Serverless Computing: State-of-the-Art, Challenges and Opportunities

Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu, *Fellow, IEEE*

Abstract—Serverless computing is growing in popularity by virtue of its lightweight and simplicity of management. It achieves these merits by reducing the granularity of the computing unit to the function level. Specifically, serverless allows users to focus squarely on the function itself while leaving other cumbersome management and scheduling issues to the platform provider, who is responsible for striking a balance between high-performance scheduling and low resource cost. In this article, we conduct a comprehensive survey of serverless computing with a particular focus on its infrastructure characteristics. Whereby some existing challenges are identified, and the associated cutting-edge solutions are analyzed. With these results, we further investigate some typical open-source frameworks and study how they address the identified challenges. Given the great advantages of serverless computing, it is expected that its deployment would dominate future cloud platforms. As such, we also envision some promising research opportunities that need to be further explored in the future. We hope that our work in this article can inspire those researchers and practitioners who are engaged in related fields to appreciate serverless computing, thereby setting foot in this promising area and making great contributions to its development.

Index Terms—Survey, serverless computing, FaaS and BaaS, startup latency, isolation, scheduling

1 Introduction

Serverless computing as a new execution model of cloud computing has gained great popularity in recent years due to its lightweight and simplicity of management. In serverless computing, developers only need to write cloud functions in high-level languages (e.g., Java, Python), set several simple parameters, and upload these functions to a serverless platform. Then, they could use the returned API or HTTP requests to execute their well-defined computation tasks. As opposed to serverful computing models, developers using serverless do not need to care about the management of infrastructure resources as platforms shield such details on behalf of them.

The *cloud functions* — written by developers and invoked through the Internet as units of execution — represent the core of serverless computing. And from the view of developers, serverless computing can be deemed as *Function as a Service* (FaaS), which allows developers to develop and run their applications (or functions) without incurring the complexity of building and managing the underlying infrastructure. On the other hand, serverless providers always deploy application-dependent services, also known as *Backend as a Service* (BaaS), for their customers, such as Database and Object Storage Service (OOS). Thus, from a comprehensive point of view, serverless computing is an integration of both BaaS and FaaS [65] as a unified service form to its customers.

Consequently, with the automatic management and lightweight features, serverless computing could not only help developers focus on the core logic of applications but also benefit them with a more refined billing policy that can be calculated at run time in units of milliseconds [147], which could dramatically reduce the cost of developers. Moreover, it also facilitates serverless providers to reduce the amounts of idle resources to nearly zero for improved resource utilization, which allows more customers to be served with a fixed amount of resources.

Given the bright prospect of serverless computing, many cloud providers have proposed their respective framework such as Amazon Lambda [96], IBM Cloud Function Openwhisk [104], and Microsoft Azure Function [70] as a versatile platform deployed in commercial clouds to support diverse public services, say, video processing [13], [52], machine learning [35], linear algebra [118], [140], data analytics [55], [74], and distributed computing [36], [51], [64], [112]. The appealing features and benefits of serverless computing have also promoted plenty of studies in recent years with focus on the optimization of its efficiency in various aspects, say, its start latency, which is one of the major challenges in the fledgling period of serverless computing.

However, as one of the novel computing paradigms, serverless computing differs significantly from serverful computing in terms of application characteristics and runtime environment. Therefore, existing technologies would face great challenges and need adaptations to the accommodation of serverless requirements. For example, serverless is

- *Yongkang Li, Yanying Lin, Yang Wang, and Kejiang Ye are with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, Guangdong 518055, China, and also with University of Chinese Academy of Sciences, Beijing 100049, China. E-mail: {yk.li1, yy.lin1, yang.wang1, kj.ye}@siat.ac.cn.*
- *Chengzhong Xu is with the Department of Computer and Information Science, University of Macau, Macau, China. E-mail: czxu@um.edu.mo.*

Manuscript received 12 July 2021; revised 7 March 2022; accepted 6 April 2022. Date of publication 12 April 2022; date of current version 10 April 2023. This work was supported in part by Science and Technology Planning Project of Guangdong Province under Grant 2020B010164002, in part by National Natural Science Foundation of China under Grant 62072451, in part by the Shenzhen Basic Research Program under Grant JCYJ20200109115418592, and in part by the Youth Innovation Promotion Association CAS under Grant 2019349. (Corresponding authors: Yang Wang and Kejiang Ye.)
Digital Object Identifier no. 10.1109/TSC.2022.3166553

more sensitive to network latency and requires more fine-grained scheduling policies, however, network communication in serverless is fairly complicated and the granularity of scheduling in traditional architecture is also relatively large. Consequently, existing approaches could not adequately satisfy these requirements, and the most urgent task is to determine what inherent problems we are facing up in the current serverless research and what potential methods might be effective to resolve these problems.

As far as we know, some challenges in serverless computing have been adequately studied while others largely remain open. Hence, we need to distinguish the emphases on different problems in serverless computing and pay more attention to those not well solved. Moreover, there also exist certain fields, such as the so-called *serverless edge computing* [18], which, though promising, have not been well stepped into as edge computing usually has higher requirements for low communication latency, and fault tolerance supported by the serverless is often not sufficient. They are thus worth further exploration and research.

Motivated by those foregoing problems, this paper focuses on the underlying infrastructure of serverless computing to figure out the coexistence of challenges and opportunities. To this end, we analyze the existing challenges and some practical approaches that could be applied in further research, and give our opinions on whether or not these problems are worthy of study. Then, we conclude some directions that could be promising in future research but have not been noticed widely yet.

In literature, some surveys on serverless computing have been conducted, each with different focuses and purposes [1], [59], [65], [72]. For example, UC Berkeley [65] once made a view for serverless computing, comparing it with cloud computing in more detail, revealing some basic characteristics and existing challenges in certain use cases. Others mainly focused on the shortcoming of serverless, e.g., Hellerstein *et al.* [59] analyzed the deficiencies of current serverless products, which are manifested in programming and hardware supports. Although these surveys are valuable to gain insight into serverless computing, they are far from reflecting the state-of-the-art of this technology. Given the merits of serverless, numerous pieces of research in this area were developed rapidly in the past three years. Thus, it is highly desired to survey the most recent achievements in this regard, which is the theme of this paper.

We first retrieved papers for the study using "serverless", "function as a service", "function-as-a-service", and "FaaS" as keywords to search from some well known academic databases (e.g., ACM Digital Library, IEEE Xplore, DBLP, and Google Scholar) starting from January 2017 onwards, and then selectively picked up those from top conferences and journals, which focus on the theme of this survey, as our starting point. Finally, we adopted the often-used *snowballing* method [142] in software engineering to facilitate the searching process. In particular, we systematically analyzed all the references of the selected papers by first browsing their abstract, introduction, and conclusion, and then scanned their remaining part to quickly filter out those not highly related to our study. We then analyzed the reference papers' references again by following the same procedure repeatedly with cross-checking among authors. Finally, 150 papers are intentionally selected, which are reasonably enough to reflect the state of the art of serverless computing.

The remainder of this paper is organized as follows: in Section 2 we overview the background of serverless computing, including its evolution history and some related concepts. In Section 3, by following a general process of serverless computing, we summarize some of its main characteristics whereby the existing challenges are identified and their state-of-the-art studies are overviewed. After that, we investigate some typical open-source frameworks in Section 4 and study how they address the identified challenges. We describe the prospects of serverless computing from our point of view and come up with some research opportunities in the future in Section 5 and conclude the paper in Section 6.

2 Background

In this section, we first introduce some background knowledge on the evolution of virtualization technology upon which serverless relies, and then describe some important related concepts, which are easily confused with the serverless definition.

2.1 Evolution

In general, serverless computing relies on virtualization technology to realize isolation and resource management. The evolution of computation virtualization, by and large, has gone through three stages: *Virtual Machine* (VM), *Container* and *Serverless*. Virtual machines, such as Xen [30] and Kernel-based Virtual Machine (KVM) [39], are characterized by multiple operating systems spinning up on one host machine with the aid of emulation or hypervisor to support higher isolation and better hardware resource utilization. However, this technology incurs high memory overhead, which in turn compromises the overall system performance. To address this issue, Containers have resurged and spread with the release of Docker technology [129], which leverages sharing kernel to increase the deployment speed and portability while lowering the costs. The container has drawn much attention as it decreases the size of the virtualization unit and resource occupancy.

The emergence of VMs and containers promotes the birth of cloud computing, which is the on-demand platform to provision the compute infrastructure without direct management from developers [15]. Generally, cloud computing consists of two parts: applications delivered over the Internet and the hardware resources and system environments in data centers [16]. Cloud computing reduces the cost to most normal users and improves the resource utilization that is often low in the past. However, the cloud users still bear a heavy burden to interact with the cloud, and thus, are not entitled to the full benefits offered by the cloud.

The desire to overcome this issue motivates the appearance of serverless computing, invented firstly by Amazon in 2015, named *AWS Lambda* [96] service. Compared to VM and container, the serverless technique is more lightweight and flexible. As illustrated in Fig. 1 where serverless, Virtual Machine and container are compared, the orange dotted frame shows where users would be involved when using those technologies. As we can see, the serverless platform

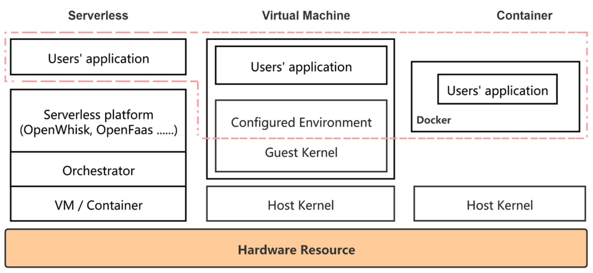


Fig. 1. Comparison between serverless, Virtual Machine and container. The orange dotted frame shows the layer that users could aware and need to involve.

serves as the interaction media between the application and the underlying infrastructure, simplifying the development of applications by sheltering resource management for developers. Consequently, no server could be observed from the user perspective, even though the serverless platforms still rely on servers in actuality to set up the basic runtime environment. In contrast, users adopting VM or container as the platform need to apply such resources and deploy the needed software environment before invoking their applications.

As we can see through the history, the granularity of virtualization has been reduced gradually from machine to function with the evolution from VM-based platforms to the serverless platform, which frees developers from the burden of resource management and complex configurations. Moreover, smaller granularity often means a more accurate billing unit, which in turn results in more efficient resource utilization, a lighter programming model, and faster product iteration as well.

2.2 Related Concepts

Over the past few decades, the system architecture and application of serverful computing have been intensively explored, and also the backend architecture has gone through a phase of rapid iterations with many emerging concepts. Since the granularity of VM is so large that it occupies too many resources and incurs high overhead, some studies are engaged in designing lightweight VMs, such as *Lightweight VM* [2], [90] and *Unikernel* [87]. The Lightweight VM reduces the size of the hypervisor or Virtual Machine Monitor (VMM) for performance improvement, while the Unikernels are built by compiling high-level languages directly into specialized machine images that can run directly on the hypervisor.

