FaaS and Curious: Performance implications of serverless functions on edge computing platforms[⋆]

Achilleas Tzenetopoulos, Evangelos Apostolakis, Aphrodite Tzomaka, Christos Papakostopoulos, Konstantinos Stavrakakis, Manolis Katsaragakis, Ioannis Oroutzoglou, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris

National Technical University of Athens, Athens, Greece
`microlab@microlab.ntua.gr`

Abstract. Serverless is an emerging paradigm that greatly simplifies the usage of cloud resources providing unprecedented auto-scaling, simplicity, and cost-efficiency features. Thus, more and more individuals and organizations adopt it, to increase their productivity and focus exclusively on the functionality of their application. Additionally, the cloud is expanding towards the deep edge, forming a continuum in which the event-driven nature of the serverless paradigm seems to make a perfect match. The extreme heterogeneity introduced, in terms of diverse hardware resources and frameworks available, requires systematic approaches for evaluating serverless deployments. In this paper, we propose a methodology for evaluating serverless frameworks deployed on hybrid edge-cloud clusters. Our methodology focuses on key performance knobs of the serverless paradigm and applies a systematic way for evaluating these aspects in hybrid edge-cloud environments. We apply our methodology on three open-source serverless frameworks, OpenFaaS, Openwhisk, and Lean Openwhisk respectively, and we provide key insights regarding their performance implications over resource-constrained edge devices.

Keywords: Serverless-Computing · Edge-Computing · Function-as-a-Service · Cloud · Kubernetes · Openwhisk · OpenFaaS.

1 Introduction

Serverless computing represents the next frontier in the evolution of cloud computing being an emerging paradigm that segregates computing infrastructure from software development and deployment. This results to resource elasticity and seamless scalability combined with lower operational costs. In serverless, also known as Function-as-a-Service (Faas), short-lived, stateless, event-driven functions are usually triggered by various sources. Today several public cloud

[⋆] This work is partially funded by the EU Horizon 2020 research and innovation programme, under project EVOLVE, grant agreement No 825061 and under project AIatEDGE , grant agreement No 101015922

vendors already support serverless, e.g., AWS Lambda (Amazon) [2] and Google Cloud Functions [3], while the global market size of serverless is projected to triplicate by the end of 2025 [8].

While originally designed for the cloud, the benefits of serverless architectures have the potential to be employed in edge computing environments, where computational resources and applications are distributed across the edge-cloud continuum [16]. In fact, serverless shares many common principles with the edge computing paradigm, which allows for low-latency response and Quality-of-Service (QoS) guarantees to event triggers, by collocating computing resources closer to the source of data. Moreover, the development of lightweight containers [20], orchestrators [4, 26] and serverless frameworks [12] for edge devices also paves the way towards the consolidation of edge and serverless paradigms.

Nonetheless, the potential benefits of such a combination can be obtained only if efficiently designed, implemented and deployed. Otherwise, the energy- and computation-limited nature of edge computing devices can lead to inconsistent and unreliable systems [10]. While previous research works have put effort on characterizing the performance of serverless infrastructures and identifying potential bottlenecks and limitations, they rely mostly on serverless solutions provided by public cloud vendors [17, 25] or custom deployments on high-end computing resources [18, 24]. However, the extreme heterogeneity both in terms of hardware resources, as well as serverless solutions available, found in edge/cloud environments requires systematic methodologies for identifying and evaluating the implications of such deployments on the performance of the system.

In this paper, we propose a methodical approach for the evaluation of serverless frameworks on hybrid edge-cloud infrastructures. Our methodology takes into account key performance knobs of serverless frameworks and exposes their implications on resource-constrained edge deployments. We apply our methodology on three open-source serverless frameworks, i.e., Apache Openwhisk [11], Lean Openwhisk [12] and OpenFaaS [6] deployed on top of a hybrid cluster, combining both high-end servers and resource-constrained edge devices.

2 Background & Related Work

Although cloud computing has been significantly improved over the last years, its full-potential has not been released yet. Cloud users continue to bear a burden from resource provisioning operations and the pay-as-you-go promise has not yet been fulfilled by the traditional cloud offerings. *Serverless computing* aims to overcome these limitations by introducing a new layer of abstraction to developers, i.e., remove the burden of server management from the end user (-less), delegating them to the platform.

