource allocations. In other words, a function can ask for, say, half the resources, but be less than twice as slow. In cost terms, this means that users may be charged less than expected under the current wall-clock time billing models presented in Equation (1).

Second, the average empirical execution duration does not have a smooth, reciprocal decline with increasing resource allocations. Instead, it falls with sudden drops, which become less frequent at higher resource allocations. These sudden drops create considerable performance jitters. Also, this means that the allocation-based component in the billing model (i.e., R_{ALLOC} in Equation (1)), also known as the capacity cost, can be reduced by choosing smaller resource limits. We observed this pattern in other functions too, and it is more pronounced with increased compute-boundness.

The performance patterns shown in Figure 10 give us clues into what might be going on. Reducing the resource allocation of the AWS Lambda function from 1 vCPU at first does not affect the performance of the function, but suddenly there are increases at slightly above 1400 MB, 700 MB, 470 MB, 350 MB, 280 MB, and so on. These follow a scaled harmonic sequence: ~$1400 \times \{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, ...\}$. This discrete $\frac{1}{n}$ sequence suggests the presence of a quantization effect,

rather than the continuous proportional allocation ($\frac{1}{x}$) initially expected. Namely, the function is sometimes given more than it is supposed to receive, since the underlying CPU allocation units are quantized, causing jumps on the performance curve. As an analogy, if you want 2 kg of sugar and it is sold in 1 kg packs, the seller gives you two packs. However, if you ask for 1.5 kg, the seller would still need to give you 2 packs, leaving you with an extra 0.5 kg (i.e., overallocation). We observe the same quantization-based overallocation on major serverless platforms.

4.2 Quantized OS Scheduling

By default, the Linux kernel leverages the Completely Fair Scheduler (CFS) or the Earliest Eligible Virtual Deadline First (EEVDF) scheduler (default scheduler since Linux kernel 6.8) to allocate resources in a fair or latency-sensitive manner [39]. The scheduler generally provides each runnable process with a baseline allocation of resources (e.g., CPU time slice), ensuring that it receives at least one opportunity to execute on the processor. It also incorporates mechanisms like CPU Bandwidth Control [104] and cgroups [55] to impose resource limits and provide resource isolation. Such mechanisms have become the foundation of resource isolation and allocation in the sandboxing solutions widely deployed in serverless, such as containers [5, 70], microVMs [2], and Wasm [99]. Our observations in §4.1 are the results of the existing allocation slices in the OS scheduler and cgroups, which seem to be coarse-grained for increasingly short serverless functions, causing issues with cost fairness and performance variability.

The OS maintains a kernel data structure (`cfs_bandwidth`) for bandwidth control of each cgroup (`task_group`), which includes information such as the enforcement period (CFS

period), runtime quota (CFS quota) within each period, remaining runtime available for use (global runtime pool) protected by a spinlock, as well as the throttled run queue [58]. Note that the newer Linux kernels with the EEVDF scheduler use a similar interface and kernel data structure for CPU bandwidth control as CFS. Therefore, the CFS period and quota we discuss in this section also apply to kernels with the EEVDF scheduler. For the rest of the section, we refer to them as (CPU bandwidth control) period and quota.

A high-resolution timer (`hrtimer`) is registered with a callback [56] to refill the global pool with the quota once per period. Each logical CPU core within the cgroup has a local pool of available runtime for per-CPU-basis runtime accounting. During runtime accounting (e.g., at scheduler ticks or context switches), the consumed runtime is subtracted from the local pool for processes running on a core within the cgroup. When the local pool runs out of runtime, it attempts to acquire more (the smaller of `sched_cfs_bandwidth_slice` [54] or remaining runtime) from the global pool. If both the global and local pools are exhausted, processes on the core are throttled and moved to the throttled run queue. When the global pool is refilled to have available runtime in the new period (`hrtimer` callback), the scheduler distributes runtime among throttled run queues and unthrottles them (i.e., marks them as eligible to be scheduled again). Under this schema, the wall clock duration of a CPU-bound process can be calculated as:

$$d = \begin{cases} \left\lfloor T/Q \right\rfloor \times P + T \bmod Q & \text{if } T \bmod Q \neq 0, \\ \left(\left\lfloor T/Q \right\rfloor - 1\right) \times P + Q & \text{otherwise} \end{cases} \quad (2)$$

Here, d is the execution duration, T is the required CPU time, P is the period, and Q is the quota. The scheduler tries to limit the CPU utilization of tasks under CPU bandwidth control to Q/P. Figure 11 shows the execution durations derived by Equation (2) for a CPU-bound workload with a CPU time of 51.8 ms (the average value[3] in Huawei serverless traces [51, 53]) under different periods from 5 ms to 100 ms and the quotas mapped by varying fractional vCPU allocations. These periods are in the same scale compared to those we found empirically (shown later in §4.3). With longer periods, the quantization effect becomes more pronounced. As periods decrease, the execution duration converges to the ideal execution duration following reciprocal scaling.