With more and more applications being deployed in the cloud, traditional computation architecture cannot satisfy the requirements of low latency and cost. Thus, *Cloud Native* [77] is proposed as a new development method and technical architecture. It can build and execute scalable applications in a dynamic environment such as public, private, and hybrid clouds [38] with an assumption that the applications are originally designed for the cloud. Some technologies (e.g., microservices [54], server mesh [83], serverless, and container) could act as the functional elements in this architecture. Cloud-native applications are typically organized in a microservice-based framework, a variant of the service-oriented architecture, which is characterized by a single application being divided into a collection of loosely-coupled fine-grained services. Different services in microservices interact with each other via lightweight protocols — *Remote Procedure Calls* or *Service Mesh*, a dedicated infrastructure layer to handle service-to-service communication in an independent and flexible way. Since the microservice architecture inherits from cloud technology, it still requires users to pay for the capacity and resources occupied by their idle services. As a result, it is more suitable for those long-time running tasks, which are different from the target of serverless, which is being evolved as another form of Cloud Native.

3 Characteristics and Challenges

In this section, we firstly analyze the general process of serverless computing and then summarize some characteristics of it for further understanding. Finally, we figure out the existing challenges and some efficient methods and propose our viewpoints to which we should pay more attention.

3.1 Characteristics of Serverless

We are now summarizing the characteristics of serverless computing as shown in Fig. 2 where its whole workflow is depicted. The flow-process consists of two phases: *function programming* and *function serving*.

Function Programming. Programmers develop their functions by following the rules provided by platform providers in local environment. After that, they deploy the functions

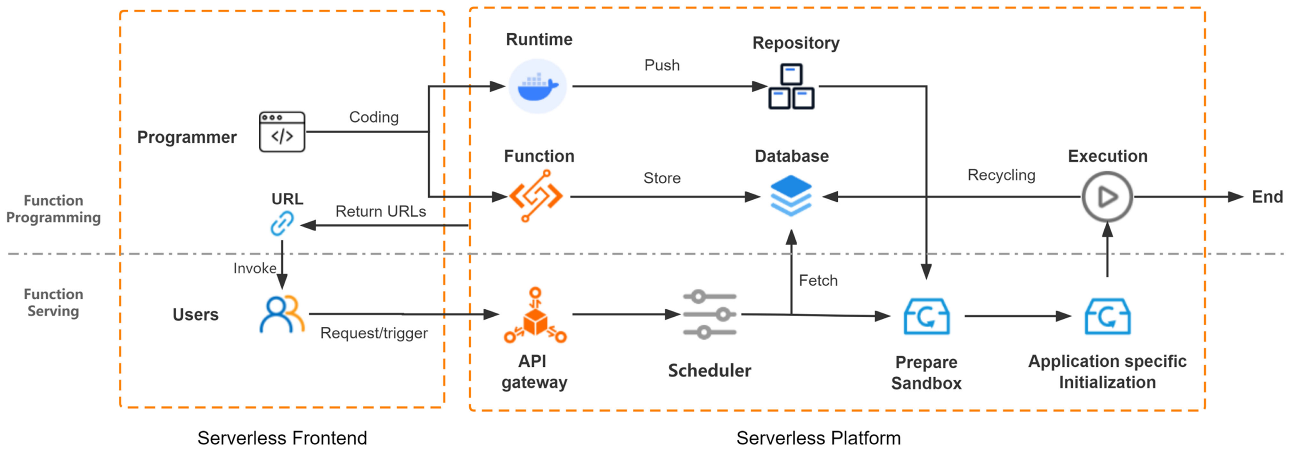


Fig. 2. The whole process of serverless computing.

to the platform using the command line where the platform first saves these functions in a database and pushes the runtimes of these functions into a repository, then it returns the URLs of these functions to the programmers.

Function Serving. Once a user wants to invoke a function, he/she could use its returned URL or a configured triggering event to invoke the function through an API gateway provided by the platform. The load balancer or the scheduler of the platform then fetches the function from a database and prepares a running environment, called *sandbox*, by obtaining the function's runtime from a remote repository. After these preparations, the platform can initialize the application-specific environment (e.g., loading class files), and then execute the function.

Again, from these descriptions, the serverless model is attractive in terms of reduced burdens in managing servers, automatic scaling, pay-for-use pricing, and event-driven streaming. Despite these benefits and the popularity of serverless systems, designing a serverless platform to fulfill the requirements of efficiency and scalability remains some big challenges. To gain a deep insight into serverless computing and identify those underlying challenges, we describe in this section a diverse set of characteristics of the serverless platform.

Hostless and elastic. Serverless is hostless and elastic, which implies that users do not need to work with the servers on their own. As such, the operational overhead could be less since users do not need to upgrade the servers and apply security patches. However, the payoff is a new challenge as it requires different kinds of metrics (e.g., resource occupation, execution time) to be monitored in an application. Given the hostless feature, the developers are not required to manage the resources by themselves, which allows the serverless billing to only count on the resources being used.

Lightweight. According to [46], serverless applications often employ a small number of serverless functions. Specifically, 82% of all use cases consist of five or fewer functions and 93% have less than ten functions. There are two potential reasons for these observations. First, developers could reduce the number of codes they need to write and concentrate on unique application logic, while all the other concerns are supported by the cloud providers. Second, it seems that the size of the serverless functions that developers often choose at present is rather large. Nevertheless, the programming complexity and schedule policy need to be taken into account when determining the optimal granularity for the serverless functions, which is still an open research challenge.

Statelessness. A stateful application could remember at least something about its state each time that it runs, so it requires persistent storage to store its states. However, serverless functions usually run in a stateless container hosted by the serverless platform, which only lives for a short period without saving ephemeral state information in its persistent storage. In this way, the scale-out of the platform is relatively easy as the users could merely spin up more instances. Due to this feature, the data locality in a single container is often not a factor to be considered. However, on the other hand, being stateless also implies that some techniques depending on transferring state information could not be used efficiently in serverless applications, e.g., iterative machine learning, graph algorithms, and interactive big data computation. This is a serious problem for some services, which have motivated some works to exploit stateful serverless computing [23], [120], [124].

Short But Variable Execution Times. Functions in serverless are typically alive in a range of a few hundreds of milliseconds to a few minutes with a billing unit at millisecond-level granularity [67]. Specifically, the execution times of serverless functions can last from $100ms$ to $15min$, which could be 4 orders of magnitude difference [116]. For this feature, AWS Lambda particularly limits its function executions to 15 minutes and bills users in a unit of $100ms$. However, such billing granularity is still large for most applications as the execution times of the majority of serverless functions are not that long [46]. Thus the non-execution overhead associated with running function might be relatively large, especially for those functions with short execution time.

Poor Intra-Function Parallelism. Most previous works often adopt a so-called *inter-function parallelization* pattern to parallelize applications across a set of separate function instances [22], [24], [118] and share data via persistent storage. However, this pattern could lead to significant communication and synchronization overhead among the concurrent function instances [22]. Rather, parallelizing the functions and executing them within a single function instance might be more suitable and economic, especially for compute-intensive workloads. Kiener *et al.* [73] carried out some experiments on such *intra-function parallelization* pattern, and the data shows that good parallelism within the functions can save some costs. For example, the cost can be saved up to 81% in Lambda and 49% in Google Cloud Functions. However, a single function is only limited to the execution on up to two CPU cores [67]. Thus, the support for parallel computation inside the function is still relatively poor.

Migration. In serverless platforms, the source code, binaries, and other dependencies need to be fetched remotely, and an isolated execution environment (e.g., docker) is also required to initialize for each function invocation. Thus, serverless functions tend to have a start-up cost, which makes the migration unappealing [139]. As a result, the system overhead and service latency could be increased during auto-scaling and load balancing of the serverless platform.

Separated Containers. Most current public serverless service providers, such as AWS, Google, and Microsoft Azure, isolate and execute each function in separated containers. Unlike traditional patterns that could share the same data if functions/tasks are in the same container/VM, serverless introduces more communication among separated containers/functions, thus there will incur higher latency and more network communication pressure.

Burstiness. Workloads usually fluctuate their degrees of parallelism by several orders of magnitude in a unit of seconds [51], [64]. A workload is bursty when it includes sudden and accidental intensity changes or load peaks. Eismann *et al.* [46] have tested a set of workloads on a serverless platform to identify whether or not functions maintain high volatility. They found that 81% of the tested workloads are bursty, implying that burstiness is a common and essential feature in serverless computing, which is different from cloud computing, where applications often run smoothly and live for a long time, consistent with its elastic property. Given this feature, some troubles are brought to task scheduling. We will discuss them in the following sections.

3.2 Existing Challenges

Despite those aforementioned features and benefits, there remain some challenges because of the differences between serverless and serverful computing. These challenges could make a great influence on system performance and security. Due to their importance, numerous pieces of research have been conducted and achieved some good results [4], [44], but there are still several noticeable problems left for further investigation. Next, we identify some existing challenges based on the serverless features, then overview the studies on them, and finally present our view on each challenge for subsequent research.

3.2.1 Startup Latency

Startup latency, including the latency of both *warm* start and *cold* start, typically refers to the period from a function invocation to its execution. In serverless computing, a developer usually sends a function to the serverless platform for execution where some initialization works have to be accomplished before the target function can execute. Once the developer sends a trigger event to the platform, the first step is to prepare a sandbox — a lightweight isolated execution environment — for the function. The sandbox initialization may start from either a shut-down or a running environment, which is the main difference between the cold start and the warm start. In general, the cost of the cold start is significantly higher than that of the warm start [115]. After the initialization, the sandbox runs a wrapped program that contains the target application specified in a configuration file and a wrapper responsible for the function invocation. This period, from the start of the wrapped program to the moment that the target function is ready to run, is called *application initialization*.

Startup latency is a critical issue in serverless computing. Since functions in serverless tend to have short execution time in general, long startup latency can lead to high relative costs. For example, the *Execution/Overall latency ratio* of all the functions in *gVisor* could achieve no more than 0.65, and 80% of functions cannot achieve the ratio no more than 0.3 [44]. This is not enough for the pay-as-you-go model and short execution time. Generally, there are three ways to optimize the startup latency: 1) cache-based method [57]; 2) optimization on sandbox initialization [98]; and 3) checkpoint/restore-based [45] optimization.

Cache-Based Optimizations. Many serverless platforms are designed to follow some caching policies in the course of function startup [4], [57], [70], [98]. *Zygote* [57] is a critical process in Android for executing Android applications. Before the execution, Zygote can reduce the initialization time by sharing the code and memory information between running VMs, loading the classes and libraries to be used into memory, and organizing their linking information. By integrating Zygote into a cache for Python interpreters, applications in SOCK [98] could be executed with pre-loaded libraries to minimize the startup latency. In contrast, SAND [4] introduces an application-level sandbox to weaken the isolation between functions from the same application by allowing those instances in the same sandbox to share function and library codes. As in-memory caching would take up valuable memory resources, *Azure Functions* [70] takes advantage of a shared file system to progressively load the libraries required by applications.

Unlike those efforts that mainly share libraries or packages, *SEUSS* [34] leverages *snapshot* to reduce the latency and support *burst resilience* more powerfully. Specifically, it deploys functions based on the snapshots of isolated unikernel that consists of function logic, language interpreter, and library Operating System (OS), bypassing the high overhead of initialization. To reduce the memory footprint of caches, SEUSS applies page-level sharing to copy the modified memory pages into a snapshot. Thus it could reduce the startup latency and increase the number of cached functions in the memory.