From the developers' point of view, responsibilities are limited to the source code submission in the serverless platform; they are relieved from pre- and post-development tasks (e.g., infrastructure setup, resource provisioning). In addition, the key economic incentive for clients stems from the cost savings due to fine-grained billing (e.g., $1ms$ billing granularity in AWS Lambda). Scaling down to

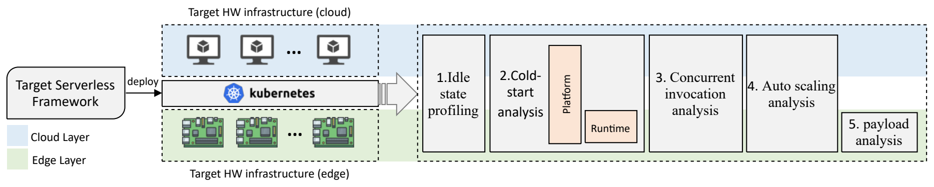


Fig. 1: Proposed methodology for systematic analysis of serverless infrastructures

zero, combined with the demand-driven resource elasticity, precludes clients from paying for idle resources. Providers, on the other hand, are given the opportunity to maximize their data-centers' utilization, by performing fine-grained resource multiplexing due to the short run time and stateless nature of functions.

While previous research efforts have examined the performance implications of serverless frameworks [13, 17, 24, 25, 27], these works mainly focus on serverless deployments on the cloud, thus, neglecting the potentials of serverless deployment on the edge. *Compared to conventional cloud serverless deployments, serverless on the edge reveals new challenges that need to be undertaken [10].* Several research works have designed custom, lightweight frameworks, designated for the edge, by either utilizing WebAssembly [14, 15] or application level isolation through multi-threading [21]. Authors in [23] optimize existing open-source frameworks and furtherly encourage [22] the serverless on edge paradigm adoption. However, these approaches either refer to single-node deployments or neglect the heterogeneity found in hybrid edge-cloud infrastructures.

3 Systematic analysis of serverless infrastructures

In this section, we describe our proposed methodology for systematically evaluating serveless infrastructures over hybrid edge-cloud deployments. This methodology, illustrated in Fig. 1, includes the evaluation of key features for serverless platforms at the edge.

3.1 Proposed Methodology

Our systematic way of evaluating serverless frameworks relies on key enablers and open challenges that characterize such infrastructures. More specifically, our methodology consists of 5 evaluation steps (Fig. 1), which are described below:

1. Idle state profiling: Due to the inefficient and insufficient resources found at the edge, idle-state resource utilization is a key factor for a platform's evaluation. As a first step of our approach, we examine the additional overhead introduced by the target platform when deployed on the underlying infrastructure.

2. Cold-start Analysis: In existing serverless platforms, delays incurred during function instantiation can lead to significant execution time overheads, compared to native execution. From the platform's point of view, this additional latency includes a) the time required to start the sandbox and b) the time required for runtime initialization. While there have been different sandboxes proposed in academia [19] and industry [9] for lightweight virtualization, in this study, we focus on performance characteristics of Docker containers, which currently form the most typical way of deploying applications to the cloud. From application's perspective, this latency delay occurs due to the different runtimes of modern programming languages, e.g., Node.js, Python, Go, each of which induces different performance overheads. This variability may get even worse when those functions are deployed on heterogeneous edge devices.

3. Concurrent invocation analysis: Serverless functions hosted on edge devices may be invoked concurrently by multiple clients or triggers. Therefore, bottlenecks on different components of a serverless platform may incur latency on invocations' end to end execution time. Thus, this step evaluates the latency distribution on different levels of invocation concurrency.

4. Auto-scaling analysis: In order to minimize costs, (e.g., energy, billing) serverless platforms need to provide elastic scalability. At the same time they need to address bursts on invocation frequency efficiently. We evaluate the responsiveness of serverless platforms on different invocation per second intensities.

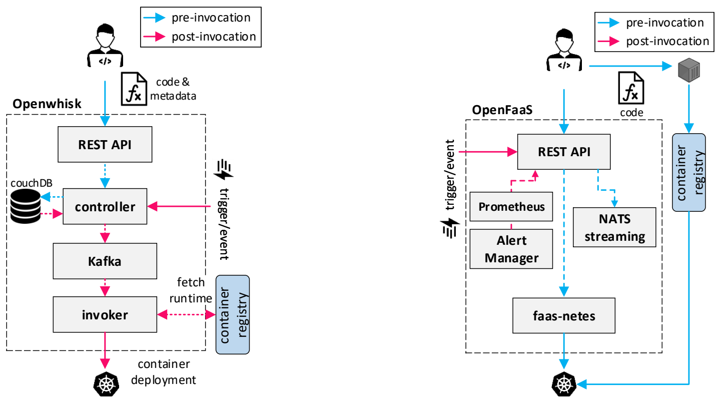
5. Payload analysis: In serverless computing, storage and computation are decoupled. Thus, fine-grained state sharing between application becomes difficult. Most platforms utilize external object storage services, like AWS S3, which induce additional costs and latency overhead on data sharing, especially on edge environments with limited network bandwidth. As a next step of our methodology, we measure the delays provoked by payload transfer between functions through the framework gateway.