The model above does not account for the fact that the runtime accounting and throttling mechanisms cannot operate with infinite frequency or precision due to excessive overhead (e.g., handling `hrtimer` interrupts [85]) in real-world systems. Since the scheduling tick frequency is usually between 100 and 1,000 Hz (`CONFIG_HZ`) [82, 86], runtime accounting and task group throttling is often delayed, especially with the relatively long scheduler tick frequency (e.g., 250 Hz or less). Therefore, a task may often consume
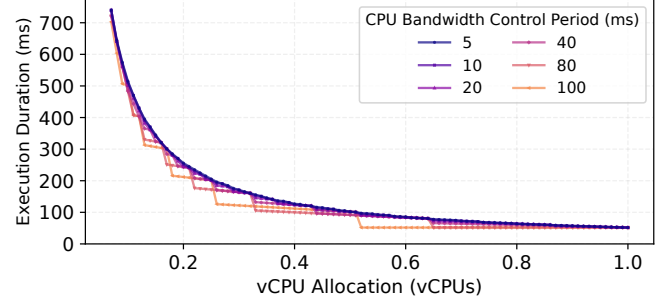
---

[3]The requests that report zero CPU usage are excluded.

Figure 11. Theoretical execution durations under fractional CPU allocations. Shorter CPU bandwidth control periods improve degradation proportionality for sub-core allocations.

runtime more than the quota within a period (overrun) due to lagged accounting, resulting in a negative runtime in the local pool [105]. In this case, the task may be throttled for one or more periods to wait for the quota refill and pay back the runtime debt. For example, consider a CPU-bound task within cgroup with 1.45 ms quota over 20 ms period (i.e., 0.072 vCPU allocated to AWS Lambda with 128 MB memory) and tick interval of 4 ms (250 Hz). A possible scenario is that it first gets 4 ms CPU time and is throttled for 36 ms (rest of the first period and the whole second period) and becomes eligible to run again in the third period (after 40 ms). Then, the task runs another 4 ms after the quota is refilled, causing overrun again with more debt, and is throttled for 56 ms until 100 ms and so on. This task repeatedly alternates between running for 4 ms and being throttled for long periods (i.e., 36 ms or 56 ms) over multiple periods due to overrun and lagged accounting.

Modern kernels often run with the tickless mechanism, with less frequent scheduling interrupts under light loads [59, 100]. Also, scheduling decisions and runtime accounting do not occur only at scheduler ticks. Events like voluntary context switches or interrupts (e.g., `hrtimer`) can also trigger accounting, rescheduling, or preemption. This can lead to variations in runtime allocation and throttled duration. Overrun issues marginally impact long tasks as the OS scheduler ensures fairness over time, but can significantly affect short tasks. However, a defining feature of serverless is the short execution for the majority of requests [52, 53, 97]. Therefore, even without the aforementioned overrun effect, CPU overallocation can still happen if a serverless workload is shorter than the CPU bandwidth control enforcement period. For example, a task that requires 10 ms CPU time running within a cgroup with a 20 ms period of a 10 ms quota is allowed to consume 100% of the CPU during its brief execution, regardless of the configured limit of 0.5 vCPUs. For relatively long tasks that span multiple periods, such overallocation can still happen within the last period before the task is finished. I/O-bound tasks are usually blocked, usually not using CPU while waiting for I/O (e.g., `epoll_wait()`). However, when

Algorithm 1 Profile Runtime and Throttle

1: $s \leftarrow get\_clock\_monotonic()$ ▷ Get monotonic clock time
2: $n\_throt \leftarrow 0$
3: $THRO \leftarrow []$ ▷ Array of tuples of throttle detected time and throttle duration
4: $last\_chkpt \leftarrow s$
5: **while** $true$ **do**
6:     $now \leftarrow get\_clock\_monotonic()$
7:     **if** $now - last\_chkpt \geq 500\mu s$ **then**
8:         $THRO[n\_throt + +] \leftarrow (now, now - last\_chkpt)$
9:     **end if**
10:     $last\_chkpt \leftarrow now$
11:     **if** $now - s \geq EXEC\_DUR$ **then return** $THRO$
12:     **end if**
13: **end while**
---

the task resumes after data becomes available, overruns and throttling across periods may occur, though this is less pronounced as the task uses the CPU intermittently, consuming less runtime and triggering fewer throttles. In a word, **(I10) current OS scheduling granularity seems to be coarse in the context of serverless computing**.

4.3 Scheduling Granularity of Serverless Platforms

The observations and discussions in §4.1 and §4.2 prompt us to further investigate the OS scheduling settings of major serverless platforms and their impact on performance and cost. We analyze three major serverless providers, namely AWS Lambda, GCP, and IBM. However, public serverless providers abstract away infrastructure details and do not expose the underlying scheduling mechanisms and parameters [50, 76]. Therefore, we run functions on target platforms to profile the scheduling behaviors and empirically peek at their scheduling behaviors from the user space.