Although the caching technology is effective in many cases, it still experiences some performance troubles when coping with the cold start problem. On the one hand, the caching technique will consume scarce memory resources in large-scale clusters. On the other hand, some latency is much higher than the average in concurrent applications, which is called *tail latency*, caused by the function initialization, storage accesses, and the burst function invocation traffic [128]. However, most existing caching methods could not resolve this problem related to the environment initialization as they in general only save the cost of storage access.

Optimizations on Sandbox Initialization. Apart from the caching policy, most platforms optimize their sandbox initialization through customization. For example, Oakes [98] proposed a specific container system with simplified operations and an on-demand initialization process. With this design, a low-latency sandbox initialization could be achieved via some optimizations on storage, communication, and isolation.

In contrast, the VM-based sandbox could provide stronger isolation at the expense of higher overhead and latency. As a result, the research on lightweight VMs has drawn more attention to the resource utilization [2], [9], [75], [90]. Some methods, including Unikernels [87] for lower memory footprint, specific hypervisors [90] for lightweight and isolation or combination of the two, have been developed for serverless computing industry [2], [136]. Despite the devoted efforts, those methods that optimize the sandbox initialization suffer the same limitation in reducing the latency resulting from the application initialization, which may take up a large part of the startup time [44].

Checkpoint/Restore-Based Optimizations. Checkpoint/ Restore (C/R) originated from Berkeley Lab's project in 2002 [45] is a widely used technology for fault tolerance and load balancing problems. So far, it has been used in Userspace (CRIU [49]) to reduce the startup time in applications. C/R reduces the startup latency by enabling a snapshot of a running application instance and restarting it from the point at which the snapshot was taken while CRIU achieves such work in a fully-transparent way at the user level. Instead of modifying the application code, CRIU injects checkpoints into the code while the application is running. As the container is a Unix process, sandboxes implemented by the container could adopt C/R to optimize its startup time. Both states of the sandbox and application should be saved, and then can be restored from a checkpoint to execute later. Leveraging such an idea, Wang *et al.* [138] proposed *Replayable Execution* which extends C/R to restore memory pages and accelerates the cold start. Those saved states are permitted to share among the different containers, breaking through the limitations of the non-root container with the in-place restoration. Similarly, Silva *et al.* [121] also exploited the C/R method with CRIU to optimize the cold start, but they just came up with a prototype and didn't make further optimizations. However, VM is not suitable for this method due to the large granularity that would incur high overhead while restoring.

To overcome the weakness of the three methods abovementioned, Du *et al.* [44] implemented an init-less booting serverless sandbox system called *Catalyzer*, which combines the C/R-based optimizations and new OS primitives to minimize the service startup latency. In particular, Catalyzer classifies the boot procedure into three types: *cold boot*, *warm boot*, and *fork boot*. It uses different methods for each type of boot, says, adopting on-demand restoration for either the cold or the warm boot, and forking the sandbox template for the fork boot. Reusing the states of application and sandbox enables Catalyzer to reduce the startup latency. The results show that Catalyzer can reduce the startup latency by an order of magnitude, where the sub-millisecond startup latency can be achieved in certain cases.

Since the startup time could incur significant latency and overhead for the serverless platform, many efforts have been made on this issue. Meanwhile, based on our experience, the execution time of a simple function, say, *hello-world*, written in Python can cost as small as $6ms$. Thus, the sub-millisecond startup latency only occupies an extremely small portion of the whole process with respect to more complicated functions. Therefore, some existing optimizations on the startup latency could fulfill the requirements of initialization in the mass. However, other factors, such as the communication among different components and the application state synchronization, could also potentially incur higher overhead during function execution. As a result, we think future studies should not only concentrate on the optimization of the startup time but also consider other factors for the overall latency optimization.

3.2.2 Isolation

Isolation is a mechanism in the platform to separate different tenants from each other for security and operation concerns, which is often used to form the basis for multi-tenancy. Traditional cloud platform providers (say, AWS EC2) in general achieve it by using VMM-based virtualization [30], [39] or offering bare-metal instances [97]. Since more instances are allowed to execute on a serverless platform than on a traditional platform, the lightweight feature of serverless functions can expand the advantages in economics. Generally, there are three ways to isolate the workloads in Linux: 1) *Virtual Machines* (VMs) — virtualizing the whole machine and executing applications on separate VMs to enable stronger isolation with higher overhead; 2) *Containers* — combining with Linux kernel features to provide the isolation mechanisms and allowing all the workloads on the same physical machine to share one kernel; 3) *Language VM* — providing the isolation and execution environment for specific languages (e.g., Java). Fig. 3 compares the different security approaches among the three methods where the red symbol indicates the position of defending security. It illustrates that the container has weak isolation as it relies on the kernel security mechanism.

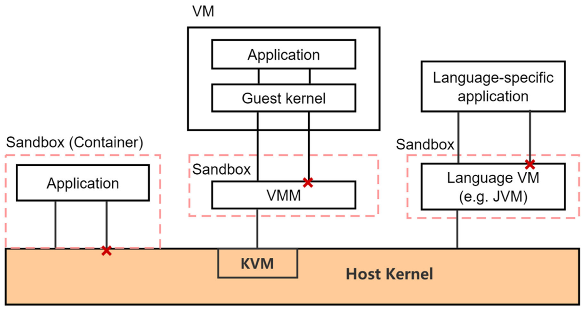


Fig. 3. Different security approaches among Container, VM, and Language VM. The red symbol indicates the position of defending security.

Virtual Machines (VMs). The VM, such as Xen [30] and KVM [39], is a widely used method to implement the isolation for different workloads. Generally, some hardware mechanisms (e.g., Intel VT-x) would be adopted to provide an isolated environment with its resource as a *sandbox*. Generally, many platforms use the combination of KVM and QEMU to provide a VM environment, in which KVM adopts Linux kernel and VT-x to virtualize CPU and memory, and QEMU emulates devices and I/O via invoking the interfaces of KVM.

However, some challenges limit the use of VMs. First, the granularity is as large as VM, which could reduce the scale of applications deployed on a single physical machine. Second, VMs need to emulate the whole execution environment while running application codes, which would incur high initialization latency. Some studies show that a typical VM's startup time is in the order of seconds, which is a significant overhead. Several solutions, such as customized hypervisor [2] and caching optimization [98], have been discussed in the previous section. Additionally, for secure computing, *Trusted Computing Base* (TCB) [144] should be configured to ensure the security of each VMs, which requires a set of security strategies and mechanisms implemented in hardware, software, and/or firmware components. But in the VM environment, TCB could be relatively complex because of the large size of the VMM or hypervisor. As we know, the larger the TCB is, the more attack interfaces would be exposed, which could weaken the security level of the serverless platform.

In spite of its challenges, the VM still provides many attractive benefits. First, VM leverages both software-assisted methods and hardware features to implement the virtualization, which shifts the security interface from the OS to more flexible software and specific hardware. So it could provide stronger isolation and security. Moreover, this benefit also liberates the kernel from the complicated security configuration. Hence, some work focus on reducing the size of the hypervisor and VMM. For example, AWS developed a project, named *Firecracker* [2], recently, which exploits the Linux Kernel's KVM virtualization infrastructure to provide minimal VMs. Firecracker still adopts KVM as the underlying virtualization tool but entirely replaces QEMU to build a new VMM with much fewer codes for the reduction of the surface area and the size of TCB. It seems that how to combine the benefits of VMs, containers, and serverless techniques to reconstruct and optimize either VMM or hypervisor is still a worthwhile research topic.

Containers. Containers have been deployed widely in cloud computing due to their lightweight and simple architecture. Most serverless platforms adopt containers as their sandboxes. Traditional containers, such as Docker and Linux Containers (LXC), share the same host OS kernel, thereby reducing the granularity of virtualization to realize lightweight and resource-efficient computation at the cost of weakening the isolation. In general, containers combine with several kernel features to provide the isolation, which includes *namespaces*, *cgroups*, *chroot*, and *seccomp-bpf*. These techniques regard the container as a process to provide isolation, in which cgroups guarantee the resource limits for diverse process groups; namespaces implements the resource isolation; chroot isolates the filesystem; and seccomp-bpf uses a configurable policy to filter out the system calls that process permits.

However, these features of container imply that it must rely on the system calls of the host machine, suffering weak isolation. There are a lot of research efforts in both academia and industry trying to design a trusted sandbox on the containers, and most of them focus on re-architecting the boundary between the container and host machine. For example, Google gVisor [136] designed a user-space kernel to sandbox the applications, which can intercept all the system calls from functions and handle them in userspace. Thus, it creates strong isolation between function and host machine without relying on virtual hardware. Therefore, gVisor provides some functionality of the operating system to userspace. Likewise, Graphene [126] and Bascule [28] also adopted the similar method to provide containers with stronger isolation. However, this approach still has some inherent limitations. For example, much overhead would be incurred while intercepting and handling syscalls in userspace.

Language VM. The last method of isolation is leveraging the characteristics of language VM, such as Java VM. There are two typical kinds of research that have been conducted to strengthen the isolation in Language VMs. The first is running multiple applications in a single process, defending side-channel attacks, and providing a sufficient performance guarantee. Others, such as SAND [4] and Alto [79], allocate a process for each trust domain to ensure stronger isolation and security. Nevertheless, Language VM is usually specific to one or several languages, while a universal platform should support arbitrary languages. As such, this method is not adopted widely.

Besides, there are also some efforts to study the optimization of a single language to improve both efficiency and security. For example, Herbert and Guha [60] presented a language-based sandbox for JavaScript to minimize the size of the TCB. It could safely run functions in an OS-based sandbox-like container with the seamless transition once the operations are not supported in the language-based sandbox. Similarly, Boucher *et al.* [32] also proposed a serverless platform that limits the programming language to Rust [63].

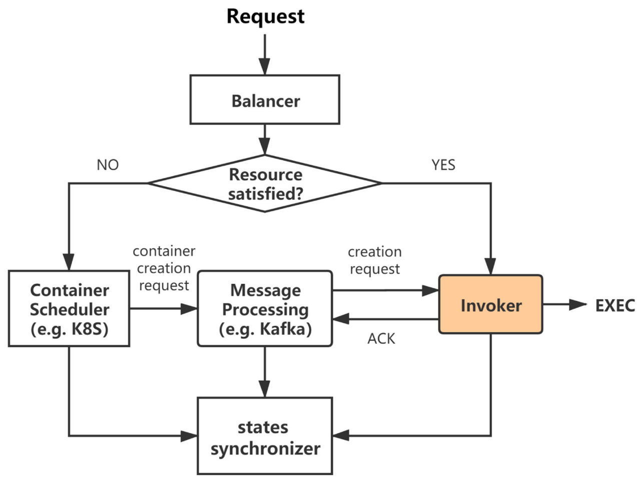


Fig. 4. The general process of serverless scheduler. The Invoker is generally either a physical machine or a VM, which provides runtime.