3.2 Target serverless frameworks

Our methodology can be applied for evaluating open-source serverless frameworks that support deployment over container orchestration frameworks (e.g., Kubernetes) that manage resource-constrained edge devices, as well as conventional x86 machines. For the purposes of this work, we focus on three open-source serverless frameworks, i.e., Apache Openwhisk [11] and OpenFaaS [6], as well as a lightweight version of Openwhisk (Lean Openwhisk [12]) specifically designed for edge computing environments. Since Openwhisk currently supports only x86 machines, we modified and recompiled the necessary components and runtimes to address deployments on `aarch64` architectures. Below, we provide an overview of the operation mechanisms for each one of the aforementioned serverless frameworks.

Openwhisk architecture overview: Apache Openwhisk is an open-source, distributed serverless platform, initially developed by IBM. In Openwhisk, developers register their functions, also referred to as `Actions`, which can be triggered either from associated events, external sources, or HTTP requests. Moreover, `Triggers` provide endpoints that can be triggered by event sources, such as databases, stream processing engines and others. Finally, through `Rules`, developers create loosely coupled associations between them.

From an architectural point of view, Openwhisk relies on four main components for handling and executing function codes (Fig. 2a). First, Openwhisk



(a) Openwhisk (b) OpenFaaS

Fig. 2: Architecture overview

exposes a public RESTful API which can be reached by developers to register their Actions, Triggers and Rules. After a request passes through the API, it triggers the `controller` component, which acts as the governor of the system. The controller communicates with a database instance (`couchDB`), which maintains and manages the state of the overall system and keeps information regarding credentials, metadata, namespaces as well as the definitions of Actions, Triggers and Rules registered by developers. Once an event triggers a new invocation, the `controller` after authenticating the invocation request, retrieves the function code, and selects the most appropriate node (`Invoker`) to handle the request. Afterwards, it publishes a message containing the code along with invocation-related meta-data, (e.g., resource allocation, input arguments) to Apache Kafka [1]. Finally, the `Invoker` builds the function code, encapsulates it in a predefined runtime container, initializes and executes it.

Lean Openwhisk architecture overview: Lean Openwhisk [12] is a customized, downsized distribution of Openwhisk which, however, shares the same design principles. By replacing Kafka with an in-memory queue, and compiling jointly some other parts, Lean Openwhisk is designed to enable the serverless paradigm within resource-constrained edge devices. Yet, to the best of our knowledge Lean Openwhisk only supports single-node setups.

OpenFaaS architecture overview: OpenFaaS (Fig. 2b), unlike Openwhisk, utilizes a container orchestrator, e.g., Kubernetes, to manage the lifecycle of the containers through a custom controller. An end-to-end workflow starts with a call to the OpenFaaS API which interacts with Kubernetes objects (Pod, Deployment, Service) leveraging the OpenFaaS controller (`faas-netes`). According to the default settings, functions auto-scale up or down depending on the requests per second, by utilizing Prometheus [7] alert manager and monitoring. In addition, OpenFaaS utilizes a message bus for asynchronous function invocation. Compared to Openwhisk, OpenFaaS does not dynamically pack and execute code at runtime. Instead, developers have to pre-define containers containing their function code and are always up and running on the cluster by default.

Table 1: VMs and edge nodes specifications

	Server	**Agent 1**	**Agent 2**	**Rasp. Pi 3b+**	**Rasp. Pi 4b**
Processor	Intel®Xeon®	Intel®Xeon®	Intel®Xeon®	Cortex-A53	Cortex-A72
Model	E5-2658A v3	E5-2658A v3	Silver 4210	(ARMv8)	(ARMv8)
Cores	4(vCPUs)	4(vCPUs)	8(vCPUs)	4	4
RAM(GB)	8	4	8	1	4

These containers are augmented by OpenFaaS with an additional process, called `watchdog`. The watchdog is responsible for processing incoming event triggers and also for initializing and monitoring the functional logic of the container.

