Toward Interference-Aware Scheduling for Serverless Functions via eBPF and Meta-Learning

Yifan Zhang
University of Connecticut
Storrs, CT, USA

Jianchang Su
University of Connecticut
Storrs, CT, USA

Zixu Shen
University of Connecticut
Storrs, CT, USA

Yang Zhou
UC Davis
Davis, CA, USA

Wei Zhang*
University of Connecticut
Storrs, CT, USA

Abstract

Serverless computing, or Function-as-a-Service (FaaS), offers resource efficiency and flexible pay-as-you-go pricing, making it attractive for diverse workloads. However, performance interference—especially in CPU scheduling—remains a key challenge due to shared resource contention, latency variability, and skewed function popularity. Existing schedulers often rely on proxy metrics like CPU utilization, which misalign with user-centric goals such as low latency and strict Service Level Objectives (SLOs).

We propose eMFS, a performance- and interference-aware scheduler that reduces SLO violations in serverless platforms. It combines an eBPF-based latency monitor, a meta-learning model for CPU prediction, and an SLO-guided Multi-Level Feedback Queue (MLFQ) scheduler for priority-based execution. Together, these components enable dynamic, SLO-aware scheduling tailored to serverless workloads.

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

Keywords: CPU Scheduling, Serverless Computing, eBPF, Meta Learning

ACM Reference Format:
Yifan Zhang, Jianchang Su, Zixu Shen, Yang Zhou, and Wei Zhang. 2025. Toward Interference-Aware Scheduling for Serverless Functions via eBPF and Meta-Learning. In *Practical Adoption Challenges of ML for Systems (PACMI '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3766882.3767181

*Corresponding author.

PACMI '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2205-9/25/10
https://doi.org/10.1145/3766882.3767181

1 Introduction

Serverless computing decouples application logic from infrastructure management, enabling simplified deployment and fine-grained resource control. Its adoption is rapidly growing, with over 50% of enterprises projected to adopt it by 2025 [4]. Serverless platforms offer cost efficiency, automatic scaling, and pay-as-you-go billing [8, 28], making them ideal for workloads such as media processing [22], data analytics [18], and machine learning [20].

In this model, developers write stateless functions in high-level languages like Go, Java, or Python [6], while the platform handles deployment and scaling [23, 27]. Users are billed only for the resources consumed, making it efficient — especially for short-lived executions [3]. Ideally, function execution time should closely match turnaround time. However, due to abstraction and resource sharing, serverless platforms often suffer from high latency variability and skewed function popularity [12, 24], causing noticeable delays in latency-sensitive tasks [5].

While contention across resources affects performance, studies show that host-level CPU scheduling is a key contributing factor [7]. Given the bursty and short-lived nature of serverless workloads, traditional schedulers often fail to ensure fairness or responsiveness. Despite this, most platforms still rely on Linux's Completely Fair Scheduler (CFS), which lacks awareness of function heterogeneity or latency sensitivity. As a result, short functions can experience disproportionately long queuing delays [1, 9, 17, 25].

Recent solutions like Kostis [13], Hermod [14], and SFS [7] attempt to prioritize short tasks but still rely on CFS for final scheduling, which reintroduces interference. These limitations underscore the need for an OS-level scheduler that is aware of serverless workload characteristics and provides performance isolation.

Furthermore, many systems use proxy metrics such as CPU utilization for scheduling [21, 30], which poorly reflect end-to-end latency or throughput. In contrast, scheduling

based on real performance metrics better aligns with user-defined Service Level Objectives (SLOs) [2].

Designing such a scheduler involves several challenges: (1) collecting accurate, low-overhead performance data at scale; (2) making timely scheduling decisions that account for latency goals and fairness; and (3) ensuring deployability without kernel changes.

To address these, we propose eMFS, a performance- and interference-aware scheduler for serverless platforms. It incorporates three key components:

- **eBPF-based Performance Monitor:** A lightweight monitor that traces function-level latency using extended Berkeley Packet Filter (eBPF).
- **Meta-Learning Resource Estimator:** A model that predicts per-function CPU needs using workload features such as request rate and recent execution time.
- **SLO-Instructed MLFQ Scheduler:** A user-space Multi-Level Feedback Queue (MLFQ) scheduler that prioritizes functions based on predicted CPU demand and SLO gaps.