As the computing unit becomes lighter and lighter, the startup latency for public cloud computing would also get smaller and smaller. Serverless takes the function as a basic compute unit and generally relies on the container as the underlying runtime, which could also incur relatively high overhead and poor isolation. This may lead to security problems, say one workload infers the data belonging to another workload, or performance is influenced by other workloads on the same host. Fundamentally, the containers and VMs were born in the age of cloud computing that may not be well suitable for the serverless. Given this consideration, Gadepalli *et al.* [53] implemented a lightweight isolation model based on WebAssembly [58], a portable low-level bytecode sandbox with software fault isolation and memory-light sandbox. As such, designing a specialized runtime for serverless computing, such as a new virtualization technology that supports a smaller computing unit, maybe also a good method, apart from making optimization on the containers or VMs.

3.2.3 Scheduling Policy

As shown in Fig. 4, the scheduling process of serverless functions can be generally split into two phases: *runtime configuration* and *function scheduling*, which usually arise in two different platforms. The runtime configuration often originates on serverless platforms, say, OpenWhisk and OpenFaas, where the source codes are bound to their running instance. Whenever receiving a function call, the serverless platform first retrieves the function's source codes as well as the required resource configuration (CPU, Memory, etc.) from a database. Then, it enables a balancer to evaluate the current running state (say, the resource utilization of living instance, the concurrency of running instances, etc.) of the platform, and finally initiate an invocation to create a container or send a call directly to a pre-warming runtime instance, according to the Quality of Service (QoS) requirements.

Once the current running environment could not satisfy user demands, the life cycle of the function would proceed to the phase concentrating on the system resource allocation pipeline, such as container creation, container orchestration, container placement, etc. This phase usually takes effect in the resource management system, such as Kubernetes [33] and Mesos [61]. And the requests for container creation are normally queued in messaging middleware (e.g., Kafka [66]) to wait for an invoker, who is either the host machine or the VM that accomplishes the request and returns acknowledgment messages (ACK).

The second phase of serverless scheduling targets the scheduling of configured serverless functions. It mainly relies on general-purpose task scheduling policies, which are roughly subdivided into two types: *centralized scheduling* [131], [132] and *distributed scheduling* [92]. The centralized scheduler is also referred to as *monolithic scheduler*, which adopts a centralized approach to managing resources and scheduling tasks with a global scheduler. In contrast, the distributed scheduling exploits a per-machine scheduler to support large-scale clusters. It generally shares a common architecture: the submitted tasks are typically assigned to compute servers via a simple *load balancer*, and the load imbalances are detected and handled by a per-machine scheduling agent, who can migrate the selected tasks away from busy servers to light ones.

However, existing scheduling mechanisms designed for cloud computing are not amenable to serverless functions. First, the cloud scheduling is typically centralized, which might not be suitable for the serverless platform when large-scale bursty scheduling requests are made even though burstiness is the basic feature of serverless as in this case much higher latency or even failure could be incurred if the platform is not elastic promptly. Second, general task scheduling frameworks often assume that the execution environment is already set up across all the servers in a cluster. However, in serverless platforms, the source code, binaries, and other dependencies need to be fetched remotely on-demand, which may result in start-up cost [139], making the migration unappealing. Furthermore, the execution time of the serverless function can span a wide range of four orders of magnitude. Such a large variability would result in imbalances of the task queue length, which are hard to detect and may lead to spurious task migrations. For example, even the lower variability associated with the exponential execution times has been shown to result in migration rates as high as 50% [101].

Motivated by the requirements of the urgent demands on a suitable serverless scheduler, Kaffes *et al.* [67] proposed a centralized scheduler, which could assign functions to processor cores rather than servers. Unlike traditional server-level scheduling, it could predict the performance given the better management of the states of cores while downgrading the scheduling granularity to the core. Moreover, the centralized core-granular scheduling runs at a cluster level, maintaining a global view among the cluster resources which could schedule the workloads to any core in the cluster. Wang *et al.* [137] optimized scheduling from another aspect. They adopted a scheduler with deep reinforcement learning in *SIREN* to dynamically adjust the number of functions and their resources, making a trade-off between cost and performance.

As users delegate the function scheduling to a serverless platform, the platform thus needs to have a task scheduler with high performance and good scalability. However, many research efforts on serverless scheduling mainly concentrate on the startup latency, rather than the reconstruction of the scheduler, which is a more desirable work to be accomplished. Additionally, because of the variable time and burst feature of the serverless function, the prediction of its execution time during the whole life of the workload is still a troublesome problem. The scheduler should adopt a global view, which is similar to the case in a traditional distributed scheduler. Considering the characteristics of serverless, the scheduling mechanism should fulfill these requirements: (1) withstanding bursty scheduling requests within a short period; (2) fine-grained scheduling in the unit of function; (3) optimizing the system performance by balancing the cost and user requirements; (4) taking into account the function communication and parallelism.

3.2.4 Facility

A serverless platform could be simply viewed as a combination of FaaS and BaaS as mentioned above. In practice, the FaaS is usually supported by the BaaS, which provisions the basic infrastructure and auto-scaling policy. For example, the BaaS in AWS includes a datastore (e.g., Amazon S3, DynamoBD [7]), an application integration component (e.g., Amazon Simple Notification Service, EventBridge [29]), and so on. Among these basic constituents, the storage and network are the most important ones as they could substantially influence the system performance due to their data transmission speeds.

Storage. Data processing frameworks such as those in big data analytics (say, MapReduce [41] and Spark [14]) and iterative machine learning (e.g., K-means clustering [69]) need to share states among multiple tasks. Traditional frameworks generally implement a long-running storage service on each node for data sharing and cache the ephemeral data in local memory. Tasks in different execution stages can directly exchange the intermediate state information through point-to-point communications. However, these methods could not be effectively used in the serverless cases since (1) long-running storage services are not always available in serverless computing to manage the local storage; (2) most existing serverless platforms lack sufficient support for point-to-point communications between serverless functions, rather they rely on remote storage to this end.

Serverless frameworks typically use object storage (e.g., Amazon S3 [8]), database (e.g., CouchDB [40]), or distributed cache (e.g., Redis [113]) to share the state information between functions. However, this kind of storage is not perfectly suitable for short and sudden ephemeral data in serverless computing since it is not well designed for such data, suffering from bad performance, say, fetching data from remote storage could take up to 70% of the execution time [59]. Thus, to fulfill the requirements of data access, the ephemeral storage system for the serverless platform is suggested to meet the following requirements:

1) *Low latency and high performance:* Applications in serverless usually have short but variable execution times, which call for the storage to achieve low access latency, especially when a large number of workloads arrive, the ephemeral storage system needs to ensure a timely response.

2) *Fine-grain auto-scaling:* As serverless supports elastic and lightweight computing, the platform must be able to monitor the I/O requests at runtime and in the meantime satisfy the resource requirements of bursty workloads. Also, once the workloads finish, the storage resources should be released immediately to guarantee low cost. Thus, the storage should be able to scale up and down in a time unit of the order of seconds, or even sub-seconds.
3) *Transparent configuration:* Developers on the serverless platform only need to focus on their applications while leaving the management tasks to the service provider, who needs to provision the storage and other resources to the users, according to their needs and expenses.

One of the approaches is to build *distributed memory* by optimizing the network stack, reducing the latency when transmitting data. Both *RAMCloud* [99] and *FaSST* [68] leverage *Remote Direct Memory Access* (RDMA) to build in-memory systems, which can offload the package processing and transferring tasks to the Network Interface Card (NIC) for high performance and low latency at sub-seconds level. Nevertheless, they require the developers to specify the demands for storage manually, which violates the basic principle of serverless computing. Others [76], [103] try to build fast storage to provision resources and adjust them during the running process to realize the auto-scaling and transparent configuration. When considering the obstacle in the storage, *Shredder* [149] executes a small unit of computation in the storage, which could interact with the required data directly. Shredder allows users to transmit some functions and execute those functions locally, rather than moving data across the network. In contrast, *HydroCache* [143] adopts caching to optimize network traffics. It exploits a distributed caching layer to achieve low latency for I/O operations, thereby guaranteeing the *transactional causal consistency* [93].

Network. In serverless computing, functions are typically stored in remote storage (e.g., databases) while runtimes (e.g., dockers) are generally maintained in repositories and fetched as remote images. Often, serverless is gradually deployed for burst and parallel functions. However, it is difficult for the functions to communicate with each other via the network directly, but mainly through shared storage. It implies that the communications among serverless functions will send more packets than in traditional frameworks, which are typically built on top of point-to-point communication. Fig. 5 shows the structure of storage and network communication in serverless, where we can see that most data transmissions rely on remote networks, and the inter-function communications resort to third-party remote storage. Hence, all of the processes during the running of the application require a good network condition to guarantee low latency and high performance. However, the network status could be complex and unpredictable due to the uncertainties in a large number of concurrent workloads run in the execution environment.

In practice, there are several ways to address these problems:

Use High-Speed Network Technologies to Achieve Low Latency and Better Performance. For example, both the *Data Plane Development Kit* (DPDK) [109] and RDMA could reduce the network delay and achieve high speed by *zero-copy* to accelerate the data processing. By leveraging *Userspace I/O*, DPDK puts most of the tasks on packet processing in user mode, while RDMA offloads such tasks to NIC device and bypasses CPU during the data transmission. These technologies have been deployed in traditional distributed systems and achieved relatively good results, but they require further research on the effective integration with serverless techniques. For example, *Particle* [125] redesigns the network stack to make it amenable to the multi-node serverless system. It optimizes the network startup by dynamically providing ephemeral IPs pool and avoiding any side effects on its applications.

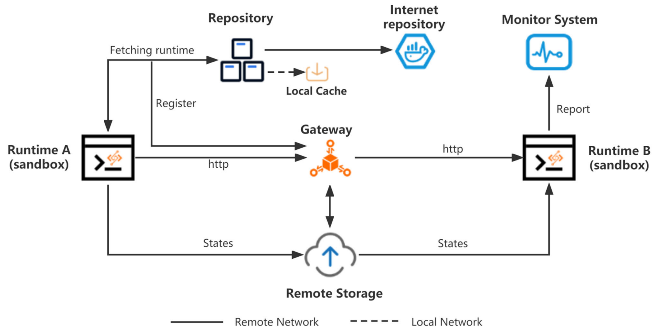


Fig. 5. Architecture of network communication and storage in serverless. Different runtimes could exchange small data or requests through gateway, or transmit much data via remote storage.

Adjust the Function Placement Policy to Ease the Communication Pressure. A recent study [139] shows that the function placement and other underlying factors could make a great influence on the system performance. As such, the platforms allow users to place their functions on the same VM or container instance to reduce the overhead. And the providers also may adopt machine learning algorithms to predict the quality of the function placements based on the historical results [88] or fusing multiple functions to form a single function appropriately [47] for cost reduction.

Appropriate Intervention of Developers. Furthermore, platforms could allow developers to provide logical relationships between the functions (e.g., some functions may need to communicate with each other frequently), enabling the platform to adjust the communication policies and the function placements to minimize the network overhead. The infrastructure underneath is still transparent to users when adopting such a method, but the platforms could get more experiences and take more suitable strategies.