3.3 Target cluster infrastructure

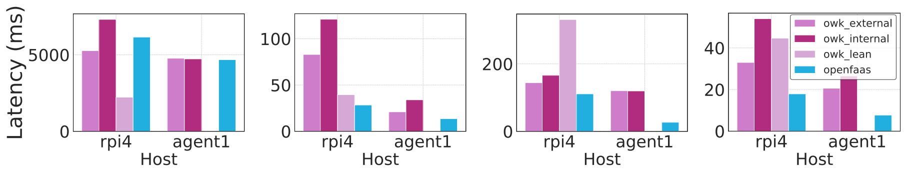
Our experiments have been performed on a distributed cluster, that consists of VMs deployed on top of high-end servers as well as typical, resource-constrained edge devices, the specifications of which are outlined in Table 1. Moreover, we utilize K3s as our container orchestrator, which is a lightweight distribution of conventional Kubernetes, built for IoT and edge computing devices.

4 Evaluation

In this section we apply the profiling and analysis steps of our proposed methodology to assess and evaluate our target serverless frameworks (sec. 3.2) over our cluster infrastructure (sec. 3.3). For the purposes of our experiments, we intentionally place all the required components for the frameworks' functionality, (e.g., Apache Kafka, CouchDB, Prometheus), on the Server node, which is dedicated to orchestration rather than workload execution. Therefore, resources of VM agents and edge devices (Raspberry Pis) are exclusively exploited by the scheduled applications. Additionaly, in order to evaluate the performance of Openwhisk on edge devices, we place the Invoker component either externally (Agents) or internally (RPi4), utilizing the *Kubernetes container factory*, which allows the placement of Openwhisk to be managed solely by Kubernetes.

Idle-state profiling: Regarding the resource consumption of our container orchestrator, K3s introduces neglectable utilization. Its resource utilization sums up to 10 millicpu and 200MB of RAM on average. Openwhisk, due to its complicated components, has considerable resource needs. While the agents need 600 millicpu and 161MB of RAM when hosting the Invoker, its main components deployed on the Server node contribute 1800 millicpu and 2.4GB of RAM jointly. The Lean, single-node version of Openwhisk uses 5 millicpu and 218.93 MB of RAM on Rpi4. OpenFaaS, on the other side, being closely integrated with Kubernetes, contributes a modest footprint on the Server node that sums up to 66 millicpu and 83MB of RAM on the Server node. Agent and edge nodes experience negligible overhead.

Cold-start Latency analysis: The resource-constrained edge devices increase the latency of container fetching, creation and initialization.



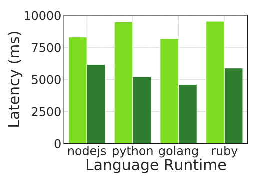
(a) Cold init. (b) Cold exec. (c) Warm init. (d) Warm exec.

Fig. 3: Per platform Cold and Warm start latency breakdown

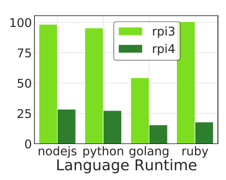
Platform dependence: Fig. 3 illustrates the cold/warm initialization and execution time of a simple Node.js function on Rpi4 and on Agent1. As initialization latency, we define the time elapsed between function invocation and the runtime execution. In every platform, while we observe higher latency on the edge node, the impact of function initialization remains tremendous. As mentioned before (Fig. 2a), since Openwhisk alleviates more tasks (e.g., source code injection), additional post-invocation overhead is added. However, when the Invoker is hosted externally, Openwhisk post-invocation procedure is accelerated and results in decreased instantiation and execution time. Lean Openwhisk supporting only Node.js-6 runtime presents the less overhead on coldstart function instantiation. Yet, since the operational logic of the framework is embedded to a single edge node, the warm times are increased. On the contrary, on warm invocations, we observe decreased latency on OpenFaaS, which requires more pre-invocation tasks, e.g., function container deployment, (Fig. 2b) from the user.