2 Background

Serverless Computing: Serverless computing abstracts away infrastructure management, enabling developers to focus on stateless function logic [11]. Platforms automatically manage provisioning, scaling, and fine-grained billing, making them ideal for event-driven workloads. Large-scale traces from Microsoft Azure [24] and Huawei Cloud [12] reveal three key characteristics: (1) execution times vary over seven orders of magnitude, (2) function popularity is heavily skewed, with a small fraction handling most requests, and (3) invocation patterns are bursty yet exhibit stable aggregate trends. These patterns pose significant challenges for fair and responsive CPU scheduling.

eBPF Technology: Extended Berkeley Packet Filter (eBPF) is a kernel-level framework that enables safe, low-overhead execution of custom programs attached to kernel hooks. Unlike traditional BPF, eBPF supports a richer instruction set, more registers, and persistent in-kernel data structures called maps [29]. eBPF allows developers to collect fine-grained performance metrics—such as function-level latency—in real time, without modifying the kernel or disrupting running systems. This makes it a powerful foundation for monitoring and policy enforcement in production environments.

Meta Learning: Meta-learning [10], or "learning to learn," enables models to rapidly adapt to new tasks using prior experience. Unlike supervised or transfer learning, meta-learning generalizes across tasks with minimal data and computation [16]. This property is particularly suited to serverless computing, where workloads vary widely in behavior, resource needs, and latency sensitivity. By training across diverse tasks, meta-learned models can quickly infer CPU requirements in dynamic, heterogeneous environments.

3 Motivation

We investigate how performance interference affects short serverless functions when co-located with long-running ones on the same CPU core. Using several Linux kernel schedulers—including default and latency-optimized policies—we conduct controlled experiments under 80% and 100% CPU utilization to capture both moderate and saturated scenarios.

Our results show that short functions experience noticeable latency inflation even at 80% utilization, indicating that contention arises before saturation. This suggests that CPU-level scheduling—not hardware resource limits—is the primary source of degradation. Moreover, interference severity varies significantly across schedulers, underscoring the impact of scheduling policy. These findings reveal that traditional schedulers, not designed for bursty, latency-sensitive workloads, often fail to provide fair or responsive scheduling in mixed serverless environments.

3.1 Quantifying the Interference

We use FaaSBench [7], built on a parameterized Fibonacci function, to evaluate how conventional schedulers affect short serverless functions. The benchmark exposes two knobs: an integer N for controlling CPU time and a boolean flag IO to simulate I/O-bound behavior.

To isolate CPU scheduling effects, we disable all I/O by turning off the IO flag. System load is controlled by adjusting request intervals and arrival rates to target 80% and 100% CPU utilization—without altering function logic. This setup ensures that observed latency stems from scheduling behavior. Workload patterns are derived from Azure traces [24], maintaining relevance to production deployments.

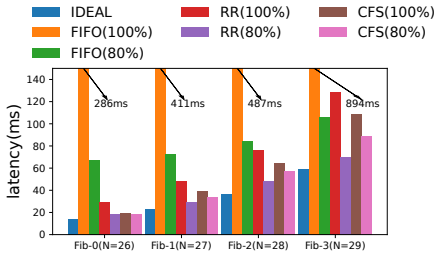


Figure 1. Performance of workloads with different scheduling policies and different loads

In each test, we co-locate a short and a long-running function on the same CPU core to emulate typical serverless scenarios. We evaluate three Linux scheduling policies: the real-time SCHED_FIFO and SCHED_RR, and the default SCHED_NORMAL (CFS). An IDEAL baseline with infinite CPU and no contention serves as the best-case reference.

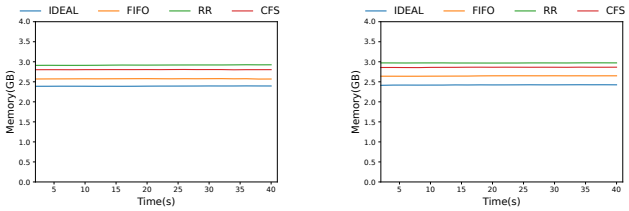
Figure 1 shows the average latency of short functions under 80% and 100% CPU utilization. None of the tested Linux schedulers provides consistently low latency across both conditions. FIFO performs the worst due to head-of-line blocking, while CFS performs best overall but still causes

up to 1.54× latency increase at 80% load and 1.82× at 100%, relative to the IDEAL baseline.