Since storage and network resources are generally invisible to users, serverless providers have to propose methods on behalf of users to make the best decision on resource provisioning and management. As discussed above, the ephemeral storage and network are always employed together and they influence each other. Specifically, the storage relies on the network to achieve low transferring latency while the network needs to combine with the function placements in storage to reduce its invocation frequency. Thus, the storage and network should be optimized together when designing a serverless platform. For example, *Zion* [111] is a data-driven middleware for object storage with an attempt to move computation close to its required data, which could not only reduce the data transmission traffic but also lower the access latency.

3.2.5 Fault Tolerance

Currently, some serverless frameworks (e.g., *ExCamera* [52] and *PyWren* [64]) are not developed inherent strategies for fault tolerance, while others like *GG* [51] may simply ask user to retry the execution of function whenever it fails. Given this strategy, the function execution is required to be definite with a large number of historical data. However, one cannot guarantee that the remote storage storing the function has been cleaned up. As a result, applications running in parallel could access incorrectly updated data, leading to wrong results or status. This problem is especially serious for those stateful serverless functions and the Internet of Vehicles that handles emergencies. Thus, the fault tolerance mechanism of the serverless platform should avoid the restart process whenever the function execution or coordinator are failed to complete [150].

One simple approach is to leverage the concept of serializable transactions. It could guarantee the *sequential execution* of concurrent functions, which means that the updates from a single request should be visible to either all the other requests or none of them. However, this strong consistent model introduces high overhead, and thus, it is not feasible for the scalability of the serverless platform. Another method is *atomic read* [21], [94], which realizes the *access atomicity* at a fixed storage layer. This method is designed against the requirements of auto-scaling and short executing time in the serverless.

The traditional methods are not amenable to the serverless functions with required states. As such, most present providers encourage users to develop and program *idempotent* functions [11], which could ensure the impact of any number of executions is identical to that of one execution. However, this approach imposes some requirements on users, which compromises the principle of easy-to-use, and forces the providers to redesign the error processing mechanism in the serverless platform.

To provide stronger fault tolerance for serverless, AFT [123] inserts a component between the serverless platform and the cloud storage to address the fault tolerance for functions. With this design, AFT can transparently enforce the read atomicity and ensure consistent transactions without depending on pre-declared access sets, and in the meanwhile, achieve high throughput and low overhead at the cost of weak isolation. However, AFT needs intervention with servers, which may limit the platforms that could deploy it. In order to support any existing serverless platforms, *Beldi* [148] provides a library and new data structure for unified storage, which could record both application states and runtime logs for the optimization of the existing protocols for concurrency and distributed commits. And it could periodically re-execute unfinished serverful functions with no-repeat execution through logging.

Currently, many distributed frameworks have been deployed broadly in the commercial area, and it accelerates the development of cloud computing. In return, cloud computing also promotes the widespread use of distributed frameworks. As an emerging computation paradigm, nowadays, serverless was mainly deployed to serve distributed frameworks. Therefore, as significant factors for distributed computing, both concurrency control, and fault tolerance are still important in serverless platforms, which are worth further investigation.

4 Frameworks in Practice

Given the understanding of the identified challenges as well as their related studies, in this section, we showcase some typical open-source serverless computing platforms and analyze how they address the challenges, and discuss what should be concerned about when implementing serverless platforms in the future.

4.1 Typical Open-Source Frameworks

Currently, there are many open-source serverless frameworks (Apache OpenWhisk [104], IronFunctions [108], Oracle's Fn [107], OpenFaaS [48], Kubeless [127], Knative [12], Project Riff [110], etc.) that have been proposed to meet the increasing demands of serverless computing. Among these frameworks, some are built on top of existing cloud service infrastructures (e.g., Kubernetes) to serve applications, while others are implemented from scratch by deploying the functions directly on servers. The implementation methods will affect the performance of the scheduling in those frameworks.

According to the degree of dependency on the existing cloud service infrastructure, the open-source platforms could be classified into three categories: *weak dependency*, *semi-dependency*, and *strong dependency*. The weak dependency means there is almost no dependence on the cloud infrastructure when implementing the platform, the semi-dependency instead refers to the case that parts of components of the cloud infrastructure are exploited to implement the platform, and finally the strong dependency implies that the platform almost totally relies on the cloud service infrastructure. We select *OpenWhisk*, *OpenFaaS*, and *Kubeless* as the representative frameworks of these three types, respectively, based on their market occupancy and popularity on Github. In this section, we discuss their performance with respect to the identified challenges. In addition, we summarized the architecture of those platforms in Fig. 6, in which the blue frame represents some components using Kubernetes while the orange frame means that this part is optional and users could choose the suitable components.

OpenWhisk. *Openwhisk* is a mature serverless framework supported by the Apache Foundation, which supports many deployment options. It consists of many basic components to provide sufficient quality and scalability. Compared to other open-source projects (e.g., Fission [130], Kubeless [127], IronFunctions [108]), Apache OpenWhisk has a larger codebase, high-quality features, and a large number of participants. However, its mega tools (e.g., CouchDB [40], Kafka [66], Nginx [43], Redis [113], and Zookeeper [62]) pose a challenge to both developers and users for learning these tools. The complexity of the components also brings additional performance overhead, which challenges the system performance, especially in high concurrency scenarios.

OpenFaaS. Compared to OpenWhisk, *OpenFaaS* is a cloud-free, native server framework with a much simpler architecture and a collection of components, which include

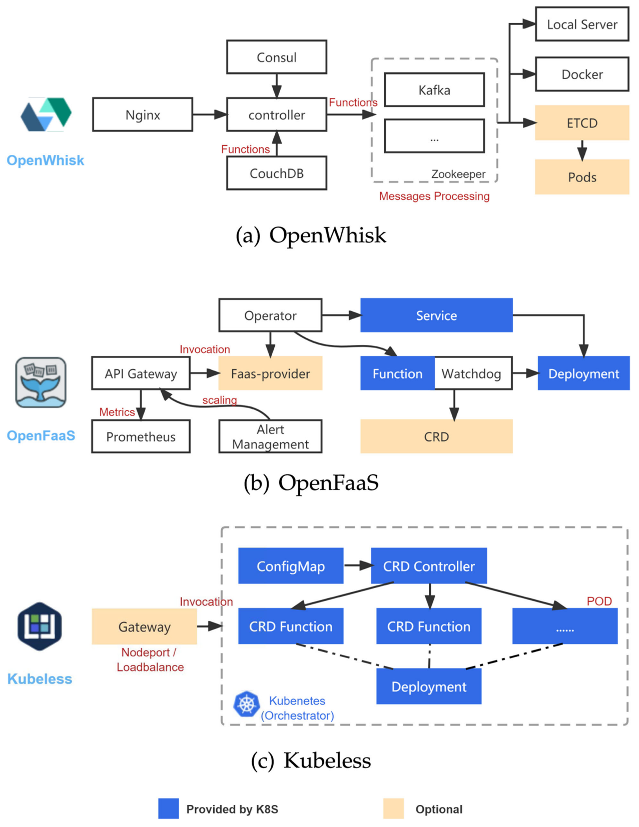


Fig. 6. Architecture of three platforms. The blue frame represents some components using Kubernetes, while the orange frame means that this part is optional and users could choose the suitable components.

API gateways for asynchronous calls, invocation tools, and auto-scaling components. OpenFaas could be deployed in the private or public cloud, and even in edge devices with constrained resources because of its lightweight feature. Users can deploy it on many different cloud management platforms (e.g., Docker Swarm [81], Kubernetes [33], and Mesos [61]) as well as on bare-metal servers.

Kubeless. A distinctive product of the open-source serverless platform is *Kubeless*, a Kubernetes-native platform built on top of core Kubernetes features like deployment, service, ConfigMap, and so on. Kubeless leverages a Custom Resource Definition (CRD) to create functions as a custom Kubernetes resource, and uses a controller to monitor those resources and execute the functions on-demand. Kubeless simplifies the elementary code so developers do not have to replay most of the scheduling logic codes that already exist in the Kubernetes kernel itself. This architectural principle achieves a trade-off between the platform performance and the convenience of architecting a serverless platform.

4.2 Performance on Challenges

The serverless framework is lightweight with a design focus on those compute tasks that have a short execution time. However, the implementation of serverless frameworks in practice is usually complicated with many involved components, which harms its performance with respect to the identified challenges. To fully understand these issues, we summarize some quantitative results in the literature to compare the three selected open-source frameworks. and also show how their complexities influence practical performance.

Startup Latency. Based on the proposed serverless benchmark, Yu *et al.* [147] and Li *et al.* [82] found out the differences in latency between the selected frameworks. The results show that the median latency of Kubeless is the lowest, as it forks the function for every request. In contrast, the image pulling operations in OpenWhisk would take more than 80% of the whole execution time in the cold start, which incurs a high overhead. Even though Openwhisk provides a warm start mechanism to reuse the container, it would pause the container after $50ms$ since it has finished. Additionally, OpenFaaS also has lower latency and short response time compared to OpenWhisk [100], and it performs better than Kubeless for large payloads [82].

Isolation. All of these three frameworks adopt the dockers as their runtimes. Thus they all should have the same performance under isolation. However, more efforts need to make for stronger isolation as discussed in the previous section.

Scheduling. In general, both open-source and commercial serverless frameworks are built on top of some existing technologies (e.g., Kubernetes) to serve as an orchestrator for application services, which are composed of multiple containers. Therefore, they only need to focus on the underlying scheduling issues as both OpenFaas and Kubeless do. In contrast, OpenWhisk implements the controller by itself, so it can support any container-based orchestrator to enhance the scalability of its function executions. OpenWhisk thus performs better than the other two frameworks in this aspect.

Facility. These three frameworks also adopt some existing remote databases, such as *Redis* [113] and *CouchDB* [40], to store data and states of functions [78] over the network. Thus, accessing the data and state would incur high network traffic. However, all of these frameworks leverage traditional network technologies to achieve communications between different components, lacking the optimization of the collaboration between storage and network. Consequently, it could incur high overhead during the framework executions.

Fault Tolerance. Currently, fault tolerance is not well supported in the selected frameworks. They all only use a basic retry mechanism once the execution failure is detected. Notably, compared to the other two frameworks, the success rate of OpenWhisk is relatively stable when running a single service instance. However, for multiple running instances, its success rate would drop dramatically due to the bottleneck of some components, such as the *Nginx* component, which needs to handle too many HTTP requests. As such, these components compromise the scalability of OpenWhisk for fault tolerance [100].

5 Research Opportunities

Despite the challenges we discussed above, there are still some other aspects that, we think, have hardly ever been touched, but might be promising in the future. For example, the exploitation of machine learning for the design of serverless infrastructure, the combination of the serverless with edge computing, etc. In this section, we focus squarely on these aspects, and from our points of view envision some research trends and opportunities in serverless computing. We hope these analyses could inspire some relevant studies.

5.1 Machine Learning for Resource Management

Applications in cloud computing usually rely on basic infrastructure resources, which are organized as their runtimes, to fulfill the requirements. Consequently, a set of resource configuration parameters should be prepared before the running of applications. However, different workloads might have different behaviors, which in turn have resource requirements. Therefore, the adopted resource policy should be varied from case to case when finding a suitable resource configuration.

