The Serverless Computing Survey: A Technical Primer for Design Architecture

ZIJUN LI, LINSONG GUO, JIAGAN CHENG, and QUAN CHEN, Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

BINGSHENG HE, National University of Singapore, Department of Computer Science, Singapore

MINYI GUO, Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

The development of cloud infrastructures inspires the emergence of cloud-native computing. As the most promising architecture for deploying microservices, serverless computing has recently attracted more and more attention in both industry and academia. Due to its inherent scalability and flexibility, serverless computing becomes attractive and more pervasive for ever-growing Internet services. Despite the momentum in the cloud-native community, the existing challenges and compromises still wait for more advanced research and solutions to further explore the potentials of the serverless computing model. As a contribution to this knowledge, this article surveys and elaborates the research domains in the serverless context by decoupling the architecture into four stack layers: Virtualization, Encapsule, System Orchestration, and System Coordination. We highlight the key implications and limitations of these works in each layer, and make suggestions for potential challenges to the field of future serverless computing.

Note: This paper has been accepted by ACM Computing Surveys (CSUR), and the current e-print version is our major revision. For a complete view, please visit ACM CSUR.

1 INTRODUCTION

1.1 Definition of Serverless Computing

Traditional Infrastructure-as-a-Service (IaaS) deployment mode demands a long-term running server for sustainable service delivery. However, this exclusive allocation needs to retain resources regardless of whether the user application is running or not. Consequently, it results in low resource utilization in current data centers by only about 10% on average, especially for an online service with a diurnal pattern. The contradiction attracts the development of a platform-managed on-demand

Authors' addresses: Zijun Li, lzjzx1122@sjtu.edu.cn; Linsong Guo, gls1196@sjtu.edu.cn; Jiagan Cheng, chengjiagan@sjtu.edu.cn; Quan Chen, chen-quan@cs.sjtu.edu.cn, Department of Computer Science and Engineering, Shanghai Jiao Tong University, China; BingSheng He, hebs@comp.nus.edu.sg, National University of Singapore, Department of Computer Science, Singapore; Minyi Guo, guo-my@cs.sjtu.edu.cn, Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.
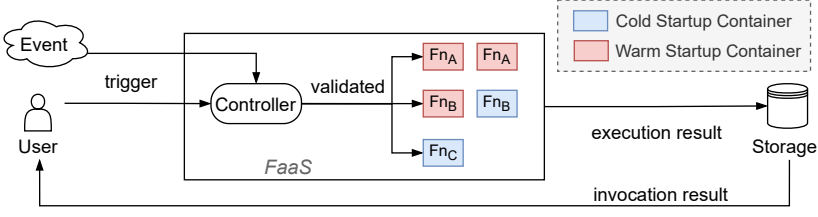
Fig. 1. The example of an asynchronous invocation in serverless computing.

service model to attain higher resource utilization and lower cloud computing costs. To this end, serverless computing was put forward, and most large cloud vendors such as Amazon, Google, Microsoft, IBM, and Alibaba have already offered such elastic computing services.

In the following, we will first review the definition given in Berkeley View [65], and then we will give a broader definition. We believe that a narrow perception of the FaaS-based serverless model may weaken its advancement. So far, there is no formal definition of serverless computing. The common acknowledged definitions from Berkeley View [65] are presented as follows:

- *Serverless Computing = FaaS(Function-as-a-Service) + BaaS(Backend-as-a-Service)*. One fallacy is that 'Serverless' is interchangeable with 'FaaS', which is revealed in a recent interview [78]. To be precise, they both are essential to serverless computing. *FaaS* model enables the function isolation and invocation, while *BaaS* provides overall backend support for online services.
- In the *FaaS* model (aka Lambda paradigm), an application is sliced into functions or function-level microservices [26, 45, 57, 65, 117, 141]. The function identifier, the language runtime, the memory limit of one instance, and the function code blob URI (Uniform Resource Identifier) together define the existence of a function [94].
- The *BaaS* covers a wide range of services that any application relies on can be categorized into it. For example, the cloud storage (Amazon S3 and DynamoDB), the message bus system for passing (Google cloud pub/sub), the message notification service (Amazon SNS), and DevOps tools (Microsoft Azure DevOps).

To depict the serverless computing model, we take the asynchronous invocation in Figure 1 as an example. The serverless system receives triggered API queries from the users, validates them, and invokes the functions by creating new sandboxes (aka the **cold startup** [15, 28, 65]) or reusing running warm ones (aka the **warm startup**). The isolation ensures that each function invocation runs in an individual container or a virtual machine assigned from an access-control controller. Due to the event-driven and single-event processing nature, the serverless system can be triggered to provide on-demand isolated instances and scale them horizontally according to the actual application workload. Afterwards, each execution worker accesses a backend database to save execution results [23]. By further configuring triggers and bridging interactions, users can customize the execution for complex applications (e.g., building internal event calls in a $\{Fn_A, Fn_B, Fn_C\}$ pipeline).

In the broader scenario, we think that the serverless computing model should be identified with the following features:

- **Auto-scaling.** Auto-scalability should not be only narrowed to the FaaS model (e.g., container black boxes as scheduling units in OpenWhisk [134]). The indispensable factor in identifying a serverless system is performing horizontal and vertical scaling when accommodating workload dynamics. Allowing an application to scale the number of instances to zero also

introduces a worrisome challenge - cold startup. When a function experiences the cold startup, instances need to start from scratch, initialize the software environment, and load application-specific code. These steps can significantly drag down the service response, leading to QoS (Quality-of-Service) violations.

- **Flexible scheduling.** Since the application is no longer bound to a specific server, the serverless controller dynamically schedules applications according to the resource usage in the cluster, while ensuring load balancing and performance assurances. Moreover, the serverless platform also takes the multi-region collaboration into account [154]. For a more robust and available serverless system, flexible scheduling allows the workload queries to be distributed across a broader range of regions [119]. It avoids serious performance degradation or damage to the service continuity in case of unavailable or crash nodes.

- **Event-driven.** The serverless application is triggered by events, such as the arrival of REST-ful HTTP queries, the update of a message queue, or new data to a storage service. By binding events to functions with triggers and rules, the controller and functions can use metadata encapsulated in context attributes. It makes relationships between events and the system detectable, enabling different collaboration responses to different events. Cloud-Native Computing Foundation (CNCF) serverless group also published CloudEvents specifications for commonly describing event metadata to provide interoperability.

- **Transparent development.** On the one hand, managing underlying host resources will no longer be a bother for application maintainers. It is because they are agnostic about the execution environment. Simultaneously, cloud vendors should ensure available physical nodes, isolated sandboxes, software runtimes, and computing power while making them transparent to maintainers. On the other hand, serverless computing should also integrate DevOps tools to help deploy and iterate more efficiently.

- **Pay-as-you-go.** The serverless billing model shifts the cost of computing power from a capital expense to an operating expense. This model eliminates the requirement from users to buy exclusive servers based on the peak load. By sharing network, disk, CPU, memory, and other resources, the pay-as-you-go model only indicates the resources that applications actually used [1, 2, 26], no matter whether the instances are running or idle.

We regard an elastic computing model with the above five features incorporated as the key to the definition of serverless computing. Along with the serverless emergence, application maintainers would find it more attractive that resource pricing is billed based on the actual processing events of an application rather than the pre-assigned resources [2]. Nowadays the serverless computing is commonly applied in backend scenarios for batch jobs, including data analytics (e.g., distributed computing model in PyWren [64]), machine learning tasks (e.g., deep learning) [78, 111], and event-driven web applications.

1.2 Survey Method by the Layered Serverless Architecture

Several surveys in serverless computing have discussed the characteristics of serverless generalization [15, 52, 65, 112, 116, 144]. However, they only propose literature reviews from a high-level perspective while ignoring to provide enough architecture implications. As a result, researchers and serverless vendors may find it struggling to grasp and comprehend each issue in the real serverless architecture. In the lack of systematic knowledge, challenges and proposed solutions will lack high portability and compatibility for various serverless systems. To this end, this survey is inspired to propose a layered design and summarize the research domains from different views. It can help researchers and practitioners to further understand the nature of serverless computing. As shown in Figure 2, we analyze its design architecture with a bottom-up logic and decouple the serverless
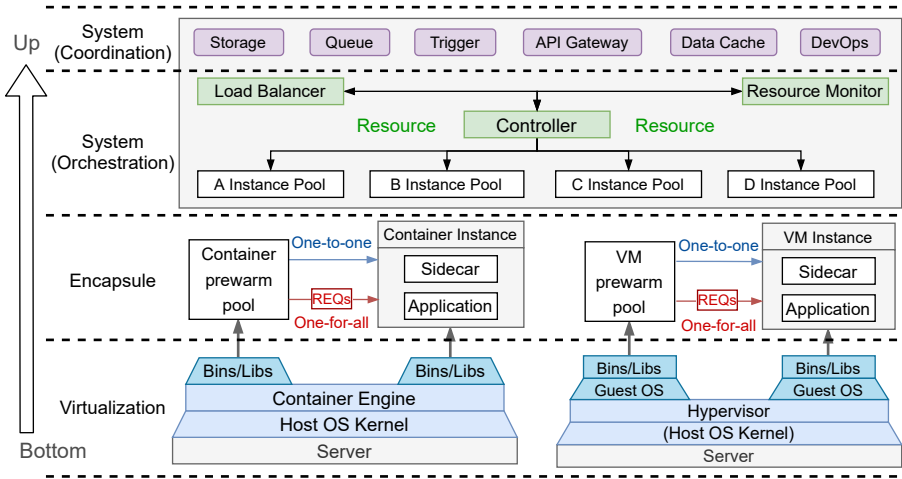
Fig. 2. General implementation of the serverless architecture.

computing architecture into four stack layers: Virtualization, Encapsule, System Orchestration, and System Coordination.

Virtualization layer. The Virtualization layer enables function isolation within a performance and functionality secured sandbox. The sandbox serves as the runtime for application service code, runtime environment, dependencies, and system libraries. To prevent access to resources in the multi-application or multi-tenant scenarios, cloud vendors usually adopt containers/virtual machines to achieve isolation. Currently, the popular sandbox technologies are Docker [41], gVisor [49], Kata [67], Firecracker [3], and Unikernel [86]. Section 2 introduces these solutions to isolate functions and analyze their pros and cons.

Encapsule layer. Various middlewares in the Encapsule layer enable customized function triggers and executions, and they also provide data metrics collection for communicating and monitoring. We call all these additional middlewares the *sidecar*. It separates the service's business logic and enables loose coupling between the functions and the underlying platform. Meanwhile, to speed up instance startup and initialization, the prewarm pool is commonly used in the Encapsule layer [44, 97, 104, 105]. In addition, serverless systems may use prediction by analyzing the load pattern to prewarm each by one-to-one approach [118, 146], or build a template for all functions to dynamically install requirements (REQs) according to the runtime characteristics by a one-for-all approach. Those concepts are introduced and compared in Section 3.