Takeaway 1: Short and long functions co-located on the same core experience interference even before full saturation. Existing Linux schedulers fail to mitigate this, highlighting the need for serverless-aware scheduling strategies.

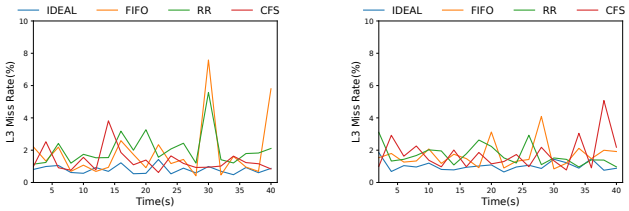
3.2 Understanding the Interference

As noted in § 1, performance interference in serverless systems may stem from contention over shared resources such as CPU, memory, cache, and network bandwidth. To identify the primary cause of latency degradation, we profile runtime metrics—memory usage, L3 cache activity, and network throughput. This analysis helps isolate the dominant bottleneck and confirm whether CPU contention is the main source of interference.



(a) 100% Load **(b)** 80% Load

Figure 2. Memory usage of workloads



(a) 100% Load **(b)** 80% Load

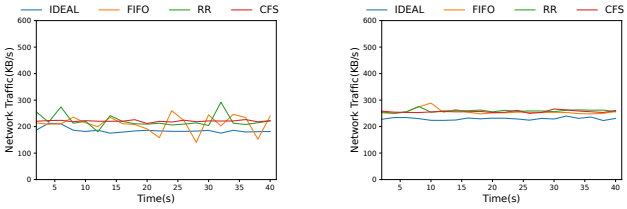
Figure 3. L3 Cache Miss of workloads

Figure 2(a) and Figure 2(b) show memory usage during co-execution at 100% and 80% CPU load. Across all Linux schedulers, memory consumption remains minimal—never exceeding 3 GB, far below the available 64 GB—indicating that memory contention is negligible.

L3 cache behavior in Figure 3(a) and Figure 3(b) reveals consistently low miss rates (<8%) under all schedulers. While FIFO shows some fluctuation due to head-of-line blocking, cache contention is insufficient to explain the observed latency degradation.

Figure 4(a) and Figure 4(b) report network traffic during the same tests. Even under FIFO, throughput remains below 600 KB/s—negligible compared to the 25 Gbps NIC — confirming network bandwidth is not a limiting factor.

Together, these results show that memory, cache, and network are not under significant pressure. The persistent latency interference must therefore stem from the CPU scheduler's inability to handle the bursty, heterogeneous nature of serverless workloads.



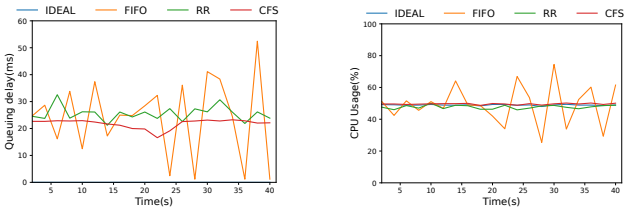
(a) 100% Load **(b)** 80% Load

Figure 4. Network Traffic of workloads

3.3 Effect of the CPU contention

As shown in § 3.2, memory, cache, and network contention do not explain the latency degradation observed in short-term serverless functions. This points to CPU contention as the dominant source of interference.

Figure 5(a) shows average queuing delay under different Linux schedulers. Despite identical load, delays vary widely across policies—indicating that the scheduler itself is responsible. Existing schedulers (FIFO, RR, CFS) lack awareness of function duration or latency sensitivity, leading to unfair scheduling and increased delay for short functions.



(a) Queuing Delay **(b)** CPU Usage

Figure 5. Queuing Delay and CPU Usage

Takeaway 2: CPU contention is the root cause of interference, stemming from schedulers that ignore the bursty and latency-critical nature of serverless workloads.

To address this, we examine whether CPU usage can guide better scheduling. Prior work often uses utilization as a proxy for performance. However, as shown in Figure 5(b), short-term functions receive similar CPU time under RR and CFS, yet experience different queuing delays—revealing that CPU usage alone is not a reliable metric.

Instead, we argue that real-world metrics such as end-to-end latency or queuing delay better capture performance impact and should guide scheduling decisions.

Takeaway 3: Performance-aware metrics provide a more accurate basis for scheduling than raw CPU usage, and better align with user-defined Service Level Objectives (SLOs).

This insight motivates our design of a scheduler that is both latency-aware and interference-resilient.