Traditional commercial cloud service providers, such as Google and Amazon, have developed their own platforms, which could offer configurations that can well manage the resources in manual or automatic ways for specific workloads. For example, Google Cloud Compute Engine [10] designed a recommendation approach to the VM types for particular applications while AWS proposes a service with automatic scaling policy [20] for their VM instances. In comparison, for others, say *Brog* [135] and *Quasar* [42], they all could calculate the best parameters based on the requirements of users (e.g., running time) to manage the resources for the scheduled VM instances. Users who intend to apply these technologies to the serverless platform need to be familiar with its basic infrastructure since such detailed information regarding the resource configuration is invisible to them.

As machine learning has been gaining much popularity in recent years, some researchers attempt to combine resource management with machine learning algorithms to optimize resource configurations. In order to figure out the configuration space for different VM instance types and cluster sizes, *CherryPick* [5] leverages *Bayesian Optimization* [117] to adaptively choose the best configurations with low cost for big data analysis. Similarly, *Jockey* [50] and *ARIA* [134] resort to the analysis technologies on the history traces to predict subsequent function workloads.

Unlike the above studies, some researchers advocate that resource configurations are determined based on only a subset of workloads. For example, Venkataraman *et al.* [133] specified that decision making on resource configuration based on workload execution history is not always feasible since the generated datasets might be different even though a job is repeated periodically. To address this issue, they proposed a performance prediction system, called *Ernest*, which executes a set of instances of the job on the samples of input data, and thus the generated data is used as training data to build a performance model, which is in turn employed to predict the performance on the large input dataset.

Unfortunately, the cloud providers currently have not yet found fine-grained resource configuration policies, which are highly efficient for serverless computing [3]. They only use default parameters (e.g., 256MB memory in Openwhisk [104]) or likewise require developers to provide a small set of parameters for the configurations of memory and CPU. These configurations are often very simple, lacking the notion of other factors, such as function placement and network condition, which could also influence the platform performance.

To make full use of resources, the resource management for efficient function execution can be divided into three phases: (1) constructing a model for resource demand prediction; (2) allocating functions to suitable containers and configuring the resources with QoS guarantees; (3) flexible resource scaling to improve its utilization.

In order to predict application demands to effectively configure the resources, Akhtar *et al.* [3] presented *COSE* to predict the needed resources at runtime and then find the optimal configuration for these resources. To this end, COSE is designed to have two critical components: *Performance Modeler* and *Config Finder*. The former could learn a performance model for the application by leveraging *Bayesian Optimization* while the latter is designed to find the best configuration for those parameters that could satisfy the user requirements with minimized resource cost. In contrast, Lin and Khazaei [84] treated function workflow as a Directed Acyclic Graph, and proposed a more complicated analytical model with consideration of resource, execution time, and the number of invocations. With this model, they introduced a heuristic algorithm based on four greedy strategies to compute the optimal resource configurations under the given conditions.

For resource auto-scaling in serverless, Schuler *et al.* [114] developed a Q-learning-based algorithm, which can allocate resources adaptive to the degree of concurrency of the applications without needing any prior knowledge on the incoming workload. Besides, Somma *et al.* [122] conducted experiments to show that provisioning more dockers for applications could result in better performance than scaling up the resources (CPU and memory) in each docker. Based on this observation, they further presented an auto-scaling approach for dockers using the Q-learning algorithm.

Opportunities. Machine learning algorithms have been evolving for the past decades, especially drawing much attention in recent years to promote the developments of diverse areas. We can attribute this observation to either the virtues of their prediction abilities based on historical data or the absence of user intervention while finding the optimal solution. These are well suited for the requirements of serverless computing. Nevertheless, at present, machine learning has not yet been widely exploited to optimize serverless computing due to its high latency requirements and complex settings. However, exploiting the vast amount of data generated in the process of function executions to efficiently facilitate serverless computing is appealing and promising.

Given the aforementioned three phases of resource management, we can see that different machine learning methods could be applied to each of them. For example, we can exploit Naive Bayesian [5] and Random Forest [31] to establish the performance model, and linear regression and gradient descent to achieve the minimal configuration for reduced cost and delay. Auto-scaling during function execution could be adjusted by using the feedback data (such as the delay of the function running) with a short period of trial and error. In this case, reinforcement learning [114], [137], [146] could be more suitable than others as they do not rely on expert experience.

5.2 Integration With Edge Computing

Next-generation technologies, such as the *Internet of Things* (IoT) [102], [141], often require high performance for time-bounded applications. To this end, they need fast data processing with low latency in communication to make respond

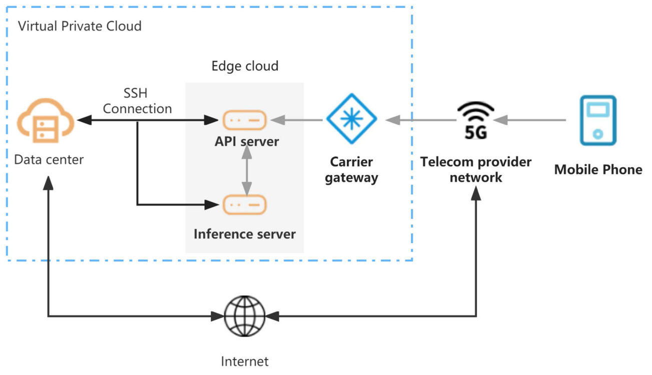


Fig. 7. A case to show how a mobile phone uses AWS Wavelength to make image object inference.

as quickly as possible. However, the cloud platforms, in general, need mobile/IoT devices to maintain a connection with a remote datacenter while serving the requests, which would incur high delay and overhead. Due to this problem, edge computing [71] has gained much popularity by virtues of its ability to bring the compute resources and data storage near to the location from which the mobile/IoT requests are made, achieving high performance and low latency, say in auto-driving [85] and smart city [86].

The basic elements in edge computing are IoT devices (e.g., mobile phones and wearable devices), each being able to collect data from its local environment and transmit them via wired or wireless network to a gathering point, called *smart edge device*, sitting between IoT devices and cloud datacenter [119]. In the edge device, the gathered data could be either processed locally for low latency and quick response or sent to the cloud for heavy computation. Fig. 7 is a case to show how a mobile phone exploits AWS Wavelength [6] to make image object inference, where the phone uses 5G to upload its task to the edge cloud, which after the task processing, either pushes back the inference results to the phone or interacts with the cloud datacenter for further processing via SSH connection.

Compared to the cloud datacenter, which typically employs VMs and containers to execute long-time and resource-heavy compute tasks, the edge platform usually has finite compute resources. Therefore, edge computing is appropriate for serverless computing, which is characterized by short and small tasks in lightweight environments.

Early work on deploying serverless architecture at the edge is proposed by Baresi *et al.* [25], [27] to facilitate the *Augmented Reality* (AR) for mobile devices. They adopted serverless in edge computing for latency-sensitive and data-intensive applications and developed distributed architecture among different nodes to collaborate in the allocation of resources and provisioning of functions. Aske and Zhao [17] then followed up this work and introduced a monitoring controller and customized configuration file to support serverless computing among multiple edge computing platforms. Tasks in this work could be scheduled by user-defined scheduling to determine the most suitable provider.

Based on their early work, Baresi *et al.* [26] took one step forward to propose *A3-E* — a deployed Faas model on the continuum, formed by the mobile devices, edge, and the cloud platform. A3-E could choose the best edge node for the offloaded function based on the execution context and requirements of the user. And it could reduce the latency and battery consumption at the same time. This idea is also adopted by Cicconetti *et al.* [37], who described an efficient framework to allocate stateless tasks to the most suitable edge node where a router calculates the destination based on the rapidly changing workloads and the network conditions. Similarly, Mittal *et al.* [95] also proposed a serverless edge computing framework, called *Mu* on Knative, which introduces a load balancer in the Ingress Gateway to receive the requests from edge device and serve as an entrance to edge serverless. Moreover, it uses a lightweight regression-based mechanism to predict incoming rates and fairly allocate resources.

Even though the discussed methods have tried to deploy serverless in the edge with optimization on function placement for latency reduction, we think there are still some problems that remain largely to be solved.

- *Network latency.* Low-latency distributed communication is critical for sharing states in edge computing, especially for the Internet of Vehicles, which needs to make a timely response to emergencies. However, as we have discussed before, the network communication in serverless is not sufficient in this case as it generally relies on the remote storage.
- *Security.* As opposed to the deployment of serverless in datacenters, which is in general protected by a secure environment, say, firewalls and *Trusted Execution Environment*, the edge devices are usually exposed to various attacks [56], especially, when the user data is required to process locally by serverless functions executed in containers with weak isolation at the edge. Thus, the security problems, including data privacy, need to be carefully considered while deploying the serverless in the edge clouds.
- *Function placement.* The decision making for function placements is further restricted when deploying the serverless computing on the edge equipment [106] since inappropriate distances between the function nodes, repository, and remote storage could result in significant communication latency, which is intolerant to the serverless computing.

Opportunities. With the developments of 5G, edge computing is gaining much popularity as the 5G base station with lower latency and computation power can enable the *Internet of Everything*. As a result, edge computing in conjunction with the deployed serverless functions can exploit the potential computation powers of thousands of edge devices to provide a high quality of service. As the number of edge devices grows up, the combination of edge computing and serverless will be promising in the future [19], [105]. However, not all applications developed for edge computing are suitable for serverless platforms [25]. Developers have to pay extra attention to the collaboration between the smart edge device, serverless function, and the cloud datacenter [106]. Additionally, the startup latency and the runtime environment also need to be considered given the constrained resources.

5.3 Developments of Benchmarking

Although serverless computing is thriving and prospering in recent years, it is in its infancy stage as there are still many problems that remained not yet been well addressed. One of those is the *serverless benchmarking*, which is often employed by the developers to facilitate the deployments of their functions to the platform, or by the users to help the selection of the best suitable framework for their workloads.

Some studies have been conducted on the developments of the serverless benchmark suites for the exploration of this emerging computing model [80], [89], [91], [115]. For example, Martins *et al.* [91] introduced a benchmark suite for comparing the performance among several serverless platforms provided by some mainstream cloud providers, such as Amazon and Google. The benchmarks are designed and implemented to cover the influence of memory, CPU, and language on service performance. Thus, they could be leveraged by the user to decide on the selection of suitable and economical platforms. Similar work with the same goal is also conducted by Maissen *et al.* [89], who proposed a benchmark called *Faasdom*, to help the user choose the most appreciated platform in terms of performance, reliability, or cost for their workloads. Unlike these studies on general workloads, Lin *et al.* [145] proposed a benchmark for bursty-traffic workloads in four types of application — Big-Data, Stream processing, Web Applications, and Machine Learning Inference. The use cases studied in this benchmark better fit the bursty characteristics of serverless, which could help developers analyze the platform, and guide the better design of serverless architecture.