System Orchestration layer. The System Orchestration layer allows users to configure triggers and bind rules, ensuring the high availability and stability of the user application by dynamically adjusting as load changes. Through the cloud orchestrator, the combination of online and offline scheduling can avoid resource contention, recycle idle resources and ease the performance degradation for co-located functions. The above implementations are also typically integrated into container orchestration services (e.g., Google Kubernetes and Docker Swarm). While in the serverless system, **resource monitor**, **controller**, and **load balancer** are consolidated to resolve scheduling challenges [4, 32, 50, 57, 66, 70, 88, 139]. They enable the serverless system to achieve scheduling optimizations in three different levels: resource-level, instance-level, and application-level, respectively. Section 4 detailly analyzes the scheduling methodology from these three angles.

System Coordination layer. The System Coordination layer consists of a series of Backend-as-a-Service (BaaS) components that use unified APIs and SDKs to integrate backend services into

functions. Distinctly, it differs from the traditional middlewares that use local physical services outside the cloud. These BaaS services provide the storage, queue service [94, 99], trigger binding [75, 77], API gateway, data cache [6, 7], DevOps tools [24, 25, 63, 122], and other customized components for better meeting the System Orchestration layer's flexibility requirements. Section 5 discusses these essential BaaS components in a serverless system.

Each stack layer plays an essential role in the serverless architecture. Therefore, based on the above hierarchy, we conclude the contributions of this survey as follows:

(1) Introduce the serverless definition and summarize the features.
(2) Elaborate the architecture design based on a four-layer hierarchy, and review the significant and representative works in each layer.
(3) Analyze the current serverless performance and its limitations.
(4) Explore the challenges, limitations, and opportunities in serverless computing.

The rest of the survey is organized as follows: Sections 2-5 introduce the four stack layers and elaborate current research domains in serverless computing. Section 6 analyzes several factors that degrade performance, and compares the current production serverless systems. Finally, the challenges, limitations, and opportunities of serverless computing are given in Section 7-8. We conclude this paper in Section 9.

2 VIRTUALIZATION LAYER

Whenever a user function is invoked in serverless computing, it will be loaded and executed within a virtualized sandbox. A function can either reuse a warm sandbox or create a new one, but usually not co-run with different user functions. In this premise, most of the concerns in virtualization are isolation, flexibility, and low startup latency. The isolation ensures that each application process runs in the demarcated resource space, and the running process can avoid interference by others. The flexibility is demonstrated by the ability of testing and debugging, and the additional supports for extending over the system. Low startup latency requires a fast response for the sandbox creation and initialization. The current sandboxing mechanism on the Virtualization layer is broken into four representative categories: traditional VM (Virtual Machine), container, secure container, and Unikernel. Table 1 compares these mainstream approaches in several respects.

In the table, "Startup latency" represents the response latency of cold startup. "Isolation level" indicates the capacity of functions running without interference by others. "OSkernel" shows whether the kernel in GuestOS is shared. "Hotplug" allows the function instance to start with minimal resources (CPU, memory, virtio blocks) and add additional resources at runtime. "OCI supported" means whether it provides the Open Container Initiative (OCI), an open governance structure for expressing container formats and runtimes. Moreover, "✓" in all tables of this survey means this technique or strategy is used, and vice versa.

The traditional VM-based isolation adopts a VMM (virtual machine manager, e.g., hypervisor) which provides virtualization capabilities to GuestOS. It can also mediate access to all shared resources by provided interfaces (or using Qemu/KVM). With snapshots, VM shows high flexibility in quick failsafe when patch performing on applications within each VM instance. Though VM provides a strong isolation mechanism and flexibility, it lacks the benefits of lower startup latency for user applications (usually >1000ms). This tradeoff is fundamental in the serverless computing, where functions are small while the relative overhead of VMM and guest kernel is high.

Container customization: provide high flexibility and performance.
Another common function isolation mechanism in serverless computing is using containers. The container engine leverages the Linux kernel to isolate resources, and creating containers as different processes in HostOS [19, 92]. Each container shares the HostOS kernel with the read-only

Table 1. Techniques in the Virtualization layer.

Virtualization	Startup latency	Isolation level	OSkernel	Hotplug	Hypervisor	OCI supported	Backed by
Traditional VM	>1000ms	Strong	unsharing		✓	✓	/
Docker [41]	50ms-500ms	Weak	host-sharing	✓		✓	Docker
SOCK [101]	10ms-50ms	Weak	host-sharing	✓		✓	/
Hyper-V [58]	>1000ms	Strong	unsharing	✓	✓	✓	Microstft
gVisor [49]	50ms-500ms	Strong	unsharing		✓	✓	Google
Kata [67]	50ms-500ms	Strong	unsharing	✓	✓	✓	OpenStack
FireCracker [3]	50ms-500ms	Strong	unsharing		✓	✓	Amazon
Unikernel [86]	10ms-50ms	Strong	Built-in		✓		Docker

attribute, which typically includes binaries and libraries. The high flexibility is also attached to the container with the UnionFS (Union File System), which enables the combination of the layered container image by read-only and read-write layers. Essentially, a container achieves the isolation through namespace to enable processes sharing the same system kernel and Linux Cgroups to set resource limits. Without hardware isolation, container-based sandboxing shows lower startup latency than coarse-grained consolidation strategies [11, 147] in hypervisor-based VMs.

The representative container engine is Docker [41]. Docker packages software into a standardized RunC container adapted to the environment requirements, including libraries, system tools, code, and runtime. Docker container has been widely applied to various serverless systems for its lightweight nature. Some works further optimize the container runtime for better adaption to the application requirements in the serverless system. SOCK [101] proposes an integration solution for serverless RunC containers, where redundant mechanisms in Docker containers are discarded in this lean container. By only constructing a root file system, creating communication channels, and imposing isolation boundaries, SOCK container makes serverless systems running more efficiently in startup latency and throughput. The startup latency of SOCK container is reduced to 10ms-50ms, comparing to docker containers that usually takes 50ms-500ms. Different from condensing redundance in lean containers, as additional tools (e.g., debuggers, editors, coreutils, shell) enriching the container and increasing the image size, CNTR [130] splits the container image into "fat" and "slim" parts. A user can independently deploy the "slim" image and expand it with additional tools by dynamically attaching the "fat" image to it. The evaluation of CNTR shows that the proposed mechanism can significantly improve the overall performance and effectively reduce the image size when extensively applied in the data center.

Secure Container: compromise security with high flexibility and performance.

At the same time, security concerns in Virtualization Layer arise for the relatively low isolation level of containers. Any process-based solution must include a relaxation of the security model for its insufficiency for mutually-untrusted functions. It requires containers to prevent code vulnerabilities in the case of shared kernel architecture. Side-channel attacks such as Meltdown [84], Zombieload [114], Spectre [72] prompt mitigation approaches toward vulnerabilities, especially for multi-tenants in serverless context. In this case, container isolation should concern with preventing privilege escalation, information, and communication disclosure side channels [3]. The state-of-the-art solution to this issue is leveraging Secure Container. For example, Microsoft proposes their Hyper-V Container for Windows [58]. Hyper-V offers enhanced security and broader compatibility. Each instance runs inside a highly optimized MicroVM and does not share its kernel with others on the same host. However, it is still a heavy-weight virtualization that can introduce more than 1000ms of startup latency. In Google gVisor [49], the kernel in it acts as a non-privileged process to
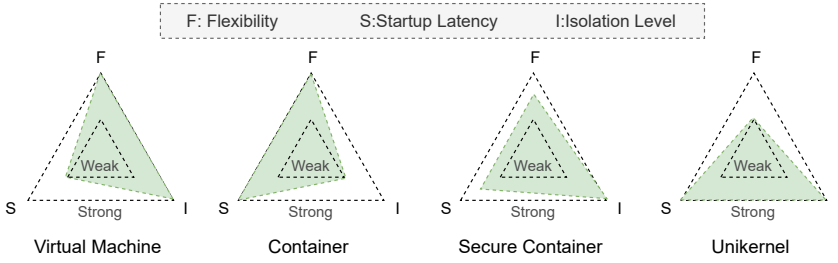
Fig. 3. The flexibility, startup latency, and isolation level of four virtualization mechanisms.

restrict *syscalls* that called in userspace. However, the overhead introduced during interception and processing *syscalls* in a sandbox is high. As a result, it is not well-suited for applications with heavy *syscalls*. In order to isolate different tenants with affordable overhead, FireCracker [3] creates MicroVMs by customizing VMM for cloud-native applications. Each Firecracker sandbox runs in user space and is restricted by Seccomp, Cgroup, and Namespace policies. With a container engine built-in MicroVMs, Kata [67] adopts an agent to communicate with the kata-proxy located on the host through the hypervisor, thus achieve a secure environment in a lightweight manner. Both FireCracker and Kata containers can significantly reduce startup latency and memory consumption, and they all need only 50ms-500ms to start a sandbox. With secure containers, it can provide complete and strong isolation from the HostOS and other tenants, at the cost of the limited flexibility in the condensed MicroVM. Meanwhile, it still results in instances' long startup latency due to the additional application initialization, e.g., JVM or Python interpreter setup.

Specialized Unikernel: enhance flexibility with high security and performance.

Another emerging virtualization technique is called Unikernel [86], which leverages libraryOS, including series of essential dependent libraries to construct a specialized, single-address-space machine image. Because the Unikernel runs as a built-in GuestOS, the compile-time invariance rules out runtime management, which significantly reduces the applicability and flexibility of Unikernel. However, unnecessary programs or tools such as ls, cd, tar are not contained, so the image size of a Unikernel is smaller (e.g., 2MB by mirage-skeleton [95] that compiled from Xen), the startup latency is much less (e.g., startup within 10ms), and the security is more substantial than containers. Based on it, LightVM [90] replaces the time-consuming XenStore and implements the split tool stack, separating functionality that runs periodically from that which must be carried out, thus improving efficiency and reducing VM startup latency. From the perspective of the software ecosystem, to solve the challenge that traditional applications are struggling to be transplanted to the Unikernel model [86, 113], Olivier proposes HermitTux [102], a Unikernel model compatible with Linux binary. HermitTux makes the Unikernel model compatible with Linux Application Binary Interface while retaining the benefits of Unikernel. However, Unikernel is not adaptable for developers once built, making it inherently inflexible for applications, let alone the terrible DevOps environment. Furthermore, in heterogeneous clusters, the heterogeneity of the underlying hardware forces Unikernel to update as drivers change, making it the antithesis of serverless philosophy.

Tradeoffs among Security, performance, and flexibility.

At last, we make the indicatrix diagram of these four technologies in Figure 3 to show the tradeoffs among security, performance, and flexibility. To conclude, hypervisor-based VM shows better isolation and flexibility, while the container can make the instance start faster and flexible to customize the runtime environment. Secure Container offers both high security and relatively low

Table 2. Works in Encapsule layer.