4 System Design

Building on our earlier motivation, we propose eMFS, a performance- and interference-aware CPU scheduler tailored for serverless platforms. It leverages extended Berkeley Packet Filter (eBPF) for lightweight performance monitoring and meta-learning for adaptive CPU allocation. The core idea is to dynamically adjust CPU time based on two signals: predicted resource needs and the gap between observed performance and target Service Level Objectives (SLOs). eMFS is designed around three system goals:

G1 Lightweight and scalable performance monitoring suitable for high-throughput environments.

G2 Intelligent CPU allocation that reduces SLO violations and minimizes resource waste.

G3 Easy integration with existing serverless platforms, without kernel modifications.

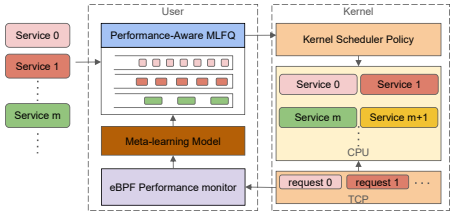


Figure 6. Architecture of eMFS

4.1 eBPF Performance Monitor

The eBPF Performance Monitor forms the foundation of eMFS 's real-time feedback loop. It provides precise, low-overhead latency data for each function, enabling timely and informed scheduling decisions. To estimate execution latency, the monitor captures two key timestamps per request: the arrival time of the last inbound TCP message and the departure time of the first outbound response. The latency is calculated as the time difference between these events, representing the function's processing delay.

We implement the monitor using the BPF Compiler Collection (BCC), attaching eBPF programs to TCP-related kernel functions such as tcp_set_state, tcp_send_msg, and tcp_recv_msg. These hooks enable accurate detection of connection state changes and message boundaries. Each connection is uniquely identified using a five-tuple (source and destination IP and port, plus protocol). To manage memory overhead when tracking many concurrent flows, we apply client-port masking, which maintains connection-level granularity with reduced resource usage.

To minimize runtime overhead, raw metrics are first aggregated and filtered in-kernel. Only relevant, latency-critical data are periodically transferred to user space. There, percentile-based latency summaries (e.g., p95, p99) are computed to detect functions experiencing degradation or SLO violations. These latency signals are then fed into the meta-learning model to guide dynamic CPU reallocation and queue assignment, ensuring responsiveness and efficiency at scale.

4.2 Meta-Learning Model

The Meta-Learning Model in eMFS enables adaptive scheduling by predicting how much CPU each function needs to satisfy its SLO under dynamic workload conditions. It builds upon real-time feedback from the eBPF monitor, using features like request rate, recent execution latency, and resource usage to infer performance bottlenecks.

Traditional models often underperform due to the high degree of heterogeneity and constantly shifting workload patterns [19]. Their reliance on static assumptions makes them brittle in dynamic settings, and maintaining their accuracy typically requires frequent, costly retraining using large volumes of labeled data [15]. This overhead is impractical for real-time platforms that host thousands of ephemeral, short-lived functions with limited observability. In contrast, our meta-learning approach generalizes across a wide range of function-environment pairs (F_i, E_j) and supports few-shot adaptation. This enables rapid deployment and robust prediction, even for previously unseen workloads.

The model architecture comprises a supervised Base Learner and a Meta Learner. The Base Learner maps workload features to predicted function latency. Its training data—collected by the eBPF monitor and stored in an in-memory Data Logger—includes both feature vectors and ground-truth execution times. The Meta Learner learns to generate environment embeddings that capture behavioral patterns specific to each (F_i, E_j) pair, which are then incorporated into the Base Learner's input space.

Unlike traditional hyperparameter-based meta-learning, our design uses embeddings as the primary adaptation mechanism. This allows the Base Learner to effectively transfer knowledge across function types and environments by placing similar execution contexts close together in embedding space. As a result, the model can predict resource-to-performance mappings accurately with limited new data, even in highly dynamic settings.

4.3 SLO-instructed MLFQ Scheduler

To allocate CPU time fairly and responsively across heterogeneous serverless functions, eMFS adopts a Service Level Objective (SLO)-instructed Multi-Level Feedback Queue (MLFQ) scheduler. Traditional MLFQ frameworks define multiple queues with increasing time quanta and rely on runtime behavior to promote or demote tasks [26]. However, in serverless settings, this approach is insufficient: short functions may still queue behind long ones, and long-running functions can be starved. eMFS enhances MLFQ with SLO-based initialization and performance-driven adjustment.