Runtime dependence: In the OpenFaaS platform, which offers greater flexibility on multi-arch runtimes, we deploy a `helloworld` function on different language runtimes to Rpi3 and Rpi4 to examine the invocation latency breakdown. We execute each experiment 5 times for consistency, and the average latency is illustrated in Fig. 4. Cold start latency is high for every language runtime. The difference for function initialization between the warm and cold start is 8 seconds for the Rpi3 and 5 seconds for the Rpi4, respectively. Therefore, the inefficient and heterogeneous resources at the edge may vary in the latency incurred during container fetching, creation and initialization. In the warm start cases, latency is decreased up to 4.3x in Rpi4. Comparing the alternative programming languages, Python and Ruby runtimes provoke the greatest latency even in the warm start cases (Fig. 4c, 4d), while Golang seems to offer the modest latency footprint among them. Finally, except for the performance variability in function instantiation between cold/warm start and edge device, similar results are observed in function execution time. Post-initialization (exec.) latency is smaller after a warm compared with a cold start. Possible reasons for this phenomenon may be trained branch predictors, or cache locality.

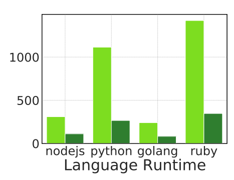
Concurrent Invocation Analysis: Fig. 5 depicts the distribution of 120 warm invocations of an `Optical Character Recognition` (OCR) (164KB *png* image) function in Node.js on Rpi4, when invoked concurrently by multiple sources. Invocations were generated using the `loadtest` [5] tool on the Server. In



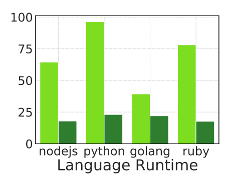
(a) Cold init.



(b) Cold exec.



(c) Warm init.



(d) Warm exec.

Fig. 4: Per language runtime Cold and Warm start latency breakdown

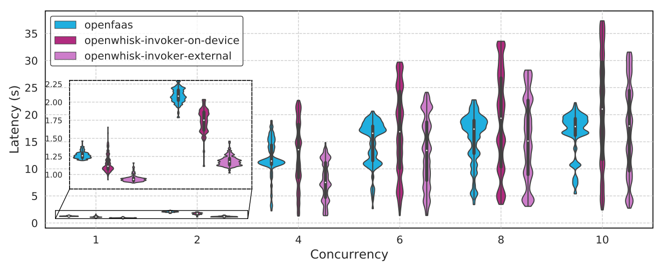


Fig. 5: Applications relative performance in different workload density.

both platforms, latency increases drastically when 4 or more invocations occur concurrently. The high standard deviation of Openwhisk workload distribution depicts the accumulated congestion occurring in its complicated pipeline after the function invocation. Thus, there is a great performance improvement when the Invoker is offloaded externally (Agent2). While OpenFaaS provides a more robust distribution for higher numbers of concurrent requests (32% lower 90th percentile latency for 10 requests), it is outperformed by Openwhisk for lower numbers of concurrent requests which delivers lower median end-to-end latency.

Auto-Scaling: In order to evaluate the auto-scaling, we invoke the OCR application with 1,2 and 3 invocations per second (ips). This time, instead of the default OpenFaaS auto-scaler which utilizes Prometheus monitoring, we employ the Horizontal Kubernetes Auto-scaler. We assign 500 millicpus per function and define 75% as the limit that must be exceeded before scaling up. In Fig. 6 is illustrated the latency of Openwhisk (external Invoker) and OpenFaaS overtime on a Rpi4 for different densities. While Openwhisk provides gradual, finer-grained scalability, additional latency is built-up overtime due to its complex components. Moreover, another inefficiency observed is that invocation requests are assigned to function replicas before they are instantiated. OpenFaaS scales the function much less aggressively, but it applies more efficient load balancing.

Payload Transfer: Fig. 7 illustrates the payload transfer latency distribution of 10 experiment repetitions for data sizes 1-80KB on Rpi4. Again, Openwhisk requires increased latency on routing and passing the request to the message queue. However, OpenFaaS and Lean Openwhisk incur modest latency on

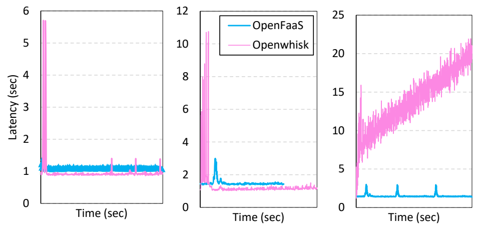


Fig. 6: Auto-Scaling: 1, 2 and 3 ips

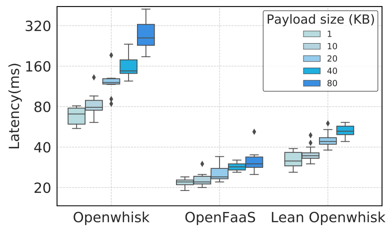


Fig. 7: Payload Transfer

data transfer; therefore local state passing to avoid data transfers from the serverless edge environments to the Cloud, forms a promising field for further study.