Representative work	Template	Static image	Pool	Exclusive	Fixed size	Predict /Heuristic	REQs	C/R	Sidecar based	Imp
Pause container [55, 94]		✓		✓				✓		/
Azure functions [105]		✓	✓	✓	✓			✓		AWS
Fission [44]		✓	✓	✓	✓					Kubernetes
Adaptive Warm-up [146]		✓		✓		✓			✓	Kubernetes
Serverless in the Wild [118]		✓		✓		✓			✓	OpenWhisk
Replayable Execution [140]		✓						✓	✓	FaaS FW
Catalyzer [42]	✓	✓						✓	✓	gVisor-based
Mohan *et al.* [97]	✓		✓				✓			OpenWhisk
Apache OpenWhisk [104]	✓	✓	✓		✓					/
SOCK [101]	✓					✓	✓		✓	OpenLambda

startup latency with flexibility compromise. Unikernel demonstrates great potential in terms of performance and security, but it loses flexibility. When offering adaptable images in the production environment by either virtualization mechanism, it is also critical to avoid that built ones are signed and originated from an unsafe pedigree, with the solutions [69, 128] by keeping continuous vulnerability assessment and remediation program.

3 ENCAPSULE LAYER

A cold startup in serverless computing may occur when the function fails to capture a warm running container, or experiences a bursty load. In the former, a function is invoked for the first time, or scheduled with a longer invocation interval than the instance lifetime. The typical characteristic is that instances (or pods) must get started from scratch. In the latter case of a bursty load, instances need to perform horizontal scaling during a surge in user workloads. Function instances will autoscale as load changes to ensure adequate resource allocation. Besides taking less than one second to prepare a sandbox in the Virtualization layer, the initialization of software environment, e.g., load Python libraries; and application-specific user code can dwarf the former [42, 65, 83, 101, 117]. While we can provide a more lightweight sandboxing mechanism to reduce the cold startup latency in the virtualization layer, the state-of-the-art sandboxing mechanism may not demonstrate perfect compatibility for containers or VMs when migrated to the existing serverless architecture. In response to the tradeoff between performance and compatibility, an efficient solution is to prewarm instances in the Encapsule layer. This approach is known as the prewarm startup, which has been widely researched. Representative work about instance prewarm is listed in Table 2.

Before giving the detailed analysis and comparison, we first describe the taxonomy in each column. "Template" reflects whether the cold startup instance comes from a template. "Static image" shows whether the VM/container image for prewarm disables dynamically updating in each cold startup. "Pool" indicates whether there is a prewarm pool for function cold startups. "Exclusive" and "Fixed-size" represents whether the prewarmed instance and its prewarm pool is exclusive and size-fixed. "Predict/Heuristic" points out whether the prediction algorithm or heuristic-based method is used to prewarm instances. "REQs" reflects whether the runtime libraries and packages are dynamically loading and updating in the prewarm instance. "C/R" reflects whether it supports checkpoint and restore to accelerate the startup. "Sidecar based" represents whether the relevant technologies can be implemented or integrated into the sidecar. "Imp" indicates where it is implemented.

There are two common prewarm startup approaches: **one-to-one** prewarm startup and **one-for-all** prewarm startup. In the one-to-one prewarm startup, each function instance is prewarmed from a size-fixed pool, or by dynamic prediction based on the historical workload traces. While in the one-for-all prewarm startup, instances of all functions are prewarmed from cached sandboxes, which are pre-generated according to a common configuration file. When a cold startup occurs, the function only needs to specialize these pre-initialized sandboxes by importing function-specific code blob URI and settings. For higher scalability and lower instance initialization latency, C/R (Checkpoint/Restore) is also combined with prewarmed instances in a serverless system. C/R is a technique that can freeze a running instance, make a checkpoint into a list of files, and then restore the running state of the instance at the frozen point. A common pattern in serverless implementations is to pause the instance when idle to save resources, and then recovery it for reusing when invoked [55, 94].

One-to-one prewarm by size-fixed pool: effective but resource-unfriendly.

The one-to-one strategy prewarms instances for each function, which means it is crucial to determine the warm-up time. Otherwise, a slow or quick warm-up cycle will either result in less efficiency at reducing cold startup or unnecessary prewarmed instances wasting a mass of resources. The current solution is to build an exclusive prewarm pool with a fixed size for each function to maximize the service stability. For example, Azure Functions [105] warms up instances of each function by setting up a fixed-size prewarm pool. Once the always-ready instance is occupied, prewarmed instances will be active and continue to buffer until reaching the limit. The open-sourced Fission [44] also prewarms like Azure Function. It introduces a component called *poolmgr*, which manages a pool of generic instances with a fixed pool size and injects function code into the idle instances to reduce the cold start latency.

One-to-one prewarm by predictive warm-up: ways to make resource friendly.

One-to-one strategy prewarms exclusive instances in each size-fixed prewarm pool, and load codes whenever invocations arrive. In this pattern, it is a safe strategy without introducing other security concerns. However, it would produce massive idle instances in the background and make the serverless system resource unfriendly. Such a deficiency inspires researchers to propose more flexible prewarm strategies like using prediction-based and heuristic-based methods. Xu *et al.* [146] design an AWU (Adaptive Warm-up) (AWU) strategy by leveraging the LSTM (Long Short-Term Memory) networks to discover the dependence relationships based on the historical traces. It predicts the invoking time of each function to prewarm the instances, and initialize the prewarmed containers according to the ACPS (Adaptive Container Pool Scaling) strategy once AWU fails. Shahrad *et al.* [118] propose a practical resource management policy for the one-to-one prewarm startup. By characterizing the FaaS workloads, they dynamically change the instance lifetime of the recycling and provisioning instances according to the time series prediction. CRIU (Checkpoint/Restore In Userspace) [39] is a software tool on Linux to implement Checkpoint/Restore functions. Replayable Execution [140] makes improvements based on CRIU, using "mmap" to map checkpoint files to memory and leveraging the Copy-on-Write in OS to share cold data among multiple containers. By exploiting the intensive-deflated execution characteristics, it reduces the container's cold startup time and memory usage.

One-for-all prewarm with caching-aware: try to make the prewarm generalized and resource friendly with privacy guaranteed.

One-for-all prewarm startup shares a similar mechanism with the **Template** method, which is hatched and already pre-imported most of the bins/libs after being informed by the socket. When a new invocation arrives and requires a new instance, it only needs to initialize or specialize from the templates. In the process above, Catalyzer [42] optimizes the restore process in C/R by accelerating the recovery on the retrenched critical path. Meanwhile, it proposes a sandbox fork to leverage a

template sandbox that already has pre-loaded the specific function for state resuing. To make the cold startup less initialization together with more flat startup latency, Mohan *et al.* [97] propose a self-evolving pause container pool by pre-allocating virtual network interfaces with lazy pause containers binding. As performance improves, so arises vulnerability. The security concerns in the one-for-all prewarm strategy are usually referred to as privacy concerns in Encapsule Layer. In other words, it is essential to make the private packages/libraries (REQs) inaccessible. For example, the famous open-sourced Apache OpenWhisk [103] resolves it by allowing that users can assign private packages in a *ZIP* or *virtualenv* to dynamically specialize the prewarmed container [104]. Zygote mechanism is a cache-based design used in Android for applications instantiation. SOCK [101] leverages the generalized Zygote mechanism that when creating prewarmed containers by considering the internal characteristics of workloads. Specifically, it designs a packages-aware caching model to dynamically adjust cached packages with the highest benefit-to-cost ratio. Because that it performs the metric allocating with runtime sampling in the system-level, malicious activities cannot reveal the sensitive packages/libraires for a specific function.

One-to-one and one-for-all prewarm: the challenging points.

For one-to-one prewarm startup and one-for-all prewarm startup, both can be beneficial for optimizing the cold startup in the Encapsule Layer of serverless architecture. Their respective flaws are also apparent. The one-to-one prewarm startup focuses on significantly less initialization latency by exchanging the memory resource. It meets the challenge that a warm-up time in point is usually hard to measure or predict while ensuring the reasonable allocation of memory resources, according to the research [118]. On the one hand, prediction-based and heuristic-based methods are particularly effective when historical data is sufficient to build an accurate model, but degrades when the trace is scarce. On the other hand, the prediction and iteration operations can introduce high CPU overhead when massive applications and function chains co-exist.

The template mechanism in the one-for-all prewarm startup is adopted to ease the high cost of functions cold startup from scratch. In addition, maintaining a global prewarm pool introduces less additional memory resource consumption than the one-to-one prewarm startup. However, it still suffers from several challenges, including the huge template image size [8, 51] and confliction of various pre-imported libraries. It may also potentially reveal the vicinity in which applications with a similar portrait are widely deployed. It is very important to "suit the remedy to the case" for cold startups in different scenarios. For example, it is much more efficient to generate a template by one-for-all prewarm startup when the function is invoked for the first time, or with poor predictions during the trace analysis. The one-to-one prewarm startup performs better for functions with general rules or diurnal patterns, and vice versa.

4   ORCHESTRATION LAYER

The main challenge in the System Orchestration Layer is the friendly and elastic support for different services. Even though the current serverless orchestrators are implemented differently, the challenges they encounter in the system are much the same. As hundreds of functions co-exist on a serverless node, it raises challenges for scheduling massive functions with inextricable dependencies. Similar to the traditional solutions [26, 35, 59, 76, 126], the serverless model also concerns the ability to predict the on-demand computing resources, and an efficient scheduling strategy for services. As shown in Figure 4, researchers usually propose to introduce the load balancer and resource monitor components into the controller, to resolve provision and scheduling challenges. The load balancer aims to coordinate the resource usage to avoid overloading any single resource. Meanwhile, the resource monitor keeps watching the resource utilization of each node, and passes the updated information to the load balancer. With the resource monitor and

(a) Provision adjustment at resource-level   (b) Load balance at instance-level

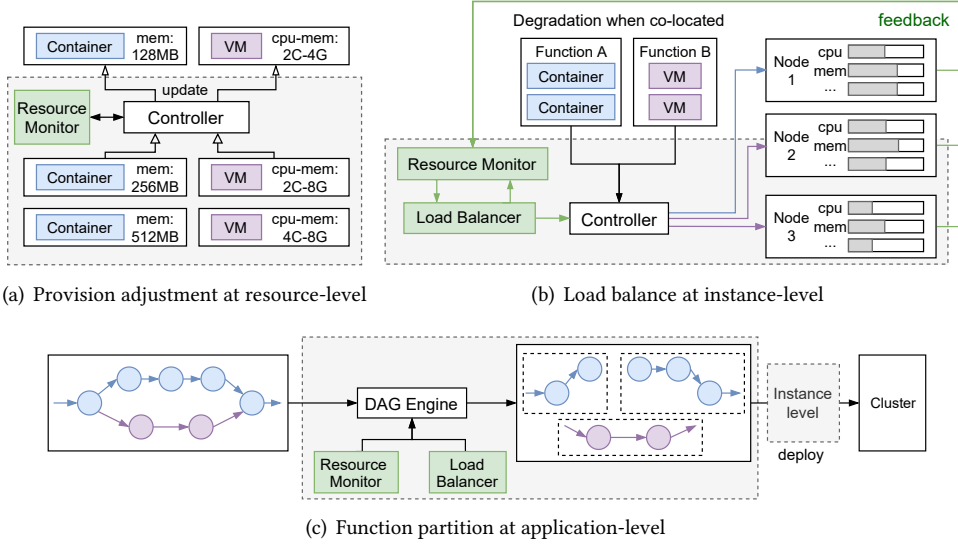(c) Function partition at application-level

Fig. 4. System logic and scheduling levels in Orchestration Layer.

load balancer, a serverless controller can perform better scheduling strategies in three aspects: **resource-level**, **instance-level**, and **application-level**. We summarize the hierarchy in Table 3.