As opposed to the foregoing studies in general, some research efforts only focus on a particular platform to explore the potentials of serverless computing. For example, Shahrad *et al.* [115] revealed the architectural implications of serverless and the characteristics based on the evaluation of Apache Openwhisk. Similarly, researchers from Microsoft then characterized the serverless workloads on Azure Functions [116]. With these benchmarks, one can gain a deep insight into the performance characteristics of the platform for further optimization. However, this insight is not sufficient for the overall performance improvements as sometimes the performance bottleneck is located in the programming logic of the applications, not the platform. Therefore, it is not adequate to only optimize the platform.

Given the deficiencies of prior works, some other researchers pay more attention to the underlying infrastructure and present some programming guidelines to help developers design better applications. For example, Chen *et al.* [147] proposed an open-source benchmark suite, called *ServerlessBench*, which is composed of 12 testcases, to evaluate the performance of applications and platforms in terms of function composition, startup latency, stateless execution, and performance isolation. Based on the test results, they further present some programming implications for each testcases as design guidelines. Such implications could assist both the developer with the optimization of application services and the provider with the improvement of the serverless platform as well. However, this benchmark suite lacks the notion of more complicated workloads such as mixed applications with different behavior patterns. As such, it cannot fully reflect some complex situations, such as burst scenarios or mixed scenarios, which still need to be further explored.

Opportunities. Benchmarking is an effective practice to discover the deficiencies in applications and platforms by quantifying their performance. However, as described, current studies on benchmarking are still insufficient to cope with the diverse use-cases of the serverless in practices. At present, most of the proposed benchmarks are over-simplified either to only cover certain particular cases or to over-emphasize the platform components while missing other important factors. As such, further efforts on benchmarking should be made for more comprehensive scenarios, which involve diverse application mixtures in particular. On the other hand, with the developments of autonomous driving and IoT-based technologies, serverless computing would also find new applications in these areas by supporting machine learning and streaming processing technologies. Consequently, the benchmarking for the integration of serverless computing to these technologies is also a pragmatic concern, and thus highly desired.

6 Conclusion

By virtue of its lightweight and simplicity of management, serverless computing as a new cloud computing model has been being widely studied and deployed for diverse application services. Our goal for this survey is not only to help engaged researchers learn the state-of-the-art of this cutting-edge technology but also to encourage those not in this area to make contributions to it.

To this end, we surveyed this technology in terms of its state-of-the-art, challenges, and opportunities. In particular, we first characterized serverless computing centering around its execution process and then identified five existing challenges that have not yet been fully explored, together with their respective state-of-the-art research. With the identified challenges in mind, we further selected three typical open-source serverless frameworks — OpenFaaS, OpenWhisk, and Kubeless — as the representatives to illustrate how the practical frameworks cope with the identified challenges. Finally, to envision the future of serverless computing, we also advocated some research opportunities in three carefully selected directions that have not yet been fully studied but are expected to embrace potential and promising values in future studies.

References

[1] P. Aditya *et al.*, “Will serverless computing revolutionize NFV?,” *Proc. IEEE*, vol. 107, no. 4, pp. 667–678, 2019.

[2] A. Agache *et al.*, “Firecracker: Lightweight virtualization for serverless applications,” in *Proc. 17th USENIX Symp. on Netw. Syst. Des. Implementation*, 2020, pp. 419–434.

[3] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, “COSE: Configuring serverless functions using statistical learning,” in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 129–138.

[4] I. E. Akkus *et al.*, “SAND: Towards high-performance serverless computing,” in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 923–935.

[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, “CherryPick: Adaptively unearthing the best cloud configurations for Big Data analytics,” in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 469–482.

[6] Amazon,‘ 5G edge computing infrastructure – AWS wavelength – amazon web services. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/wavelength/

[7] Amazon, Amazon dynamodb | nosql key-value database | amazon web services. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/dynamodb/

[8] Amazon, Cloud object storage – amazon S3 – amazon web services. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/s3/

[9] N. Amit and M. Wei, “The design and implementation of hyper-upcalls,” in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 97–112.

[10] J. Amsterdam and J. de Klerk, Compute engine: Virtual machines (VMS)-google cloud. Accessed: Jul. 2021. [Online]. Available: https://cloud.google.com/compute/

[11] J. Amsterdam and J. de Klerk, Retrying event-driven functions. Accessed: Jul. 2021. [Online]. Available: https://cloud.google.com/functions/docs/bestpractices/retries#make_retryable_event-driven_functions_idempotent

[12] S. Andrews and R. Escarez. Knative. Accessed: Jul. 2021. [Online]. Available: https://knative.dev/

[13] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, “Sprocket: A serverless video processing framework,” in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 263–274.

[14] Apache, Apache spark - unified engine for large-scale data analytics. Accessed: Jul. 2021. [Online]. Available: https://spark.apache.org/

[15] M. Armbrust *et al.*, “Above the clouds: A berkeley view of cloud computing,” Univ. California Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009–28, 2009.

[16] M. Armbrust *et al.*, “A view of cloud computing,” *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[17] A. Aske and X. Zhao, “Supporting multi-provider serverless computing on the edge,” in *Proc. 47th Int. Conf. Parallel Process. Companion*, 2018, pp. 1–6.

[18] M. S. Aslanpour *et al.*, “Serverless edge computing: Vision and challenges,” in *Proc. Australas. Comput. Sci. Week MultiConf.*, 2021, pp. 1–10.

[19] AWS, AWS IoT greengrass. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/cn/greengrass/

[20] A. Babenko and T. Thongsringklee, Amazon EC2 auto scaling. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/ec2/autoscaling/

[21] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Scalable atomic visibility with RAMP transactions,” *ACM Trans. Database Syst.*, vol. 41, no. 3, pp. 1–45, Jul. 2016.

[22] D. Barcelona-Pons , P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, “FaaS orchestration of parallel workloads,” in *Proc. 5th Int. Workshop Serverless Comput.*, 2019, pp. 25–30.

[23] D. Barcelona-Pons , M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, “On the FaaS track: Building stateful distributed applications with serverless architectures,” in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 41–54.

[24] D. Barcelona-Pons , M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, “On the FaaS track: Building stateful distributed applications with serverless architectures,” in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 41–54.

[25] L. Baresi and D. Filgueira Mendonça, “Towards a serverless platform for edge computing,” in *Proc. IEEE Int. Conf. Fog Comput.*, 2019, pp. 1–10.

[26] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, “A unified model for the mobile-edge-cloud continuum,” *ACM Trans. Internet Technol.*, vol. 19, no. 2, pp. 1–21, Apr. 2019.

[27] L. Baresi, D. F. Mendonça, and M. Garriga, “Empowering low-latency applications through a serverless edge computing architecture,” in *Proc. 6th IFIP WG 2.14 Eur. Conf. Service-Oriented Cloud Comput.*, 2017, pp. 196–210.

[28] A. Baumann *et al.*, “Composing OS extensions safely and efficiently with bascule,” in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 239–252.

[29] J. Beswick, V. Fulco, and M. Mevenkamp, Amazon eventbridge | event bus | amazon web services. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/eventbridge/

[30] J. Beulich and A. Cooper, Xen project. Accessed: Jul. 2021. [Online]. Available: https://xenproject.org/

[31] M. Bilal, M. Canini, and R. Rodrigues, “Finding the right cloud configuration for analytics clusters,” in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 208–222.

[32] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, “Putting the ”micro” back in microservice,” in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 645–650.

[33] E. A. Brewer, “Kubernetes and the path to cloud native,” in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, Art. no. 167.

[34] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “SEUSS: Skip redundant paths to make serverless fast,” in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.

[35] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, “Cirrus: A serverless framework for end-to-end ML workflows,” in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 13–24.

[36] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, “LADS: A high-performance framework for serverless parallel computing,” in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 239–251.

[37] C. Cicconetti, M. Conti, and A. Passarella, “A decentralized framework for serverless edge computing in the Internet of Things,” *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 2, pp. 2166–2180, Jun. 2021.

[38] CNCF, Who we are | cloud native computing foundation. Accessed: Jul. 2021. [Online]. Available: https://www.cncf.io/about/who-we-are/

[39] K. contributors, Linux KVM. Accessed: Jul. 2021. [Online]. Available: https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792

[40] P. J. Davis and A. Kocoloski, Apache CouchDB. Accessed: Jul. 2021. [Online]. Available: https://couchdb.apache.org/

[41] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. 6th Conf. Symp. Oper. Syst. Des. Implementation* , 2004, Art. no. 10.

[42] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-efficient and QoS-aware cluster management,” in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2014, pp. 127–144.

[43] M. Dounin and V. V. Bartenev, Nginx | high performance load balancer, web server, & reverse proxy. Accessed: Jul. 2021. [Online]. Available: https://www.nginx.com/

[44] D. Du *et al.*, “Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting,” in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2020, pp. 467–481.

[45] J. Duell, P. Hargrove, and E. Roman, “The design and implementation of berkeley lab’s Linux checkpoint/restart,” *J. Phys. Conf. Ser.*, vol. 46, pp. 494–499, 2003.

[46] S. Eismann *et al.*, “A review of serverless use cases and their characteristics,” *CoRR*, 2020, *arXiv:2008.11110*.

[47] T. Elgamal, “Costless: Optimizing cost of serverless computing through function fusion and placement,” in *Proc. IEEE/ACM Symp. Edge Comput.*, 2018, pp. 300–312.

[48] A. Ellis and B. Rheutan, OpenFaaS - serverless functions made simple. Accessed: Jul. 2021. [Online]. Available: https://www.openfaas.com/

[49] P. Emelyanov and A. Vagin, OpenFaaS - serverless functions made simple. Accessed: Jul. 2021. [Online]. Available: https://criu.org/Main_Page

[50] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, “Jockey: Guaranteed job latency in data parallel clusters,” in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 99–112.

[51] S. Fouladi *et al.*, “From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 475–488.

[52] S. Fouladi *et al.*, “Encoding, fast and slow: Low-latency video processing using thousands of tiny threads,” in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 363–376.

[53] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: A serverless-first, light-weight wasm runtime for the edge,” in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 265–279.

[54] Y. Gan *et al.*, “An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems,” in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2019, pp. 3–18.

[55] V. Giménez-Alventosa, G. Moltó, and M. Caballer, “A framework and a performance assessment for serverless MapReduce on AWS lambda,” *Future Gener. Comput. Syst.*, vol. 97, pp. 259–274, 2019.

[56] A. Glikson, S. Nastic, and S. Dustdar, “Deviceless edge computing: Extending serverless computing to the edge of the network,” in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, Art. no. 28.

[57] Google, Overview of memory management | android developers. Accessed: Jul. 2021. [Online]. Available: https://developer.android.com/topic/performance/memory-overview

[58] A. Haas *et al.*, “Bringing the web up to speed with webassembly,” in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 185–200.

[59] J. M. Hellerstein *et al.*, “Serverless computing: One step forward, two steps back,” in *Proc. 9th Biennial Conf. Innov. Data Syst. Res.*, 2019, pp. 1–9.

[60] E. Herbert and A. Guha, "A language-based serverless function accelerator," 2019, *arXiv:1911.02178*.

[61] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.

[62] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, Art. no. 11.

[63] C. L. Jan-Erik Rediger and P. Hertleif, Rust programming language. Accessed: Jul. 2021. [Online]. Available: https://www.rust-lang.org

[64] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, 2017, pp. 445–451.

[65] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, 2019, *arXiv:1902.03383*.