Methodology: Algorithm 1 presents the pseudocode of the scheduler profile function, in which the function runs for a predefined duration (EXEC_DUR) and records the time and value of sudden increases (>500 μs) in monotonic clock time (CLOCK_MONOTONIC) readings. The default minimal preemption granularity for CPU-bound tasks in the kernel is 750 μs [57], and such time jumps can effectively suggest the occurrence of throttles. We invoke the function with different vCPU configurations, each with 300 invocations. Each function request runs for 10 s, leading to runtime/throttle data collected over 3,000 s of execution span for each configuration. Additionally, to be able to assess the effect of different quotas, periods, and OS schedulers, we use in-house VMs, each with 10 vCPUs (Intel Xeon E5-2673 v4), Linux kernel 6.2 (CFS) or 6.8 (EEVDF scheduler), and the timer frequency of either 250 Hz or 1,000 Hz, to profile the function within containers (runC runtime). We analyze the interval between throttles, the throttle duration, and the consumed CPU time before each throttle by calculating the differences between consecutive events in the recorded data.
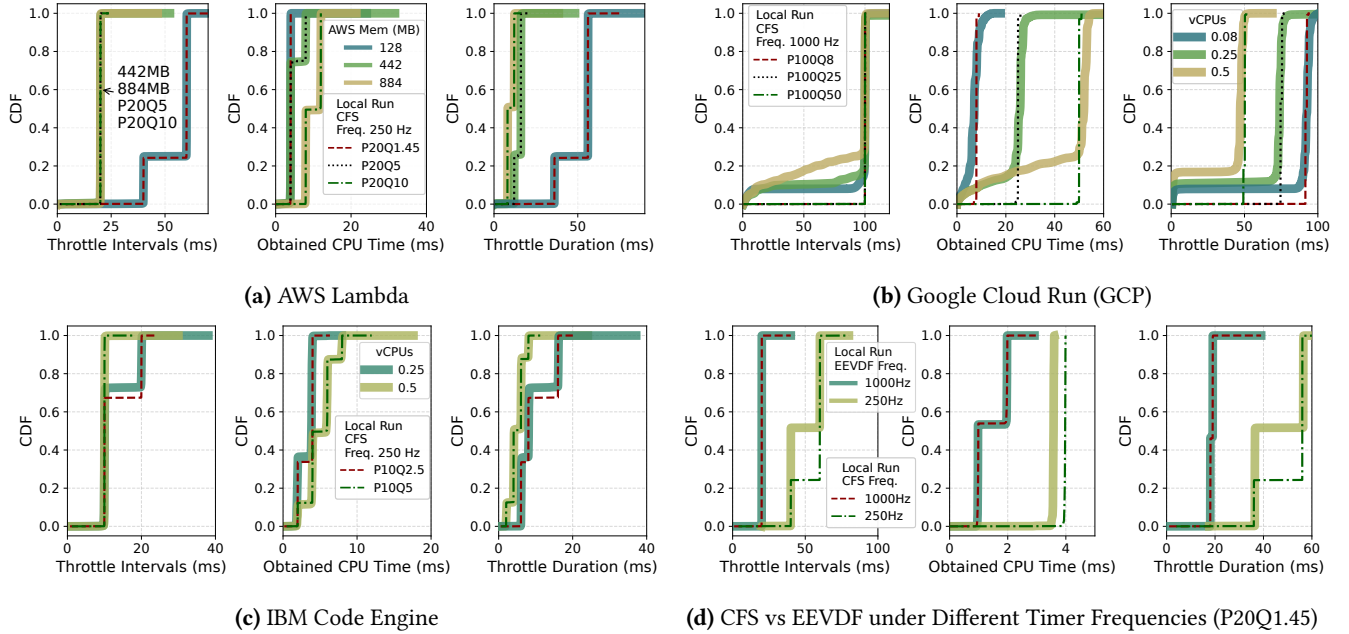
Serverless Platform	Bandwidth Control Period (cfs.cpu_period)	Scheduler Tick Freq (CONFIG_HZ)
AWS Lambda	20 ms	250
Google Cloud Run Functions	100 ms	1000
IBM Cloud Code Engine Functions	10 ms	250

Table 3. Scheduling parameters obtained by empirical analysis (as of 2025-05-15), which vary across different providers.

Empirical Analysis: Figures 12(a) to (c) present the distribution of throttle intervals, durations, and obtained CPU time (runtime) of the studied settings. AWS Lambda functions have throttle intervals that are multiples of 20 ms, whereas IBM functions show multiples of 10 ms. The interval, duration, and runtime results closely align with local runs with corresponding vCPU allocations, periods of 20 ms (for AWS) and 10 ms (for IBM), and the timer frequency of 250 Hz. Also, the runtime and throttle duration of the AWS function (128 MB, 0.072 vCPUs) and their distributions align with the theoretical analysis discussed in § 4.2. The quantized obtained CPU time of AWS Lambda suggests a coarse scheduling granularity under a lower timer frequency (i.e., 250 Hz). The overrun almost happens every time the task is scheduled. Functions on IBM show similar quantized scheduling patterns. The GCP functions exhibit throttle intervals of 100 ms in most cases, while they have 6.42% - 14.83% of throttle durations shorter than 2 ms, indicating frequent context switches and preemption events even within the CPU bandwidth control quota. Compared to other platforms, the less quantized obtained CPU time (i.e., a smoother curve without distinct step-like jumps as shown in Figure 12(b)-Mid) indicates finer-grained time slice allocation under a higher timer frequency. Table 3 presents the scheduling parameters obtained by our empirical analysis, which suggest that public cloud providers do not have a unanimous configuration.

Does the new EEVDF scheduler solve the overallocation issue? The EEVDF scheduler has replaced the CFS scheduler in Linux kernel version 6.8, which introduces a virtual deadline mechanism that improves system responsiveness by prioritizing latency-sensitive tasks with shorter time slices [39, 101]. However, overrun issues still persist under EEVDF because runtime accounting and scheduling granularity remain tied to the timer frequency. As shown in Figure 12(d), when using EEVDF with a 250 Hz timer, the CPU time obtained often exceeds the configured quota, though it is slightly better than CFS with less overrun. Raising the timer frequency to 1000 Hz significantly mitigates the overrun issue. However, even with higher timer frequencies, the fundamental overallocation problem still exists. Whenever required CPU time falls below the quota, overallocation cannot be avoided, regardless of scheduler or timer settings.