Specifically, the "focused hierarchy" indicates an optimized method (aka resource adjusting) is designed besides an essential strategy for resource auto-provision, which can be classified into "R" (Resource-level), "I" (Instance-level), or "A" (Application-level), respectively. "Resource adjusting" shows whether the scheduling provides an adjustment for resource provision. "SLO" reflects whether SLO constraints are considered. "Intf" represents whether the resource contention or interference is discussed. "Usage feedback" reflects whether the feedback of resource usage in a physical node is considered. "Dynamic strategy" indicates whether it is a dynamic and runtime scheduling strategy. "Trace driven" indicates whether making choices depends on traces or collected data metrics. "Predict/Heuristic" reflects whether a prediction-based or heuristic-based method is used. "Implement" points out where it is implemented ("P" represents it is a prototype). Finally, "Insight" summarizes its unique insight and key motivation.

4.1 Dynamic Adjustment of Resource Provision (Resource-level)

Resources including CPU, Memory serve as the basic scheduling objects in serverless computing. For isolation and stability, resources are configured by an orchestrator, and access to them is restricted. The serverless controller will allocate resources for a new instance and isolate the execution environment when a cold startup occurs. Therefore, the key to building an efficient serverless controller is auto-scaling just the right amount of resources to satisfy the resilient workloads. However, the controller component itself cannot adjust appropriately because the resource is highly dynamic in the cluster and the potential inaccurate resource specifications by default.

Make resource provision of the container "just the right amount".

The common solution for avoiding resource over-provisioning is building feedbacks regarding historical traces. For example, works in [33, 34] optimize the original resource settings by varying the trace-driven patterns for VMs. In serverless computing systems with more fine-grained functions, a real-time resource monitor can be employed to help the controller make dynamic resource

Table 3. Works by focused hierarchy in System Orchestration layer.

Representative work	Focused hierarchy	Resource adjusting	SLO	Intf	Usage feedback	Dynamic strategy	Trace driven	Predict /Heuristic	Implement	Insight
Pigeon [82]	R	✓			✓	✓			kubernetes	Static pool
FlowCon [155]	R&(I)	✓			✓	✓	✓		P	DL tasks
SIREN [139]	R&(I)	✓				✓	✓	✓	AWS Lambda	ML tradeoff
CherryPick [5]	R	✓	✓		✓	✓	✓	✓	P	Bayesian Opt
Lin *et al.* [81]	R&(A)	✓	✓		✓	✓	✓	✓	AWS Lambda	Profiling
MPC [57]	R	✓	✓	✓	✓	✓	✓	✓	OpenWhisk	/
Chang *et al.* [32]	I&(R)	✓	✓		✓	✓	✓		kubernetes	/
Kaffes *et al.* [66]	I&(R)				✓	✓			P	/
FnSched [127]	I&(R)	✓	✓		✓	✓	✓		OpenWhisk	/
Guan *et al.* [50]	I&(A)	✓			✓	✓			P	Library
McDaniel *et al.* [93]	I		✓	✓	✓	✓			Docker Swarm	Two-tiered
Kim *et al.* [70]	I&(R)	✓	✓	✓	✓	✓	✓	✓	P	CPU cap
Smart spread [88]	I		✓	✓	✓		✓	✓	AWS Lambda	Profiling
Xanadu [40]	A&(R)	✓			✓	✓	✓	✓	OpenWhisk	Profile&predict
Step Functions [43]	A				✓				AWS Lambda	Health check
WUKONG [29, 30]	A				✓	✓			AWS Lambda	Graph to seq
Viil *et al.* [136]	A&(R)	✓				✓		✓	Pegasus	Partition
SAND [4]	A	✓		✓					P	Colocation
GlobalFlow [154]	A&(I)						✓		AWS Lambda	Cross-regions
SONIC [129]	A&(R)	✓			✓	✓	✓	✓	AWS Lambda	Hybrid exchange

adjustments, as shown in Figure 4(a). For example, Pigeon [82] builds a serverless framework, introducing a function-level resource scheduler and an oversubscribed static pool. The scheduler assigns containers with different resource configurations to the queries based on the node capacity and function requirement. However, their container pool is based on a static configuration, which may lead to resource segmentation and low utilization. FlowCon [155] facilitates dynamic resource allocation for container-based DL training tasks in the near future and resets resource configuration elastically. Though they design a dynamic auto-provision strategy based on the monitor feedback, the SLO constraints and resource interference are not considered further. It results in their lack of flexibility and practicality in a real production environment. DRL (Deep Reinforcement Learning), evolving from Deep Q-learning, is a widely used combination algorithm by learning control strategies from higher-dimensional perceptual inputs [96], which can be used to make resource provision decisions. For example, Wang *et al.* [139] propose a serverless scheduler based on DRL for ML training jobs. It can dynamically adjust the number of function instances needed and their memory size, to balance high model quality and the training cost.

The keys to making resource provision robust to performance.

Recent works take the SLA into account to ensure stability when functions are invoked in a shared-resource cloud. CherryPick [5] leverages the Bayesian optimization, which estimates a confidence interval of an application's running time and cost, to help search the optimal resource configurations. Unlike static searching solutions, it builds a performance model to distinguish the unnecessary iteration trials, thus accelerating the convergence. However, CherryPick's performance model targets big data applications specifically, not generalized to other applications. Similarly, Lin *et al.* [81] build an analytical model to help general serverless applications deployment. It can predict the application's end-to-end response time and the average cost under a given configuration. They also propose a Probability Refined Critical Path Greedy algorithm (PRCP) based on the transition probability, recursively searching the critical path of execution order. With PRCP, they

can achieve the best performance with a specific configuration under budget constraints or less cost under QoS constraints. Besides SLA, shared-resource contention should also be noticed in the multi-tenant environment. HoseinyFarahabady *et al.* [57] discuss this topic. Their proposed MPC optimizes the serverless controller for resource predictively allocation. By introducing a set of cost functions, it reduces the QoS violation, stabilizes the CPU utilization, and avoids serious resource contention. However, these resource and workload estimations based on ML (Machine Learning) or AI (Artificial intelligence) usually achieve a trade-off between an optimal global solution and robust performance to inaccurate workload information [26, 31, 60, 133]. Whether they can avoid fragile robustness and improve resource utilization in the production environment is unknown and remains a critical avenue to explore.

4.2 Load Balancing for Instance Scheduling (Instance-level)

In addition to dynamic adjustment at the resource level, the most important part of a serverless system is instance-level scheduling. From the perspective of cloud vendors, they hope to achieve either higher throughput, higher resource utilization, or less energy consumption. At the same time, users prefer cheaper deployment costs and less end-to-end invocation latency. To this end, instances from multi-functions or multi-tenants should be carefully scheduled across the cluster to achieve the above targets.

The mainstream solution is leveraging load balancer, which is also shown in Figure 4(b). It is designed as the queries router to help schedule functions and achieve the load balancing between nodes in the cluster. The strategies can be classified into two categories: **Hash-based** and **Multi objective-based** methods. In the hash-based method, the controller uses a hash function to decide a home node (or executor) of a given function for default routing. Then it will set a stepsize to recursively filter out an alternative if the home is not available or under resource-constrained. They are usually done by a health check in each physical node. Until the cloud provider has a full understanding of the characteristics of workloads running in its serverless system and the cluster, we recommend using the hash-based method to implement a load balancer. In the multi-objective-based balancing method, the load balancer aims at multiple optimizations, for example, throughput, response time, resource utilization, etc. Therefore, it should balance different factors to satisfy both the cloud vendors and users.

Leverage resource monitor and load balancer to make scheduling decisions.

Resource monitor can provide a global view of resource status in a cluster, which helps the load balancer make better scheduling decisions. Chang *et al.* [32] design a comprehensive monitoring mechanism for the Kubernetes-based system. It can provide a variety of runtime information to the scheduler, including system resource utilization and the QoS performance of an application. The flaw of the study is that it does not provide a complicated resource scheduling algorithm. Kaffes *et al.* [66] propose a centralized and core-granular scheduler. Centralization provides a global view of the cluster to the scheduler so that it can eliminate heavy-weight function migrations. Core-granularity binds cores with functions and therefore avoids core-sharing among functions and promises performance stability. However, they only consider the scheduling of CPU resource, but ignore other important resources like memory. FnSched [127] regulates CPU-shares to accommodate the incoming application invocations by checking the available resource. A key advantage of employing a greedy algorithm is that fewer invoker instances are scheduled by concentrating invocations in response to varying workloads. Though FnSched makes a tradeoff between scalable efficiency and acceptable response latencies, it is limited by the assumption that function execution times are not variable. Guan *et al.* [50] propose an AODC-based (Application Oriented Docker Container) resource allocation algorithm by considering both the available resources and the required libraries. They model the container placement and task assignment as an optimization problem, and then

take a Linear Programming Solver to find the feasible solution. The Pallet Container performs the AODC algorithm, serving as both a load balancer and resource monitor. The downside is that plenty of containers will occupy the memory space as the number of functions increases.

Take the performance interference and QoS constraints into consideration.

While improving utilization, load balancing strategies also bring the interference challenge that sharing resources between instances may result in performance degradation and QoS violation. Different functions' sensitivities to different resources may vary, which means that we should avoid physical co-location of functions that are sensitive to the same resource (e.g., CPU-sensitive containers may cause serious CPU contention when co-located). The load balancer should notice and moderate the interference when scheduling containers. McDaniel *et al.* [93] manage the I/O of containers at both the cluster and the node levels to reduce resource contention and eliminate performance degradation effectively. Based on a resource monitor in Docker Swarm, it refines the container I/O management by providing a client-side API, thus enforcing proportional shares among containers for I/O contention. Kim *et al.* [70] present a fine-grained CPU cap control solution by automatically and distributedly adjusting the allocation of CPU capacity. Based on performance metrics, applications are grouped and allowed to make adjustment decisions, and application processes of the group consume only up to the quota of CPU time. Hence, it minimizes the response time skewness and improves the robustness of the controller to performance degradation. Smart spread [88] proposes an ML-based function placement strategy by considering several resource utilization statistics. It can predictively find the best performing instance as well as incurs the least degradation in performance of the instance.