[66] I. Juma and J. Gustafson, Apache kafka. Accessed: Jul. 2021. [Online]. Available: https://kafka.apache.org/

[67] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 158–164.

[68] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSSt: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, 2016, pp. 185–201.

[69] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[70] A. Karcher and J. Hollan, Azure functions serverless compute. Accessed: Jul. 2021. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[71] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Gener. Comput. Syst.*, vol. 97, pp. 219–235, 2019.

[72] A. Khandelwal, A. Kejariwal, and K. Ramasamy, "Le taureau: Deconstructing the serverless landscape & a look forward," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2641–2650.

[73] M. Kiener, M. Chadha, and M. Gerndt, "Towards demystifying intra-function parallelism in serverless computing," in *Proc. 7th Int. Workshop Serverless Comput.*, 2021, pp. 42–49.

[74] Y. Kim and J. Lin, "Serverless data analytics with flint," in *Proc. IEEE 11th Int. Conf. Cloud Comput.*, 2018, pp. 451–455.

[75] A. Kivity *et al.*, "OSv–optimizing the operating system for virtual machines," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 61–72.

[76] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 427–444.

[77] N. Kratzke and P. Quint, "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study," *J. Syst. Softw.*, vol. 126, pp. 1–16, 2017.

[78] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion*, 2018, pp. 161–168.

[79] J. Larisch, J. Mickens, and E. Kohler, "Alto: Lightweight vms using virtualization-aware managed runtimes," in *Proc. 15th Int. Conf. Managed Lang. Runtimes*, 2018, pp. 1–7.

[80] H. Lee, K. Satyam, and G. C. Fox, "Evaluation of production serverless computing environments," in *Proc. 11th IEEE Int. Conf. Cloud Comput.*, 2018, pp. 442–450.

[81] A. Lehmann and A. Luzzardi, Docker swarm. Accessed: Jul. 2021. [Online]. Available: https://docs.docker.com/engine/swarm/

[82] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proc. 5th Int. Workshop Serverless Comput.*, 2019, pp. 37–42.

[83] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *Proc. IEEE Int. Conf. Service-Oriented System Eng.*, 2019, pp. 122–1225.

[84] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 615–632, Mar. 2021.

[85] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proc. IEEE*, vol. 107, no. 8, pp. 1697–1716, Aug. 2019.

[86] Z. Lv, D. Chen, R. Lou, and Q. Wang, "Intelligent edge computing based on machine learning for smart city," *Future Gener. Comput. Syst.*, vol. 115, pp. 90–99, 2021.

[87] A. Madhavapeddy *et al.*, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Comput. Archit. News*, vol. 41, pp. 461–472, 2013.

[88] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing serverless computing: Introducing an adaptive function placement algorithm," in *Proc. 29th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2019, pp. 203–213.

[89] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "FaaSdom: A benchmark suite for serverless computing," in *Proc. 14th ACM Int. Conf. Distrib. Event-Based Syst.*, 2020, pp. 73–84.

[90] F. Manco *et al.*, "My VM is lighter (and safer) than your container," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 218–233.

[91] H. Martins, F. Araújo, and P. R. da Cunha, "Benchmarking serverless computing platforms," *J. Grid Comput.*, vol. 18, no. 4, pp. 691–709, 2020.

[92] O. Mashayekhi, H. Qu, C. Shah, and P. Levis, "Execution templates: Caching control plane decisions for strong scaling of data analytics," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 513–526.

[93] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 453–468.

[94] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 453–468.

[95] V. Mittal *et al. Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds*. New York, NY, USA: ACM, 2021, pp. 168–181.

[96] B. Moffatt and D. Quintão, AWS lambda – serverless compute. Accessed: Jul. 2021. [Online]. Available: https://aws.amazon.com/lambda/

[97] A. Mosayyebzadeh *et al.*, "Supporting security sensitive tenants in a bare-metal cloud," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 587–602.

[98] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 57–70.

[99] J. Ousterhout *et al.*, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 1–55, Aug. 2015.

[100] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *Proc. IEEE World Congr. Serv.*, 2019, pp. 206–211.

[101] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 325–341.

[102] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the Internet of Things: A case study," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 1275–1284, Apr. 2018.

[103] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 193–206.

[104] R. Rabbah and M. Thömmes, Apache openwhisk is a serverless, open source cloud platform. Accessed: Jul. 2021. [Online]. Available: http://openwhisk.apache.org/

[105] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge AI," in *Proc. 2nd USENIX Workshop Hot Top. Edge Comput.*, 2019, pp. 1–10.

[106] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Gener. Comput. Syst.*, vol. 114, pp. 259–271, 2021.

[107] T. Reeder and T. Ceylan, Fn project - the container native serverless framework. Accessed: Jul. 2021. [Online]. Available: https://fnproject.io/

[108] T. Reeder and P. Nasser, Iron.io open source - functions. Accessed: Jul. 2021. [Online]. Available: https://open.iron.io/

[109] B. Richardson and S. Hemminger, DPDK. Accessed: Jul. 2021. [Online]. Available: https://www.dpdk.org/

[110] T. Risberg and S. Andrews, Riff is for functions. Accessed: Jul. 2021. [Online]. Available: https://projectriff.io/

[111] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 121–133.

[112] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the IBM cloud," in *Proc. 19th Int. Middleware Conf. Ind.*, 2018, pp. 1–8.

[113] S. Sanfilippo and P. Noordhuis, Redis. Accessed: Jul. 2021. [Online]. Available: https://redis.io/

[114] L. Schuler, S. Jamil, and N. Kühl, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *Proc. IEEE/ACM 21st Int. Symp. Cluster Cloud Internet Comput.*, 2021, pp. 804–811.

[115] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 1063–1075.

[116] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.

[117] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.

[118] V. Shankar *et al.*, "Serverless linear algebra," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 281–295.

[119] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[120] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 419–433.

[121] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 1–13.

[122] G. Somma, C. Ayimba, P. Casari, S. P. Romano, and V. Mancuso, "When less is more: Core-restricted container provisioning for serverless computing," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2020, pp. 1153–1159.

[123] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.

[124] V. Sreekanti *et al.*, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2438–2452, 2020.

[125] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: Ephemeral endpoints for serverless networking," in *Proc. ACM Symp. Cloud Comput., Virtual Event*, 2020, pp. 16–29.

[126] C.-C. Tsai *et al.*, "Cooperation and security isolation of library OSes for multi-process applications," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.

[127] Tuna and A. M. Gotor, Kubeless. [Online]. Available: https://kubeless.io/

[128] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with STeLLAR," in *Proc. IEEE Int. Symp. Workload Characterization*, 2021, pp. 51–62.

[129] S. van Stijn and V. Demeester, Docker. Accessed: Jul. 2021. [Online]. Available: https://www.docker.com/

[130] S. Vasani and T.-C. Chen, Serverless functions for kubernetes - fission. Accessed: Jul. 2021. [Online]. Available: https://fission.io/

[131] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation*, 2014, pp. 301–316.

[132] S. Venkataraman *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 374–389.

[133] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 363–378.

[134] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for MapReduce environments," in *Proc. 8th ACM Int. Conf. Autonomic Comput.*, 2011, pp. 235–244.

[135] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–17.

[136] F. Voznika, A. Scannell, and N. Lacasse, Google gVisor: Container runtime sandbox. [Online]. Available: https://github.com/google/gvisor

[137] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1288–1296.

[138] K. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proc. 14th Eur. Conf.*, 2019, pp. 1–16.

[139] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.

[140] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai, "Serverless Big Data processing using matrix multiplication as example," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 358–365.

[141] A. Whitmore, A. Agarwal, and L. D. Xu, "The Internet of Things - A survey of topics and trends," *Inf. Syst. Front.*, vol. 17, no. 2, pp. 261–274, 2015.

[142] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 1–10.

[143] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 83–97.

[144] Z. Xu, T. Mauldin, Z. Yao, S. Pei, T. Wei, and Q. Yang, "A bus authentication and anti-probing architecture extending hardware trusted computing base off CPU chips and beyond," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 749–761.

[145] L. Yanying, Y. Kejiang, L. Yongkang, L. Peng, Y. Tang, and C.-Z. Xu, "BBServerless: Burst traffic benchmark for serverless," in *Proc. IEEE 14th Int. Conf. Cloud Comput.*, 2021, pp. 45–60.

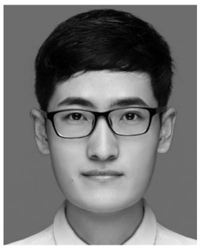
[146] H. Yu, H. Wang, J. Li, and S.-J. Park, "Harvesting idle resources in serverless computing via reinforcement learning," 2021, *arXiv:2108.12717*.

[147] T. Yu *et al.*, "Characterizing serverless platforms with serverlessbench," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 30–44.

[148] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 1187–1204.

[149] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 1–12.

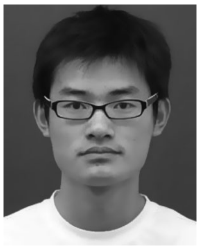
[150] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: A programming framework for serverless computing," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 328–343.



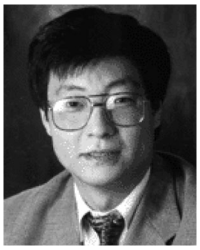
Yongkang Li received the BS degree from Shandong University, China, in 2019. He is currently working toward the master's degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China. His research interests include cloud computing, system software, virtualization and RDMA technology.

Yanying Lin received the BS degree from the Southwest University of Sciences and Technology, China, in 2016. He is currently working toward the master's degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China. His research interests span distributed systems, serverless computing, and their intersection with machine learning.

Yang Wang received the BSc degree in applied mathematics from the Ocean University of China, Qingdao, China, in 1989, the MSc degree in computer science from Carleton University, Ottawa, Canada, in 2001, and the PhD degree in computer science from the University of Alberta, Edmonton, Canada, in 2008. He currently works with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, as a professor. His research interests include cloud computing, Big Data analytics, and Java virtual machine on multicores. He is an Alberta Industry R&D associate (2009–2011), and a Canadian fulbright scholar (2014–2015).



Kejiang Ye received the BSc and PhD degrees in computer science from Zhejiang University in 2008 and 2013 respectively, and also joint the PhD degree with The University of Sydney from 2012 to 2013. After graduation, he works as postdoc researcher with Carnegie Mellon University from 2014 to 2015 and Wayne State University from 2015 to 2016. He is currently a professor with Shenzhen Institutes of Advanced Technology, Chinese Academy of Science. His research interests focus on the performance, energy, and reliability of cloud computing and network systems.



Chengzhong Xu (Fellow, IEEE) received the BSc and MSc degrees from Nanjing University in 1986, and 1989, respectively, and the PhD degree from the University of Hong Kong in 1993, all in computer science and engineering. Currently, he is a chair professor of Computer Science and the dean with the Faculty of Science and Technology, University of Macau, China. His recent research interests are in cloud and distributed computing, systems support for AI, smart city and autonomous driving. He has published more than 400 papers in journals and conferences. He serves on a number of journal editorial boards and the chair for IEEE TCDP from 2015 to 2020.