Implications: Overrun and overallocation are widespread on public serverless platforms. However, providers can absorb this under-accounted resource usage through currently

(a) AWS Lambda

(b) Google Cloud Run (GCP)

(c) IBM Code Engine

(d) CFS vs EEVDF under Different Timer Frequencies (P20Q1.45)

Note: The dashed and dotted lines are results of local runs with configurations that match the cloud profiling results most. The numbers following P and Q in the legend stand for CPU bandwidth control period and quota in milliseconds. The legend also shows the scheduler and the timer frequency of local runs.

Figure 12. Distributions of throttle intervals, throttle durations, and obtained CPU times (runtime) under the studied scheduling settings. We successfully match the local scheduling setting to cloud deployments. The scheduler profiling results (figures (a), (b), and (c)) reveal that the scheduling settings and granularity vary across serverless platforms.

high invocation fees and coarse billing granularity (rounding up), as discussed in §2.5. For example, a GCP function configured with 0.5 vCPUs and 512 MB memory can potentially consume 100% CPU within 50 ms, but GCP will round its billable wall-clock time up to 100 ms plus a high invocation fee equivalent to 30.19 ms. Also, we tested a user-side exploit on AWS Lambda. We implement an intermittent execution framework and decompose a long function (the video-processing application from SeBS [38]) into a sequence of short bursts, each falling within the quota. We could reduce billable memory GB-seconds by 66.7% on average (calculated over 100 data points). However, because AWS charges a fixed invocation fee, our actual bill increased by 76.7%. In other words, providers that plan to eliminate invocation fees and coarse billing granularity should account for these overallocation effects.

Besides billing, overallocation has clear performance impacts as shown in Figure 10. Users can experience high jitters when vCPU allocations are near quantization boundaries. Existing function-rightsizing tools [42, 71, 78] are agnostic to the quantization effect we described. However, they should be able to capture this effect if equipped with fine-grained, data-driven search. For the first time, we reveal the interplay between scheduling, performance, and billing that these frameworks implicitly use, potentially unlocking more optimal rightsizing strategies.

One potential way to address overrun and overallocation within the serverless computing context is to adopt an event-driven quota enforcement mechanism instead of periodic polling mechanisms based on periodic timers/ticks [103]. For example, one-shot timers that expire upon a function process exhausting its bandwidth control quota may be set to trigger an immediate throttle and reschedule. Also, per-task timers can be set to fire after a short, adaptive time (e.g., depending on the global bandwidth control period, overhead tolerance, accuracy requirements, and predicted task duration) to enforce more frequent and accurate CPU time accounting for short-lived tasks with fractional vCPU allocations. In addition, BPF programs can be attached to the scheduler (e.g., through sched_ext [60]) to selectively apply fine-grained quota enforcement to shorter functions that are more susceptible to overallocation.

5 Discussions

Relative contributions of each cost-related component: In this work, we chose not to quantify the relative contribution of each cost-related component since such numerical breakdowns are highly dependent on context-specific factors. These factors include workload characteristics (e.g., traffic patterns, execution durations, and resource demands), user configurations (e.g., concurrency settings, provisioned

resources [90], and subscription plans [13]), and provider-specific policies (e.g., ARM CPU and committed use discounts and free tiers [83, 92]), which vary across applications and providers. Therefore, any numerical breakdown would not be broadly applicable. Instead, our approach decomposes the inherent sources of cost inefficiencies and presents a systematic analysis framework across multiple abstraction layers from user-facing billing models to OS scheduling, which enables practitioners to measure and rank cost drivers within their own context.

Actionables for serverless users: Our findings lead to several actionable recommendations for reducing serverless costs. First, users can conduct trace-based analysis to pick an appropriate platform whose cost drivers, such as billing practices (§2 and Table 1), concurrency modes (§3.1), serving architectures (§3.2), keep-alive patterns (§3.3), and scheduling granularity (§4.3 and Table 3), best match their workload. Depending on the cost breakdown, users may consider merging similar functions to lower invocation fees, decomposing functions to better utilize resources, or configuring always-ready instances to avoid cold starts [14, 74, 90, 114]. Also, users should be wary of serverless concurrency models and tune control knobs for resources and scaling to avoid the dual penalty of slowdowns and higher bills (§3.1). Furthermore, it is a good practice to tune workload resource demands and fractional vCPU allocations to avoid performance jitters due to coarse OS scheduling granularity (i.e., quantization jumps shown in Figure 10). Lastly, serverless users may also consider the possibility of running background tasks during keep-alive periods (§3.3 and Table 2).

6 Related Work

(1) External characterization of serverless systems: Numerous studies characterized serverless systems from the users' perspective. Some investigated billing practices and cost efficiency of serverless platforms [1, 14, 40, 44, 67, 73, 114]. However, none offered a holistic top-down analysis like our study on how today's serverless billing practices, request patterns, architectural overheads, and OS scheduling translate to inflated user costs. Other works performed cross-platform characterizations of resource allocation patterns and performance variations of serverless offerings [38, 62, 109, 111, 112, 114]. They did not capture some critical factors that greatly impact serverless costs, such as concurrency models, details of resource allocation during keep-alive, and OS scheduling granularity. Moreover, some of their measurements, such as default keep-alive durations and serving architectures, are now outdated due to the rapid evolution of serverless platforms.