4.3   Data-driven Workflows for Application Deployment (Application-level)

At the application level, load balancing strategies can be categorized into two kinds: the spread strategy, which distributes functions of an application across all the physical nodes, and the bin-pack strategy, which tries to schedule functions of an application to the same node first [57]. Intuitively, the spread strategy seems to better balance the workloads on the nodes while avoiding serious resource contention. However, it weakens the data locality, which means that the spread strategy will introduce more transmission overhead than bin-pack if functions are data-dependent. So it is why we need to optimize the scheduling from the perspective of the application.

Invocation patterns and workflow execution models.

As shown in Figure 5(a), if a function is invoked from user queries via the RESTful API or other triggers, it is called external invocation. The instance-level load balancing can perform well in external invocation scenarios. However, the emerging cloud applications may consist of several functions, and there are data dependencies between multiple functions. For example, the implementation of a real-world social network consists of around 170 functions [2]. In this case, functions in such an application will get active by various triggers which may come from the user query or another function. If a function is initialized or assigned by other functions, it follows the internal invocation pattern. Currently, researchers raise their vision to the data-driven scheduling for internal invocations from the perspective of application-level topology.

Workflow is the most common implementation of internal invocations, where functions are executed in a specified order to satisfy complex business logic. The execution models of these data-driven workflows can be extracted into two approaches: sequence-based workflow and DAG-based workflow. As shown in Figure 5(b), functions are invoked in a pipeline through a registered dataflow in the sequence-based workflow. The sequence-based workflow is the basic and the most common pattern in the serverless workflow, and most cloud vendors provide such execution mode for application definition. Obviously, there is more than one sequenced workflow in one complex application, and the same functions can be executed in various sequences. If we regard

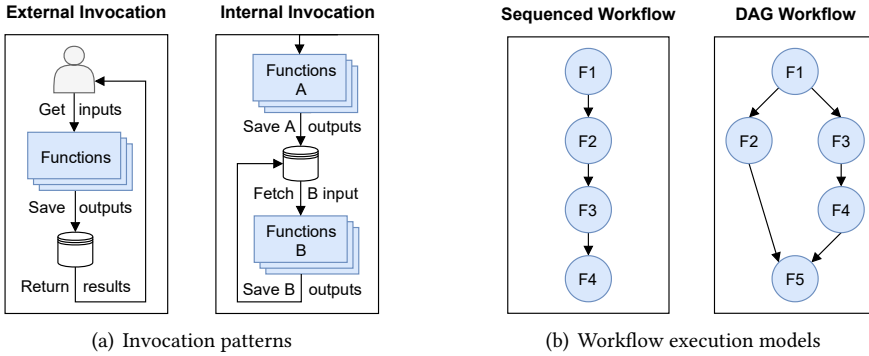(a) Invocation patterns (b) Workflow execution models

Fig. 5. Two Invocation patterns for funtions and two execution models of workflows.

each function as a node and dataflow between nodes as a vector edge, such an application with multiple interlaced sequenced workflows can be defined by the DAG (hence the name "DAG-based workflow"). Nowadays, few Cloud vendors provide services for the application definition in the DAG (Direct Acyclic Graphs) form, aka serverless workflows [1, 21, 27].

The scheduling overhead introduced in serverless sequences.

With massive functions communicate with each other, scheduling of dataflow dependencies introduces more complexities. However, the existing serverless systems in the production environment commonly treat these workflows as simple recursion of internal invocations. It raises the challenge of reducing the overhead in the System Orchestration layer by scheduling function sequences [16]. Current policy to manage the function sequences is quite simple- functions are triggered following the first-come-first-served algorithm [129]. However, as the length of the function sequence increases, cascading cold start overheads should be addressed to avoid seriously end-to-end latencies degradation of sequenced workflows [20, 40]. To this end, Xanadu [40] combines the prewarm strategy with a most-likely-path (MLP) estimation in the workflow execution. It prewarms instances by a speculative-based strategy and makes just-in-time resource provisioning. However, the prediction miss would introduce additional memory waste, especially in the scenario of multi-branch or DAGs. Moreover, serverless workflow engines prefer the Master-Worker architecture where ready functions are identified by the state and invoked directly by the master without a queue [9, 17, 30, 47, 89], including AWS Step Functions [43] and Fission Workflows [46]. As shown in Figure 5(a), the deficiency is that the additional overhead is introduced in the function workflow through unnecessary middlewares (e.g., unnecessary storage in an internal invocation).

Enhance the data locality for efficient serverless DAG executions.

To help function workflow avoid undesired middlewares, researchers usually co-locate the functions into subgraphs to enhance the data locality, as shown in Figure 4(c). For example, Viil *et al.* [136] use multilevel k-way graph partitioning to provision and configure scientific workflows automatically into multi-cloud environments. However, their partition algorithm may not match well with serverless applications, where each node in the graph can auto-scale multiple replicas in such as, foreach steps. In this case, the connections and edge weights become unpredictable. In serverless context, WUKONG [29, 30] implements a decentralized DAG engine based on AWS Lambda, which combines static and dynamic scheduling. It divides the workflow of an application into subgraphs, before and during execution, thus improving parallelism and data locality simultaneously. However, WUKONG's colocation of multi-functions within a Lambda executor may introduce additional security vulnerabilities due to its weakened isolation. SAND [4] presents a

new idea to group these workflow functions into the same instance so that libraries can be shared across forked to reduce initialization cost, and additional transmission can be eliminated in the workflow due to the data locality. SAND performs a better isolation mechanism than WUKONG by using process forking for function invocations, however it ignores the colocation interference resulting from the resource contention. When exchanging intermediate data of DAGs, SONIC [87] proposes to use the VM-storage-based transmission strategy when functions are co-located on the same node. The optimal transferring depends on application-specific parameters, such as the input size and node parallelism. By predicting such runtime metrics of functions in the workflow, it dynamically perform the data passing selection with a communication-aware function placement. GlobalFlow [154] considers a geographically distributed scenario where functions reside in one region and data in another region. It groups the functions in the same region into subgraphs and connects them with lightweight functions, so that it improves data locality and reduces transmission latency. As the authors stated, the combination of local and cross-region strategies in a holistic manner can be further explored.

Summary of the challenges in the scheduling of serverless workflows.

Workflow scheduling is an NP-hard problem, and researchers have been designing various strategies for it [1, 91]. Such optimization in the workflow aims to minimize the makespan, reduce the execution cost, and improve resource utilization while satisfying single or multiple constraints. To resolve the above challenges, leveraging enhanced data locality is a focus in serverless computing. The challenge is that the end-to-end latency of a workflow query could increase significantly due to frequent interactions with the storage from different nodes. Resource volatility becomes another focus in the serverless system, which can be unpredictable as the number of functions increases in the production environment. It introduces more difficulty to find an efficient workflow placement and scheduling strategy in a concise decision time (e.g., 10ms for load balancing). In order to evaluate the efficiency and performance for future workflow-based research, DAG-based or DG-based serverless benchmarks also urgently need to be published. They are better adapted based on real applications rather than simple micro-benchmarks [110] or function self-loops [81, 148]. Keeping a guaranteed QoS performance is also significant for applications in serverless computing, whereas it has not been widely investigated.

4.4 Security Concerns in Orchestration Layer

In the Orchestration Layer, the most serious security concern is how to resolve unavailability. It usually refers more to performance security than functional security. The attackers may establish destructive behaviors from either resource-level, instance-level, or application-level, resulting in unmatched in-memory footprint, concurrency exhaustion, or workflow exceptions. The efficient solution to these concerns is leveraging BaaS components to restrict access.

In-memory footprint by unrestricted read-in.

Contrary to the intuitive believing, serverless architecture can make the programming more complicated because the decoupled microservices have higher requirements on the normalized input data. The unrestricted memory read-in from the input data may result in the timeout or breakdown for its oversized memory footprint (e.g., 300MB memory read-in within a 256MB-limit container). Function developers may overlook this vulnerability in a public cloud where an attack can easily disguise as a legitimate invocation. On the premise that the user code is fragile against such kind of attack, a serverless system needs to bind filtering rules in event trigger to help avoid this security concern.

Concurrency exhaustion by DDoS (Distributed Denial of Service) attack.

When an application is decoupled into a mass of functions, its concurrency bottleneck depends on the maximum throughput of all function nodes. In this case, the DDoS attack on serverless is a more
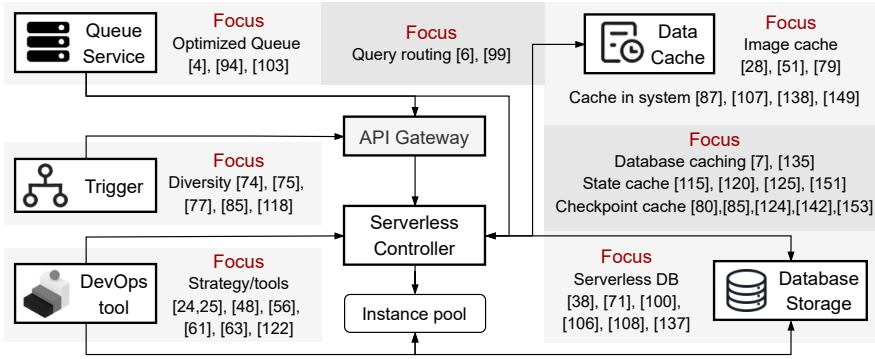
Fig. 6. Techniques and works about BaaS components in System Coordination layer.

significant threat. Any unavailable function node can seriously take down the entire application with a wilder attack surface. It can cause seriously degraded QoS by invocation exhaustion attacks in a single function node, or generate a large bill for the application account. The latter is known as the DoW (Denial of Wallet) attack in serverless computing. This security risk can be mitigated by setting an upper limit on invocations concurrency and instances quota on function creating.

Workflow exception by malicious dataflow.

When triggering invocations, function input parameters and preferences are also passed within the dataflow. Therefore, an internal invocation is more fragile than an external one if the function maintainer fails to provide essential verification of queries source. In addition, attackers can inject malicious data into the queries to generate the invocation exceptions, or make the workflow execution order out of control. In the case of serverless architecture, all function nodes in the application need to authorize their access permission to identify whether the input data tamper with the current invocation.

5   ESSENTIAL BAAS COMPONENTS IN COORDINATION LAYER

We have examined the most critical implementations in Section 2, 3, and 4, and there are also some other components in the System Coordination layer introduced to support or enhance the serverless system. We also outline the relevant techniques and research in Figure 6. In terms of implementation, a serverless system needs to integrate the six significant components or services: Storage, Queue, API gateway, Trigger, Data cache, and DevOps tools. Most of the literature focuses on Data cache, Queue service, and function storage from an academic perspective. In contrast, the cloud vendors mainly focus on productions about Trigger service and DevOps tools. We will discuss each of these components detailly in the following text.

5.1   Storage Service

One of the key requirements of a serverless workload is efficiently sharing ephemeral data between functions or saving results for asynchronous invocation. Therefore, a natural way to communicate between them is to exchange the data through a remote store.