SLO-based Initialization. When a new function arrives, instead of defaulting to the highest-priority queue, we determine its initial queue based on its declared SLO. Queues p_1 to p_6 have increasing time quanta (e.g., 20 ms to 500 ms). A function with a 24 ms SLO may begin in p_2, while a function with a 120 ms SLO would start in p_4. This policy ensures that functions receive time slices proportionate to their latency budgets from the outset, avoiding premature demotions. Within each queue, we adopt Round-Robin (RR) scheduling for fairness. The entire MLFQ logic is implemented in user space and uses Linux's RR policy for actual CPU execution, maintaining compatibility without kernel modification.

Performance-driven Adjustment. Function priorities are adjusted dynamically based on feedback from the eBPF Performance Monitor. If a function consistently violates its SLO, it is promoted to a higher-priority queue; if it maintains latency well below its SLO, it may be safely demoted to make room for more critical tasks. The Meta-Learning Model helps estimate how much additional CPU time would help meet the SLO, and this estimate guides the magnitude and direction of the adjustment.

This feedback loop allows eMFS to react quickly to changing workloads, prioritize latency-critical functions, and avoid starvation for throughput-oriented ones. By combining predictive estimation with real-time measurement, the scheduler achieves high responsiveness and fair CPU distribution in dynamic environments.

5 Conclusion

We present eMFS, a user-space scheduler that is both performance - and interference-aware for serverless computing. It leverages eBPF for real-time, low-overhead monitoring and employs a meta-learning model to predict function-level CPU needs. These insights guide an enhanced SLO-instructed MLFQ scheduler that dynamically adjusts CPU allocation. eMFS is designed for easy integration, adapts quickly to workload heterogeneity, and mitigates interference in diverse serverless environments. By aligning scheduling decisions with real-time performance feedback and SLO targets, eMFS offers a practical path toward responsive and efficient function execution.

References

[1] 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.

[2] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 623–643.

[3] Robert Cordingly, Wen Shu, and Wes J Lloyd. 2020. Predicting performance and cost of serverless computing functions with SAAF. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 640–649.

[4] Katie Costello. [n. d.]. The CIO's Guide to Serverless Computing. https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing.

[5] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[6] 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.

[7] 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.

[8] Jake Grogan, Connor Mulready, James McDermott, Martynas Urbanavicius, Murat Yilmaz, Yalemisew M. Abgaz, Andrew Mccarren, Silvana Togneri MacMahon, Vahid Garousi, Peter Elger, and Paul M. Clarke. 2020. A multivocal literature review of function-as-a-service (faas) infrastructures and implications for software developers. In *Systems, Software and Services Process Improvement: 27th European Conference, EuroSPI 2020, Düsseldorf, Germany, September 9–11, 2020, Proceedings 27*. Springer, 58–75.

[9] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with {OpenLambda}. In *8th USENIX workshop on hot topics in cloud computing (HotCloud 16)*.

[10] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. 2021. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* 44, 9 (2021), 5149–5169.

[11] 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).

[12] 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.

[13] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2019. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM symposium on cloud computing*. 158–164.

[14] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*. 289–305.

[15] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, et al. 2020. {AutoSys}: The Design and Operation of {Learning-Augmented} Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 323–336.

[16] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. 2017. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141* (2017).

[17] OpenFaaS. [n. d.]. https://www.openfaas.com/.

[18] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 131–141.

[19] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Basar, and Ravishankar K Iyer. 2024. FLASH: Fast model adaptation in ML-centric cloud platforms. In *Proceedings of the 7th Annual Conference on Machine Learning and Systems (MLSys 2024)*.

[20] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[21] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[22] Serverless Community Survey: huge growth in serverless usage. [n. d.]. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[23] 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*. 1063–1075.

[24] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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.

[25] Amoghvarsha Suresh and Anshul Gandhi. 2019. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th international workshop on serverless computing*. 19–24.

[26] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).

[27] 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.

[28] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, Antonio Brogi, and Frank Leymann. 2021. Faasten your decisions: A classification framework and technology review of function-as-a-service platforms. *Journal of Systems and Software* 175 (2021), 110906.

[29] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with {eBPF}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1391–1407.

[30] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. {RackSched}: A {Microsecond-Scale} scheduler for {Rack-Scale} computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1225–1240.