(2) Characterization of production serverless workloads: Several provider-led studies characterized serverless workloads running within their systems [52, 53, 75, 97, 108, 115]. These provided valuable insights and enabled the trace-based

analysis in parts of our study (e.g., quantifying inflated billable resources). However, none of these studies delivers an in-depth analysis that correlates the internal platform characteristics (e.g., overheads of architectures and scheduler settings) with the high costs experienced by serverless users. Our work fills this gap by holistically analyzing billing models and practices and measuring architectural overheads and OS scheduling effects, and connecting them to cost implications.

(3) Open-source serverless solutions, such as Knative [64], AWS Runtime API [88], `workerd` [37], and Azure Functions Host [9], enabled us to demystify overheads hidden in modern serverless serving infrastructures, which have not been reported before.

(4) Serverless cost-efficiency optimization: There have been recent studies that leverage dynamic or cooperative scheduling [16, 46, 47, 81, 116], adaptive overcommitment [69, 102], and new billing models [73, 81] to improve the resource utilization or cost-efficiency of serverless systems, effectively reducing costs. These studies are orthogonal to our work, which, for the first time, comprehensively characterizes the interplay of various factors affecting performance and cost in a top-down manner from user-facing billing models to OS scheduling on major production serverless systems and holistically demystifies the high costs of serverless.

7 Conclusion

For the first time, we holistically demystify the high cost of serverless by conducting a comprehensive characterization of driving factors in a top-down manner from user-facing billing models, through architectural overheads, and finally to OS scheduling. We provide novel insights into how current billing practices, request serving architectures, keep-alive resource allocation behaviors, and OS scheduling granularity contribute to the inflated costs of serverless. These insights spark future directions for serverless practitioners on optimizing the cost-efficiency of serverless systems.

Acknowledgments

References

[1] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In Proceedings of the 2017 11th joint meeting on foundations of software engineering. 884–889.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In 17th USENIX symposium on networked systems design and implementation (NSDI 20). 419–434.

[3] Thiago Almeida. 2024. Our latest work to improve Azure Functions cold starts. https://techcommunity.microsoft.com/blog/appsonazureblog/our-latest-work-to-improve-azure-functions-cold-starts/4164500 [Online; accessed May-14-2025].

[4] Korinne Alpers. 2025. Your frontend, backend, and database — now in one Cloudflare Worker. https://blog.cloudflare.com/full-stack-development-on-cloudflare-workers/ [Online; accessed May-4-2025].

[5] The Kubernetes Authors. 2025. Container Runtimes | Kubernetes. https://kubernetes.io/docs/setup/production-environment/container-runtimes/ [Online; accessed May-5-2025].

[6] Microsoft Azure. 2025. Autoscaling guidance - Azure Architecture Center | Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling [Online; accessed April-24-2025].

[7] Microsoft Azure. 2025. Azure Functions custom handlers. https://learn.microsoft.com/en-us/azure/azure-functions/functions-custom-handlers [Online; accessed September-5-2025].

[8] Microsoft Azure. 2025. Azure Functions Flex Consumption plan hosting. https://learn.microsoft.com/en-us/azure/azure-functions/flex-consumption-plan [Online; accessed April-16-2025].

[9] Microsoft Azure. 2025. Azure Functions Host. https://github.com/Azure/azure-functions-host [Online; accessed May-4-2025].

[10] Microsoft Azure. 2025. Azure Functions Premium plan. https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan [Online; accessed August-30-2025].

[11] Microsoft Azure. 2025. Azure Functions scale and hosting | Microsoft Learn. https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale [Online; accessed April-16-2025].

[12] Microsoft Azure. 2025. Azure Functions – Serverless Functions in Computing | Microsoft Azure. https://azure.microsoft.com/en-us/products/functions [Online; accessed May-2-2025].

[13] Microsoft Azure. 2025. Pricing - Functions | Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/functions/ [Online; accessed April-16-2025].

[14] Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The serverless trilemma: Function composition for serverless computing. In Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 89–103.

[15] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23). ACM, New York, NY, USA, 381–397.

[16] Tingjia Cao, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Tyler Caraza-Harter. 2025. Making Serverless Pay-For-Use a Reality with Leopard. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). 189–204.

[17] Alibaba Cloud. 2025. Billing overview - Function Compute - Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/en/functioncompute/fc-2-0/product-overview/billing-overview [Online; accessed April-19-2025].

[18] Alibaba Cloud. 2025. Function Compute: Fully Hosted and Serverless Running Environment - Alibaba Cloud. https://www.alibabacloud.com/en/product/function-compute [Online; accessed April-19-2025].

[19] Alibaba Cloud. 2025. Instance types and usage modes - Function Compute - Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/en/functioncompute/fc-2-0/user-guide/instance-types-and-instance-modes [Online; accessed April-19-2025].

[20] Google Cloud. 2025. About instance autoscaling in Cloud Run services. https://cloud.google.com/run/docs/about-instance-autoscaling [Online; accessed April-24-2025].