Different phases of storage during the function execution.

During a serverless invocation, there are three phases where the database service is required: Authentication, In-Function, and Log. Authentication is usually performed ahead of controller scheduling to avoid security issues, and it should get a fast response for access. Using an MMDB (Main Memory DataBase) to implement the Authentication phase is recommended in a serverless

system, such as Redis, a high-performance key-value database. The calls of storage APIs during the function execution make up the in-Function phase. Users can choose to use either a DRDB (Disk-Resident DataBase, e.g., MySQL) or an MMDB by different BaaS interfaces for ephemeral storage. The Log phase builds the bridge for users to return invocations results, especially for the functions invoked in an asynchronous manner. A detailed record in JSON format, including runtime, execution time, queue time, states, will be ephemerally or permanently stored and returned (e.g., CouchDB in OpenWhisk). It is recommended to be designed as serverless storage, following the invocation patterns to only pay for queries consumed by the storage operation and the storage space consumed when logging. However, the throughput of existing storage is a major bottleneck due to the frequent and vast functions interactions [64, 65]. Although current serverless systems support NAS (Network Attached Storage) to help reduce storage API calls, these shared access protocols are still network-based data communication essentially.

IO bottleneck in storage: modeling in serverless context.

Traditional solutions use predictive methods [38, 100, 137] and active storage [109, 131, 132, 143, 145, 152] to automatically scale resources and optimize the data locality on demand. For serverless storage, researchers explore using a hybrid method to ease the I/O bottleneck. For example, Pocket [71] is strict with the separation of responsibilities across the control, metadata, and data planes. By using heuristics and combining several storage technologies, it dynamically rightsizes resources and achieves a balance between cost and performance efficiency. To alleviate the extremely inefficient execution for data analytics workloads in the serverless platform, Locus [106] models a mixture of cheap but slow storage with expensive but fast storage. It makes a cost-performance trade-off to choose the most appropriate configuration variable and shuffle implementation. Middleware Zion [108] enables a data-driven serverless computing model for stateless functions. It optimizes the elasticity and resource contention by injecting computations into data pipelines and running on dataflows in a scalable manner.

Due to the data-shipping architecture of serverless applications, current works usually focus on designing a more elastic serverless storage, enhancing the data locality to ease the I/O contention of function communication on the DB-side. However, due to the potential heterogeneity of different functions, the uncertainty above still makes these technologies in practice challenging.

5.2 Specialized Queue

In various implementations of the serverless system, the Queue is acquiescently integrated into the System Orchestration layer, which passes messages between different system components. For instance, Apache Kafka, serves as a distributed message streaming platform that allows applications to write and subscribe to messages across different hosts.

Interact with the controller by node queue and function queue.

The function queue can send messages between the controller and functions. In contrast, the node queue serves for load balancing to schedule the functions to different nodes (e.g., a queue in the cluster manager). A representative of adopting function queue design is OpenWhisk [103]. When the OpenWhisk controller receives an invocation query, it decides which invoker should execute the instance and then sends it to the selected invoker via Kafka. To increase parallelism, it also leverages the topic partitioned so that messages with the same consumer can be written to the same partition. SAND [4] also follows this message queue design by introducing a two-level hierarchical message bus: a local message bus deployed on each host and a global message bus distributed across different hosts. A local message bus is partitioned into different message queues such that every function on the host subscribes to messages from this function queue. In this way, if a function and its successors are running on the same host, it can directly write its output into the local message queue subscribed by its successor. Otherwise, the output is written to the global

message bus. Other work dives into the shortcomings of the queue-based mechanism, which may lead to reduced performance and availability in the serverless context. McGrath *et al.* [94] propose to introduce the "cold queue" and the "warm queue" to assume different responsibility for function queries. DORADO [99] also uses shared memory to mediate communication and persist data. By such means, queries can be routed to any container that is replicated.

It is more convenient for developers to adopt scalable queues in serverless computing, as the scaling is delegated to cloud vendors. For example, Amazon Simple Queue Service provides scalability by processing each buffered invocation independently, scaling transparently to handle bursty loads without provisioning instructions. The idea of the scalable queue also meets the requirements for serverless computing, such as pay-per-use, dependability, convenience, and flexibility.

5.3 API Gateway and Various Triggers

In serverless computing, instances are created on-demand, and invocations are not bound with a static address. When deploying the containers or VMs to the cluster, the system will dynamically assign addresses to services. In this case, containers on the same node in the default network can communicate by IP addresses, while containers across nodes need to allocate ports for forwarding. The dynamic port allocation raises the challenge to manage as it intensifies at scale. The API Gateway component can provide a unique entry point to ensure accurate services' addresses. When the queries join the API Gateway, the service registry is inquired, and the queries are forwarded to the available service instances according to the IP route. It should also consider the availability and reliability of the serverless system, for example, the lazy reaction for services incompatible due to the hardware heterogeneity.

Meanwhile, serverless systems design various triggers to invoke functions in response to queries. A trigger defines how a function is invoked, and the binding rule represents a mapping between them. The trigger and binding rule together make up a probe for the detectable event and help avoid hardcoding access to other services [77]. Besides invoking an event-driven function, triggers can also provide a declarative way to connect data to the code (e.g., storage services).

There are four most popular triggers: HTTP, Queue, Timer, and Event. HTTP trigger is widely used [118] to handle external invocations, by which a function can be easily invoked once an HTTP query arrives. While HTTP trigger simplifies the external invocation for functions, it shows less efficiency in the case of internal invocations. An alternative way to handle internal events is using Queue trigger, by which functions get triggered whenever an invocation enqueues. For instance, Kubeless [74] provides a Kafka-based queue trigger bond with a Kafka topic so that users can invoke the function by writing messages into the topic. Specific purposes also require more extensive triggers. For example, a timer trigger in Kubernetes can invoke a function periodically. It creates a CronJob [75] object, which is written in a Cron expression representing the set of invocation time, to schedule a job accordingly. An Event trigger invokes a function in response to an event, which is the atomic piece of information that describes something that happened in the system. A convincing example of such implementation is Triggerflow [85], which maps a workflow by setting an event trigger in each edge.

5.4 Data Cache

To ensure graceful performance in case that the workload bursts and reaches a hard limit in concurrency, the common practice among cloud-based applications are utilizing multiple levels of caches [12]. Data caching can cut out unnecessary roundtrips for less response time when queries experience a full-end invocation path. One common idea is caching at API Gateway (e.g., caching solution for GET method [6]), or caching the pages and only result in a storage I/O query (e.g., Amazon DAX [7] and Aurora [135] for database caching). Another common solution is to cache in

the system [87, 107, 138, 149]. It freezes maintainers from declaring inside the function by enabling the caching of static assets or large objects. However, the cached data is only available in the ephemeral container, which makes sharing across all short-term instances of a function challenging. And this approach may not be as effective as it seems- the first invocation in every container will result in cache misses.

Image cache: on-demand loading and page sharing.

The simplest and popular method is to provide the image cache for accelerating. In a container-based serverless system, an image is composed of multiple layers and shared by numerous containers as needed. When functions are invoked on a host for the first time, images need to be downloaded and cached locally. For example, Slacker [51] builds a Docker storage driver and uses block-level COW to implement snapshots in a VMstore. Docker images are represented by VMstore's read-only snapshots, and the pull and push operations only involve sharing snapshot IDs rather than large network transfers. It makes it possible for docker workers to fetch data lazily from shared storage as needed. DADI [79] also implements an image service merging a sequence of block-based layers, and it caches recently used data blocks adopting the overlay with a tree-structured design. SEUSS [28] factoring out common execution state shared in a snapshot stack, which expresses a lineage between snapshots. It uses CoW to capture into a snapshot only the pages that were modified for fast deployment. Furthermore, SEUSS proposes to use anticipatory optimization to reduce the number of written pages captured in each snapshot.

State cache: make the serverless applications stateful.

State cache can be further combined with an active database to enable function execution stateful. Bledi [151] extended from Olive [115] adopts the refined SSF (Stateful Serverless Function) instance with a built-in database. By saving a set of intent tables recording the SSF's state and function information, it provides a fault-tolerant workflow execution manner. SNF [120] decouples the functions into computing units and state units, and relaxes the constraint of communication between cooperating units. When processing a subsequent flowlet in the same flow, the function's internal state is cached in the local memory. Then SNF can proactively replicate ephemeral state among compute units. Cloudburst [125] packs the local cache with a function executor in each instance, and periodically publishes a snapshot of cached keys to the key-value store. By such means, Cloudburst can enhance the data locality via physical colocation of mutable caches and enable state management with the remote auto-scaling key-value database.

Checkpoint cache: enable functions with fault tolerance.

The demand for fault tolerance also inspires researchers to make relevant techniques applicable in serverless context, such as C/R-based [80, 153] and log-based [85, 142]. One example of such implementation in serverless computing is AFT [124], which builds an interposition between a storage engine and a common serverless platform by providing atomic fault tolerance shim. It leverages the data cache and the shared storage to guarantee the isolation of atomic read, avoid storage lookups for frequent access, and prevent significant consistency anomalies.

In addition to the implementations we discussed above, other caching mechanisms can be explored and integrated into any layer of our proposed serverless architecture. Of course, they are all recommended to follow the pay-as-go mode based on the used resource. In summary, data caching is still an essential and important component for higher flexibility and better performance.

5.5 DevOps Tools

DevOps is a compound of development and operations, which improves collaboration and productivity. Since the responsibility of managing underlying resources and runtime environment are transferred to the cloud vendors in the serverless concept, developers only need to focus on the code logic. Operations teams are actually liberated from this process, where they are required to

check, compile, pack images, and test deployment after developers submitting the code. There is a necessity for providing DevOps tools in the serverless system. We agree with the pipeline implementation [61] to group the DevOps into CI (Continuous Integration), CD (Continuous Delivery), and CM (Continuous Monitoring) categories.

CI refers to the continuous merging of developed code with others during software development while ensuring automatical validation and building. Not only should it make the operations within the function (e.g., Jenkins [63] for integration test, Honeycomb [56] and SonarQube [122] for inspection test), but also focus on availability between functions in an application (e.g., monitoring and debugging tool IOpipe for workflows [24]).

CD requires that the serverless system can automatically update the application instances with the old version while keeping the services still available. Nowadays the common solutions are **Rolling-update**, **RED-Black** (aka Blue-Green), and **Canary** deployment, which are adopted by Kubernetes [25, 48]. Essentially, these deployment strategies share the same mechanism to keep part of instances with old version serving and then gradually replace them with new ones. The difference between them is the complexity of rollback to the last available instances.

CM enables the DevOps team to receive feedback on the problems and errors during the above steps in time. It also provides a visualized interface for monitoring applications' runtime behavior and resource utilization. Actually, the feedback tool is already implemented in most CI/CD tools and runs through the whole DevOps lifecycle of the application. By enriching the CM component, visualizing activities, and resource monitoring [69], users can better understand their services running in the cloud.