5 Conclusion and Future Work

This paper proposes a systematic methodology for evaluating serverless platforms at hybrid edge-cloud infrastructures. We apply our methodology by deploying a Kubernetes cluster on top of an heterogeneous pool of devices and evaluate three of the most widely used open-source serverless platforms, i.e., OpenFaaS, Openwhisk and Lean Openwhisk. As future steps, we aim to extend our methodology for applying to real-world serverless applications and examine the performance implications of serverless frameworks on function chains and workflows, as well as to investigate function orchestration schemes in the edge/cloud computing continuum.

References

1. Apache kafka. Website, https://kafka.apache.org/
2. Aws lambda. Website, https://aws.amazon.com/lambda/
3. Google cloud functions. Website, https://cloud.google.com/functions
4. Lightweight kubernetes - k3s. Website, https://k3s.io/
5. Loadtest. Website, https://github.com/alexfernandez/loadtest
6. Openfaas. Website, https://www.openfaas.com/
7. Prometheus. Website, https://prometheus.io, last checked: 07.04.2021
8. Serverless architecture market. Website, shorturl.at/gGMU5
9. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.M.: Firecracker: Lightweight virtualization for serverless applications. In: 17th {usenix} symposium on networked systems design and implementation ({nsdi} 20). pp. 419–434 (2020)
10. Aslanpour, M.S., Toosi, A.N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., Assuncao, M., Gill, S.S., Gaire, R., Dustdar, S.: Serverless edge computing: vision and challenges. In: 2021 Australasian Computer Science Week Multiconference. pp. 1–10 (2021)
11. Baldini, I., Castro, P., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P.: Cloud-native, event-based programming for mobile applications. In: Proceedings of the International Conference on Mobile Software Engineering and Systems. pp. 287–288 (2016)

12. Breitgand, D.: Lean openwhisk. Website, https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b, last checked: 07.04.2021
13. Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., Hoefler, T.: Sebs: A serverless benchmark suite for function-as-a-service computing. arXiv preprint arXiv:2012.14132 (2020)
14. Gadepalli, P.K., Peach, G., Cherkasova, L., Aitken, R., Parmer, G.: Challenges and opportunities for efficient serverless computing at the edge. In: 2019 38th Symposium on Reliable Distributed Systems (SRDS). pp. 261–2615. IEEE (2019)
15. Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236 (2019)
16. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al.: Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383 (2019)
17. Lee, H., Satyam, K., Fox, G.: Evaluation of production serverless computing environments. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). pp. 442–450. IEEE (2018)
18. Li, J., Kulkarni, S.G., Ramakrishnan, K., Li, D.: Understanding open source serverless platforms: Design considerations and performance. In: Proceedings of the 5th International Workshop on Serverless Computing. pp. 37–42 (2019)
19. Nikolos, O.L., Papazafeiropoulos, K., Psomadakis, S., Nanos, A., Koziris, N.: Extending storage support for unikernel containers. In: Proceedings of the 5th International Workshop on Serverless Computing. pp. 31–36 (2019)
20. Park, M., Bhardwaj, K., Gavrilovska, A.: Toward lighter containers for the edge. In: 3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20) (2020)
21. Pfandzelter, T., Bermbach, D.: tinyfaas: A lightweight faas platform for edge environments. In: 2020 IEEE International Conference on Fog Computing (ICFC). pp. 17–24. IEEE (2020)
22. Rausch, T., Hummer, W., Muthusamy, V., Rashed, A., Dustdar, S.: Towards a serverless platform for edge {AI}. In: 2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19) (2019)
23. Rausch, T., Rashed, A., Dustdar, S.: Optimized container scheduling for data-intensive serverless edge computing. Future Generation Computer Systems **114**, 259–271 (2021)
24. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 1063–1075 (2019)
25. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). pp. 133–146 (2018)
26. Xiong, Y., Sun, Y., Xing, L., Huang, Y.: Extend cloud to edge with kubeedge. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC). pp. 373–377. IEEE (2018)
27. Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C., Chen, H.: Characterizing serverless platforms with serverlessbench. In: Proceedings of the 11th ACM Symposium on Cloud Computing. pp. 30–44 (2020)