[21] Google Cloud. 2025. Architectural overview of Knative serving. https://cloud.google.com/kubernetes-engine/enterprise/knative-serving/docs/architecture-overview [Online; accessed September-5-2025].

[22] Google Cloud. 2025. Billing settings for services | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/configuring/billing-settings [Online; accessed August-30-2025].

[23] Google Cloud. 2025. Cloud Run functions | Google Cloud. https://cloud.google.com/functions [Online; accessed May-2-2025].

[24] Google Cloud. 2025. Configure CPU limits for services. https://cloud.google.com/run/docs/configuring/services/cpu [Online; accessed April-19-2025].

[25] Google Cloud. 2025. Configure memory limits for services | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/configuring/services/memory-limits [Online; accessed April-19-2025].

[26] Google Cloud. 2025. Container runtime contract. https://cloud.google.com/run/docs/container-contract [Online; accessed September-5-2025].

[27] Google Cloud. 2025. Deploying container images to Cloud Run. https://cloud.google.com/run/docs/deploying [Online; accessed September-5-2025].

[28] Google Cloud. 2025. Pricing | Cloud Run | Google Cloud. https://cloud.google.com/run/pricing [Online; accessed May-10-2025].

[29] Huawei Cloud. 2025. FunctionGraph - Run Serverless Code | Huawei Cloud. https://www.huaweicloud.com/eu/product/functiongraph.html [Online; accessed April-19-2025].

[30] IBM Cloud. 2025. Getting started with IBM Cloud Code Engine. https://cloud.ibm.com/docs/codeengine?topic=codeengine-getting-started [Online; accessed September-5-2025].

[31] IBM Cloud. 2025. IBM Cloud Code Engine. https://www.ibm.com/products/code-engine [Online; accessed May-2-2025].

[32] Oracle Cloud. 2025. Cloud Functions | Oracle. https://www.oracle.com/ca-en/cloud/cloud-native/functions/ [Online; accessed April-19-2025].

[33] Cloudflare. 2025. Cloudflare Workers. https://workers.cloudflare.com/ [Online; accessed May-2-2025].

[34] Cloudflare. 2025. Limits Cloudflare Workers docs. https://developers.cloudflare.com/workers/platform/limits/ [Online; accessed May-4-2025].

[35] Cloudflare. 2025. WebAssembly Cloudflare Workers docs. https://developers.cloudflare.com/workers/runtime-apis/webassembly/ [Online; accessed May-4-2025].

[36] Cloudflare. 2025. What is serverless computing? | Serverless definition | Cloudflare. https://www.cloudflare.com/learning/serverless/what-is-serverless/ [Online; accessed May-2-2025].

[37] Cloudflare. 2025. workerd. https://github.com/cloudflare/workerd [Online; accessed May-4-2025].

[38] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In Proceedings of the 22nd International Middleware Conference. 64–78.

[39] Jonathan Corbet. 2023. An EEVDF CPU scheduler for Linux. https://lwn.net/Articles/925371/ [Online; accessed May-5-2025].

[40] Lazar Cvetković, Rodrigo Fonseca, and Ana Klimovic. 2023. Understanding the neglected cost of serverless cluster management. In Proceedings of the 4th Workshop on Resource Disaggregation and Serverless. 22–28.

[41] Datadog. 2023. The State of Serverless 2023. https://www.datadoghq.com/state-of-serverless/ [Online; accessed April-18-2025].

[42] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. In Proceedings of the 22nd International Middleware Conference. 248–259.

[43] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2021. The state of serverless applications: Collection, characterization, and community consensus. IEEE Transactions on Software Engineering 48, 10 (2021), 4152–4166.

[44] Adam Eivy and Joe Weinman. 2017. Be wary of the economics of" serverless" cloud computing. IEEE Cloud Computing 4, 2 (2017), 6–12.

[45] Uwe Fassnacht and Enrico Regge. 2021. IBM Cloud Code Engine. https://community.ibm.com/HigherLogic/System/DownloadDocumentFile.ashx?DocumentFileKey=e10f6c31-c1dd-3cd5-09d2-098d04b6f695 [Online; accessed April-16-2025].

[46] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. 2022. Sfs: Smart os scheduling for serverless functions. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–16.

[47] Yuqi Fu, Ruizhe Shi, Haoliang Wang, Songqing Chen, and Yue Cheng. 2024. ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Functions. In 2024 USENIX Annual Technical Conference (USENIX ATC 24). 19–36.

[48] Shubham Gupta and Jeff Gebhart. Accessed on 2025-05-07.. AWS Lambda standardizes billing for INIT Phase. https://aws.amazon.com/blogs/compute/aws-lambda-standardizes-billing-for-init-phase/

[49] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651 (2018).

[50] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383 (2019).

[51] Artjom Joosen. 2025. Huawei Cloud Production FaaS Trace Data Releases. https://github.com/sir-lab/data-release [Online; accessed May-5-2025].

[52] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How does it function? characterizing long-term trends in production serverless workloads. In Proceedings of the 2023 ACM Symposium on Cloud Computing. 443–458.

[53] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. 2025. Serverless Cold Starts and Where to Find Them. In Proceedings of the Twentieth European Conference on Computer Systems. 938–953.

[54] Linux Kernel. 2015. CFS Bandwidth Control. https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt [Online; accessed Dec-10-2024].