The concept of DevOps in serverless typically appears more frequently in production scenarios, and most platforms provide various such tools to ensure good compatibility and flexibility. However, the introduction of DevOps may bring about new vulnerabilities threatening the security of instances, and serverless research should also provide more substantial support for detecting vulnerable containers [22, 69, 73, 128].

6 PERFORMANCE AND COMPARISON

This section first summarizes the performance with different VMs/containers, language runtime, and resource limits in serverless computing. Then, we analyze the current production serverless systems to show the preferences.

6.1 Performance Analysis

The runtime within the instance built from different virtualization technologies can exhibit different cold startup performances. Besides, the language runtime is another factor that can seriously affect the cold startup latency. For example, evaluations in Catalyzer [42] show the cold startup latencies with different VM/container and language runtimes. As shown in Figure 7, HyperContainer introduces the highest cold startup latency with various language runtimes. Process-based Docker runtime certainly performs significantly better than others. Generally speaking, the interpreted languages (e.g., Python) incur a higher initial cost and make startup times up to 10× slower [4, 101] than the compiled languages (e.g., C) when cold startups.

However, according to the performance tests from Jackson [62], which measure the startup and execution latency of different language runtimes, the performance of compiled and interpreted language runtime also depends on the platform. For example, the cold startup latency of .NET C# on AWS Lambda is higher than that of Node.js, while it is the opposite on Azure Functions. It is because that Azure Functions would provide better support for C# based on its core technology .NET for Microsoft, and implements by running on windows containers rather than the open-source .NET CLR (Common Language Runtime) based on Linux containers.
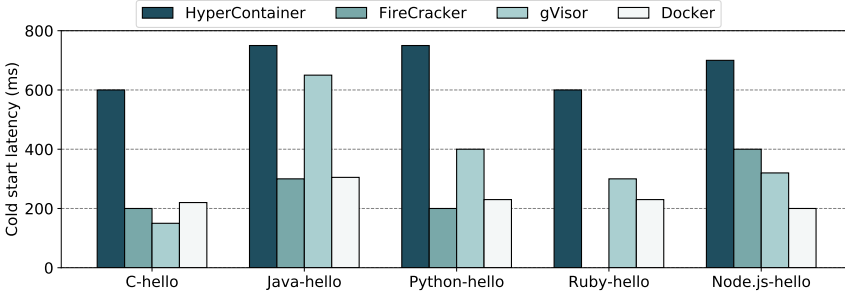
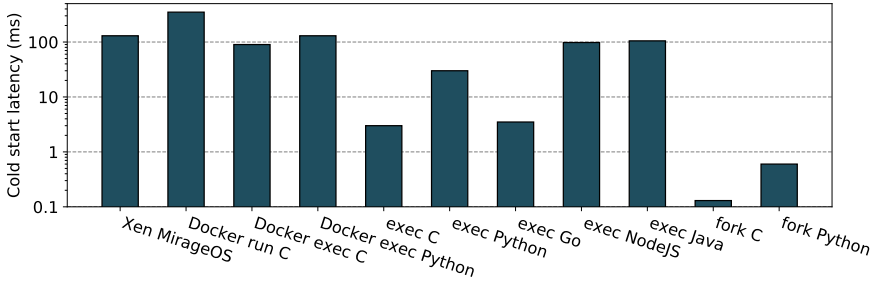Fig. 7. Cold startup latency under different runtimes [42].

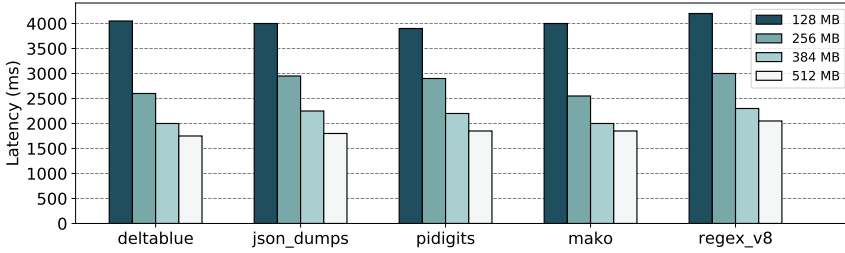Fig. 8. Cold startup latency under different isolation mechanisms [4].

Fig. 9. Cold startup latency with different container memory limits [117].

Besides the cold startup analysis of different language runtimes, SAND [4] also measures several sandbox isolation mechanisms for function executions, and we show their results in Figure 8. Native executions (exec and fork) are the fastest methods, while Unikernel (Xen MirageOS) performs similar to using a Docker container. Regardless of the recycled user code in memory in the paused container, using the Docker client interface to start a warm function (Docker exec C) is much faster than a cold startup (Docker run C).

Another significant factor that slows down the cold startups for the container-based serverless system is the memory limit. The performance evaluation about memory allocation [117] is shown in Figure 9. The cold startup latency of each microbenchmark function increases as stepping to smaller memory limits. We can also see that there is a significant decrease in container startup latency when stepping from 128MB to 256MB. And larger memory limit results in less obvious optimizations without the reasonable regime of marginal increases. It also explains why most serverless systems set 256MB as the default memory limit of the function container.

Table 4. Comparing metrics of four serverless vendors [77, 94] ("CCI" means the concurrent invocations).

Item	Amazon Lambda	Google Functions	Microsoft Azure Functions	IBM OpenWhisk
GFLOPS per function	19.63	4.35	2.15	3.19
TFLOPS in 3000	66.30	13.04	7.94	12.30
Throughput of 1-5 CCI	20-55TPS	1-25TPS	60-150TPS	1TPS
Throughput of 2000 CCI	400TPS	40TPS	120TPS	210TPS
CCI Tail latency	best	superior	worst	inferior
CI/CD performance	best	fail frequently	long latency	balanced
Read/Write (1-100 CCI)	153/83 MB/s - 93/39.5 MB/s	56/9.5 MB/s - 54/3.5 MB/s	424/44 MB/s - NA	68/8 MB/s- 34/0.5 MB/s
File I/O (1-100 CCI)	2-3.5 second	10-30	3.5-NA	15-60
Object I/O (1-100 CCI)	1.3-2.4 second	5-8	12-NA	1-30
Trigger Throughput	55-25-860 (HTTP-Object-DB)	20-25-NA	145-250-NA	50-NA-40
Language Runtime overhead	balanced 0.05s avg	(-0.06) 0.22s (+0.1)	(-0.02) 0.22s (+0.03)	(-0.02) 0.17s (+0.02)
Dependencies overhead	(-0.5) 1.1s (+0.2) avg	(-0.5) 1.9s (+0.4)	(-1.3) 3.4s (NA)	NA
Maximum Memory	3008MB	2048MB	1536MB	512MB
Execution Timeout	5 minutes	9 minutes	10 minutes	5 minutes
Price per Memory	$0.0000166/GB-s	$0.0000165/GB-s	$0.0000016/GB-s	$0.000017/GB-s
Price per Execution	$0.2 per 1M	$0.4 per 1M	$0.2 per 1M	NA
Free Tier	First 1 M Exec	First 2 M Exec	First 1 M Exec	Free Exec/40,000GB-s
Idle instance lifetime	5-7 min	15 min	Mostly 20-30 minutes	default 15 min

As the supplement of the above factors that affect serverless cold startup performance, Shahrad *et al.* [117] explore other factors that may affect the function cold startup and execution time, such as MKPI (mispredictions per kilo-instruction), LLC (Last-level Cache) size, and memory bandwidth. Firstly, they find that a longer execution time usually appears with noticeably lower branch MKPI within a function. It is easy for us to understand that functions with short execution time spend most of the time on language runtime startup, and thus the branch predictor outputs more miss when stay trained. Secondly, the LLC size is not a significant factor that affects the cold startup latency and execution time. Higher LLC size cannot bring in better performance for serverless function execution because of the insensitivity. Only when the LLC size is very small (e.g., less than 2M) will it become a bottleneck for the function execution and cold startup. Cloud vendors usually set a default LLC size and pre-profile in the serverless system to avoid serious performance degradation. BabelFish [121] also finds that lazy page table management can result in heavy TLB stress in a containerized environment. To avoid redundant kernel jobs produced in the process of page table management, they try to share translations across containers in the TLB and page tables.

6.2 Production Comparison

With more attempts to enable the rapid development of cloud-native applications, Wang *et al.* [141] evaluate the performance of three commercial serverless platforms by invoking measurement functions with stepwise memory limits to collect various system-level metrics. Lee *et al.* [77] also gives a detailed comparison between Amazon Lambda, Google Functions, Microsoft Azure Functions, and IBM OpenWhisk. They demonstrate the differences in terms of throughput, network bandwidth, I/O capacity, and computing performance. Based on these experiments, we summarize the metrics in Table 4.

From this table, we can get a glimpse of their respective strengths and weaknesses. For example, AWS Lambda shows higher capacity and throughput of concurrent function invocations, however, performing poorly in trigger throughput. Microsoft Azure Functions enable fast read and write speed when queries are invoked in sequence, and show relatively higher function cold startup latency. Undoubtedly, all cloud vendors are aware of the challenges in serverless architecture and are actively optimizing the function invocation performance and relevant BaaS bottlenecks.
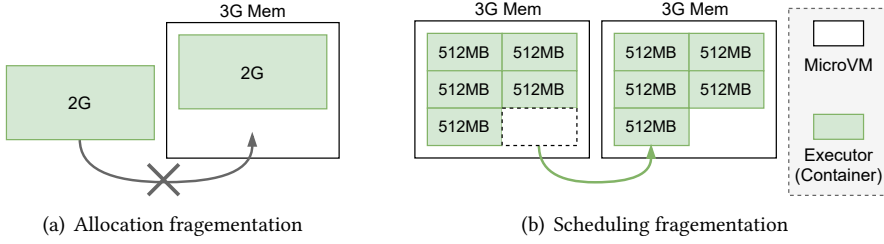
(a) Allocation fragmentation   (b) Scheduling fragmentation

Fig. 10. Two scenarios where the memory fragmentation arises.

7 OTHER KEY LIMITATIONS AND CHALLENGES

The limitations of the current works in each layer and challenges are already discussed in the corresponding sections. This section will highlight other key limitations and challenges in the Encapsule, Orchestration, and Coordination layer, respectively, as an orthogonal supplement. We refer readers for more detailed and focused discussions on the Virtualization Layer in other survey [13, 18].

7.1 Stateless within Encapsule Layer

An essential feature of serverless is that the service is loaded and executed on-demand rather than deployed in a long-term running instance. To prevent a large number of instances from occupying memory resources, the serverless controller sets an instance lifetime to recycle them automatically. Because short life-span functions within the application are no longer associated with a particular instance or server, each query processed cannot be guaranteed to be invoked by the same function instance. In other words, the application's state cannot and will not be kept on the resumed instance [68]. The stateless nature weakens the generality of the serverless architecture, limiting its scope to stateless applications, such as Web applications, IoT (Internet of Things), media processing, etc. Undoubtedly, the extension of stateful serverless architecture (see [120, 125, 151] in Section 5.4) by saving state in object storage or key-value stores fails to provide low latency and high throughput simultaneously, makes it inferior to regular sticky sessions as IaaS or PaaS does.