[55] Linux Kernel. 2015. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt [Online; accessed May-5-2025].

[56] Linux Kernel. 2024. linux/kernel/sched/fair.c at v6.12 · torvalds/linux. https://github.com/torvalds/linux/blob/adc218676eef25575469234709c2d87185ca223a/kernel/sched/fair.c#L6279 [Online; accessed January-4-2025].

[57] Linux Kernel. 2024. linux/kernel/sched/fair.c at v6.12 · torvalds/linux. https://github.com/torvalds/linux/blob/adc218676eef25575469234709c2d87185ca223a/kernel/sched/fair.c#L76 [Online; accessed January-4-2025].

[58] Linux Kernel. 2024. linux/kernel/sched/sched.h at · torvalds/linux. https://github.com/torvalds/linux/blob/adc218676eef25575469234709c2d87185ca223a/kernel/sched/sched.h#L405 [Online; accessed January-4-2025].

[59] Linux Kernel. 2024. NO_HZ: Reducing Scheduling-Clock Ticks. https://github.com/torvalds/linux/blob/master/Documentation/timers/no_hz.rst [Online; accessed Jan-10-2025].

[60] Linux Kernel. 2025. Extensible Scheduler Class. https://docs.kernel.org/scheduler/sched-ext.html [Online; accessed September-5-2025].

[61] Daniel Khan. 2021. A look behind the scenes of AWS Lambda and our new Lambda monitoring extension. https://www.dynatrace.com/news/blog/a-look-behind-the-scenes-of-aws-lambda-and-our-new-lambda-monitoring-extension/ [Online; accessed May-14-2025].

[62] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, 502–504.

[63] Knative. 2025. Configuring concurrency. https://knative.dev/docs/serving/autoscaling/concurrency/ [Online; accessed September-16-2025].

[64] Knative. 2025. Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/docs/ [Online; accessed May-4-2025].

[65] Knative. Accessed on 2025-05-07.. Additional autoscaling configuration for Knative Pod Autoscaler. https://knative.dev/docs/serving/autoscaling/kpa-specific/

[66] Yuqiao Lan, Xiaohui Peng, and Yifan Wang. 2024. Snapipeline: Accelerating Snapshot Startup for FaaS Containers. In Proceedings of the 2024 ACM Symposium on Cloud Computing. 144–159.

[67] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 442–450.

[68] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. 2023. Eigen: End-to-end resource optimization for large-scale databases on the cloud. Proceedings of the VLDB Endowment 16, 12 (2023), 3795–3807.

[69] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In Proceedings of the 2023 ACM Symposium on Cloud Computing. 32–47.

[70] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: a lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22). 53–68.

[71] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. 2022. Fine-grained performance and cost modeling and optimization for FaaS applications. IEEE Transactions on Parallel and Distributed Systems 34, 1 (2022), 180–194.

[72] Changyuan Lin and Mohammad Shahrad. 2024. Bridging the sustainability gap in serverless through observability and carbon-aware pricing. ACM SIGENERGY Energy Informatics Review 4, 5 (2024), 120–126.

[73] Fangming Liu and Yipei Niu. 2023. Demystifying the cost of serverless computing: Towards a win-win deal. IEEE Transactions on Parallel and Distributed Systems 35, 1 (2023), 59–72.

[74] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 303–320.

[75] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. Wisefuse: Workload characterization and dag transformation for serverless workflows. Proceedings of the ACM on Measurement and Analysis of Computing Systems 6, 2 (2022), 1–28.

[76] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2022. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. ACM Computing Surveys (CSUR) 54, 11s (2022), 1–36.

[77] MarketsandMarkets. 2024. Serverless Computing Market Size & Trends, Growth Analysis & Forecast. https://www.marketsandmarkets.com/Market-Reports/serverless-computing-market-217021547.html [Online; accessed May-2-2025].

[78] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. 2023. Parrotfish: Parametric regression for optimizing serverless functions. In Proceedings of the 2023 ACM Symposium on Cloud Computing. 177–192.

[79] Nima Nasiri, Nalin Munshi, Simon Moser, Marius Pirvu, Vijay Sundaresan, Daryl Maier, Thatta Premnath, Norman Böwing, Sathish Gopalakrishnan, and Mohammad Shahrad. 2026. In-Production Characterization of an Open Source Serverless Platform and New Scaling Strategies. In Proceedings of the European Conference on Computer Systems (EuroSys '26). ACM.

[80] Ashcon Partovi. 2020. Eliminating cold starts with Cloudflare Workers. https://blog.cloudflare.com/eliminating-cold-starts-with-cloudflare-workers/ [Online; accessed May-10-2025].

[81] Qi Pei, Yipeng Wang, and Seunghee Shin. 2024. Litmus: Fair Pricing for Serverless Computing. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4. 155–169.

[82] Alessandro Pellegrini and Francesco Quaglia. 2015. Time-sharing time warp via lightweight operating system support. In Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. 47–58.

[83] Danilo Poccia. 2023. AWS Lambda Functions Powered by AWS Graviton2 Processor – Run Your Functions on Arm and Get Up to 34% Better Price Performance. https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance/ [Online; accessed September-5-2025].

[84] Konstantinos Psychas and Javad Ghaderi. 2018. Randomized algorithms for scheduling multi-resource jobs in the cloud. IEEE/ACM Transactions on Networking 26, 5 (2018), 2202–2215.

[85] Andrea Righi. 2024. Enable low latency features in the generic Ubuntu kernel for 24.04 - Kernel - Ubuntu Community Hub. https://discourse.ubuntu.com/t/enable-low-latency-features-in-the-generic-ubuntu-kernel-for-24-04/42255 [Online; accessed Jan-10-2025].

[86] Matt Sealey et al. 2013. One of these things (CONFIG_HZ) is not like the others. https://linux-arm-kernel.infradead.narkive.com/sUEd68SK/one-of-these-things-config-hz-is-not-like-the-others [Online; accessed Jan-10-2025].

[87] Amazon Web Services. 2025. Augment Lambda functions using Lambda extensions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-extensions.html [Online; accessed May-14-2025].

[88] Amazon Web Services. 2025. AWS Lambda for Go. https://github.com/aws/aws-lambda-go [Online; accessed May-4-2025].

[89] Amazon Web Services. 2025. Configure Lambda function memory - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html [Online; accessed April-24-2025].

[90] Amazon Web Services. 2025. Configuring provisioned concurrency for a function. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html [Online; accessed September-5-2025].

[91] Amazon Web Services. 2025. Serverless Compute Engine–AWS Fargate Pricing–Amazon Web Services. https://aws.amazon.com/fargate/pricing/ [Online; accessed May-10-2025].

[92] Amazon Web Services. 2025. Serverless Computing – AWS Lambda Pricing – Amazon Web Services. https://aws.amazon.com/lambda/pricing/ [Online; accessed May-10-2025].

[93] Amazon Web Services. 2025. Serverless Function, FaaS Serverless - AWS Lambda - AWS. https://aws.amazon.com/lambda/ [Online; accessed May-2-2025].

[94] Amazon Web Services. 2025. Understanding the Lambda execution environment lifecycle. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html [Online; accessed May-4-2025].

[95] Amazon Web Services. 2025. Using the Lambda runtime API for custom runtimes. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-api.html [Online; accessed September-5-2025].

[96] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). ACM, 1063–1075.

[97] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX annual technical conference (USENIX ATC 20). 205–218.

[98] Mohammad Shahrad, Rordrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, and Ricardo Bianchini. 2020. Characterization and Optimization of the Serverless Workload at a Large Cloud Provider. USENIX PATRONS (2020), 35.

[99] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In 2020 USENIX Annual Technical Conference (USENIX ATC 20). 419–433.

[100] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. 2007. Getting maximum mileage out of tickless. In Proceedings of the Linux Symposium, Vol. 2. Citeseer, 201–207.

[101] Ion Stoica and Hussein Abdel-Wahab. 1995. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22 (1995).

[102] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In Proceedings of the 13th Symposium on Cloud Computing. 78–93.

[103] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2007. Secretly Monopolizing the CPU Without Superuser Privileges.. In USENIX Security Symposium. 239–256.

[104] Paul Turner, Bharata B Rao, and Nikhil Rao. 2010. CPU bandwidth control for CFS. In Linux Symposium, Vol. 10. Citeseer, 245–254.

[105] Odin Ugedal and Rakesh Kumar. 2022. Mitigating Unnecessary Throttling in Linux CFS Bandwidth Control. In 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 336–345.

[106] Vercel. 2025. Usage & Pricing for Functions. https://vercel.com/docs/functions/usage-and-pricing [Online; accessed April-19-2025].

[107] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In Proceedings of the tenth european

conference on computer systems. 1–17.

[108] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21). 443–457.

[109] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In 2018 USENIX annual technical conference (USENIX ATC 18). 133–146.

[110] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic review. ACM Transactions on Software Engineering and Methodology 32, 5 (2023), 1–61.

[111] Jinfeng Wen, Zhenpeng Chen, Federica Sarro, and Shangguang Wang. 2025. Unveiling overlooked performance variance in serverless computing. Empirical Software Engineering 30, 2 (2025), 1–26.

[112] Jinfeng Wen, Zhenpeng Chen, Jianshu Zhao, Federica Sarro, Haodi Ping, Ying Zhang, Shangguang Wang, and Xuanzhe Liu. 2025. SCOPE: Performance Testing for Serverless Computing. ACM Transactions on Software Engineering and Methodology (2025).

[113] Julian Wood. 2024. Serverless ICYMI Q1 2024 | AWS Compute Blog. https://aws.amazon.com/blogs/compute/serverless-icymi-q1-

2024/ [Online; accessed May-2-2025].

[114] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with ServerlessBench. In Proceedings of the 11th ACM Symposium on Cloud Computing. 30–44.

[115] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 724–739.

[116] Yuxuan Zhao, Weikang Weng, Rob van Nieuwpoort, and Alexandru Uta. 2024. In Serverless, OS Scheduler Choice Costs Money: A Hybrid Scheduling Approach for Cheaper FaaS. In Proceedings of the 25th International Middleware Conference. 172–184.

[117] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, et al. 2023. Dissecting overheads of service mesh sidecars. In Proceedings of the 2023 ACM Symposium on Cloud Computing. 142–157.

[118] Wolfgang Ziegler and Matthew Henderson. 2023. Function process lifetime, suspension ... · Issue #2317 · Azure/Azure-Functions. https://github.com/Azure/Azure-Functions/issues/2317 [Online; accessed May-12-2025].