7.2 Memory Fragmentation within Orchestration Layer

In the serverless architecture where multiple tenants co-exist, concurrent invocations are either processed in multiple containers and experience undesired cold startups in each one, or executed concurrently in one single container (e.g., OpenFaaS and OpenLambda). In the former, a container is allowed to execute only one invocation at a time for performance isolation. In this case, the memory footprint of massive sidecars prevents serverless containers from achieving high-density deployment and improved resource utilization [3]. The key to this challenge is slimming and condensing the container runtime by deduplication within the VMM and guest kernel, such as sharing the page cache across different instances on the host. In the latter, memory fragmentation becomes a top priority. The figure 10 depicts two common scenarios where memory fragmentation may arise. Allocation fragmentation is usually due to the improper provision of a microVM. Function executors can not fully utilize the memory allocated. Scheduling fragmentation is inevitable and usually caused by instance-level load balancing strategy when auto-scaling with workload changes. Since the serverless emergence, challenges remain in further achieving an efficient methodology in instructing how to search for a high-density container deployment solution.

7.3 API and Benchmark Lock-in within Coordination Layer

When people talk about serverless vendor lock-in, they are concerned about the portability of functions. However, the real point of this problem depends on the API from other services rather than the function itself. Though some efforts such as Apex [10] and Sparta [123] allow users to deploy functions to serverless platforms in languages that are not supported natively, the BaaS services from different platforms and their API definitions are still different. The challenge with API lock-in is derived from the tight coupling between the user functions and other BaaS components, which can add difficulty to the code migration between different FaaS platforms.

The over-simplified benchmark is another problem with API lock-in. Easy-to-build micro-benchmarks are over-emphasized and used in 75% of the current works [110]. We call for the establishment and open-source of cross-platform real-world application benchmarks besides scientific workflows [64, 89, 119]. However, when decomposing a large service into different functions and then build fine-grained node interconnections, the complexity of the application architecture makes the grading of the function challenging to guide and determine.

8 OPPORTUNITIES IN SERVERLESS COMPUTING

At last, we discuss some future opportunities that serverless computing faces and give some preliminary, constructive explorations to solutions.

8.1 Application-Level Optimization

Application-level optimization requires coordinating between different functions within the application instead of focusing on each general function. Complex interconnections like data dependence and caller-callee relation may conceal between functions. Future works could achieve application-level optimization in two ways: workflow support and workflow scheduling. Workflow support means general support for the inter-connection among functions. We think the following supports are necessary:

- **Better storage.** In some cases, functions need to exchange large ephemeral files with each other. If we register intermediate storage, transferring between storage and functions will take up most of the I/O resource and significantly slow down the response. This consequence is exacerbated in the serverless workflow scenario. Therefore, better storage demands higher priority for metadata exchange between functions within an application.
- **Higher parallelism capacity.** In an example of video processing, multiple recoding instances can be invoked simultaneously in a MapReduce way to speed up the transcoding. Distinctly, there is great potential in parallelism to optimize end-to-end latency. However, higher parallelism is hard to implement due to the considerations on resource utilization management in physical nodes. If a serverless system could provide superior parallelism with sustainable resource overhead, it can further empower users. For example, a serverless system allows multiple queries to be invoked concurrently within an instance with a guaranteed QoS, or optimizes the guest kernel, container runtime, and cgroups to achieve lighter virtualization in high concurrency and density scenarios.

Workflow scheduling drives a scheduling strategy that takes functions' interconnection into account. We think the following considerations are missing in current works:

- **Caller-callee relation.** Caller-callee relation is common in a complex application. Usually, the callee will be invoked after the caller finishes, as Figure 11(a) shows. It gives researchers an opportunity to explore: the system can prewarm function instances and execute them in advance with partial data by the dataflow architecture. As shown in Figure 11(b), Function B, C can start execution earlier while Function A is not complete, thanks to the data dependency

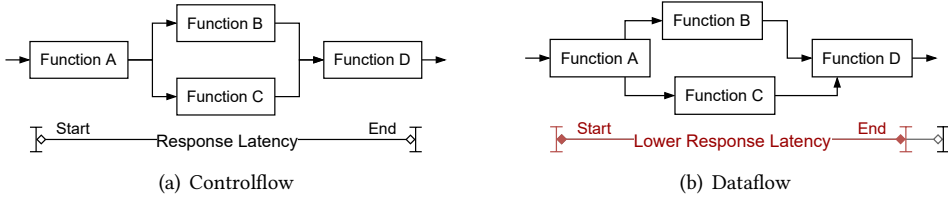(a) Controlflow   (b) Dataflow

Fig. 11. The dataflow architecture for serverless workflow.

rather than function state dependency. In the case of providing an optimized interface with dataflow canonical patterns and applying directly to functions, cloud vendors could enable an application to achieve higher parallelism and lower response latency via a data pipeline.

- **Data locality.** We have mentioned that metadata exchange may continually happen between functions in an application. If two functions with data dependency are scheduled on the same physical node, the data transmission can be significantly reduced by middleware. However, the current serverless system is a data-shipping architecture, which sends data to the code node to parse instead of sending code to the data node. Thus, on the one hand, a serverless system cannot guarantee that the data stored and the workers scheduled are just in the same physical node. On the other hand, frequent code transferring should also be avoided due to security and privacy concerns. Improving data locality can effectively reform the application design from a data-shipping architecture into a code-shipping one [54].

8.2 Robust Performance of Cold Startup Alleviation

Current works usually use predictive methods to reduce cold startups, while they all require functions' historical traces or system-level metrics. By predicting in the near future, the system will enlarge the container pool or prewarm template containers. Nonetheless, it is impractical for each function to collect enough data and build an accurate prediction model. Like Shahrad [118] shows in the Azure trace, about 40% and 30% of registered functions and applications, respectively, are invoked less than ten times daily. This fact also makes it more challenging to collect system-level information periodically for such kind of services. For example, an LRU-based template can maximize the cache hits for hotspot functions startup, whereas cold startups of non-hot functions can not benefit from the cache updating at the system level. The current compromise to this discrepancy is to use a reserved container pool for functions, in spite of a massive waste of resources.

It is crucial to explore the warm-up strategy with strong robustness to performance, especially for functions that are sporadically triggered or sensitive to latency. It requires the serverless controller and load balancer to be more general enough to alleviate cold startups or reduce the performance degradation. They may make decisions based on the information inside the functions, such as the service category, the environment libraries used, and the context diagram, for cold startup prediction and alleviation. For example, a serverless system can build shared images and template containers for functions within the same category, or pack the functions with similar environment configurations and implement more fine-grained inside isolation mechanisms.

8.3 Accelerators in Serverless

Accelerators like GPUs and FPGAs are widely used in many applications such as databases [36, 53] and graph processing [37, 156]. They can significantly speed up the processing of specific tasks, like image processing and machine learning applications. To satisfy the demand for accelerators, cloud vendors furnish accelerators in IaaS (e.g., AWS EC2 P4 and F1 instance) and SaaS (e.g., AWS

SageMaker) manner. However, the inflexibility of such accelerators impedes the instantiation in serverless computing. This circumstance leads to two obstacles: (1) it makes the usage of accelerators less convenient and flexible in the cloud; (2) it limits the range of applications that serverless can support. We think a multiplexing accelerator in serverless is the key to solving these obstacles. For example, some works [98, 150] integrate GPUs into serverless systems, and BlastFunction [14] makes FPGAs available in serverless. However, the current works are still insufficient. We think future research can focus on the following points:

- *Accelerator-aware scheduling*. Accelerators can also be considered a resource in serverless systems, except they have more irreplaceable features than others. Latency-aware scheduling and on-demanding scaling is more expensive on accelerators, stimulating the serverless controller to treat accelerators distinctively. In such a situation, the scheduling strategy should be more conservative when scheduling multiple tasks on one accelerator.
- *Accelerator virtualization*. Virtualization is an essential technology applied in a serverless system. It is used to fulfill runtime environment management, resource isolation, and high security. However, serverless accelerator schemes are not explored insofar as CPU virtualizations. It makes accelerators embarrassing to be integrated into the serverless system. Therefore, to better support accelerators in serverless, accelerator virtualization should be further explored.
- *Automatic batching*. Accelerators usually have strong I/O bandwidth restrictions. Batching queries is a common operation to conquer these restrictions and make full use of accelerators' computation ability. However, the batching operation will introduce redundancy into end-to-end latency. Therefore, a serverless batching strategy that balances utilization and latency should be investigated in future research.

9 CONCLUSION

The rapid development of the cloud-native concept inspires developers to reorganize cloud applications into microservices. Elastic serverless computing becomes the best practice for these microservices. This survey explicates and reviews the fundamental aspects of serverless computing, and provides a comprehensive depiction of four-layered design architecture: Virtualization, Encapsule, System Orchestration, and System Coordination layers. We elaborate on the responsibility and significance of each layer, enumerate relevant works, and give practical implications when adopting these state-of-the-art techniques. Serverless computing is still in its infancy, and the potential remains sealed in forthcoming years.

REFERENCES

[1] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. 2019. A Survey on Scheduling Strategies for Workflows in Cloud Environment and Emerging Trends. *ACM Comput. Surv.* 52, 4 (2019), 68:1–68:36. https://doi.org/10.1145/3325097

[2] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 884–889. https://doi.org/10.1145/3106237.3117767

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, and Anthony Liguori. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[5] Omid Alipourfard, Hongqiang Harry Liu, and Jianshu Chen. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[6] Amazon. 2021. Enabling API caching to enhance responsiveness in AWS. https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html

[7] Amazon DynamoDB Accelerator (DAX): A fully managed, highly available, in-memory cache service 2021. https://aws.amazon.com/dynamodb/dax/

[8] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, and Lukas Rupprecht. 2018. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, Nitin Agrawal and Raju Rangaswami (Eds.). USENIX Association, 265–278. https://www.usenix.org/conference/fast18/presentation/anwar

[9] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 263–274. https://doi.org/10.1145/3267809.3267815

[10] Apex: Serverless Architecture 2021. https://apex.sh/

[11] Vincent Armant, Milan De Cauwer, Kenneth N. Brown, and Barry O'Sullivan. 2018. Semi-online task assignment policies for workload consolidation in cloud computing systems. *Future Gener. Comput. Syst.* 82 (2018), 89–103. https://doi.org/10.1016/j.future.2017.12.035

[12] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. 2016. CloudCache: On-demand Flash Cache Management for Cloud Computing. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 355–369. https://www.usenix.org/conference/fast16/technical-sessions/presentation/arteaga

[13] Naylor G. Bachiega, Paulo S. L. Souza, Sarita Mazzini Bruschi, and Simone do Rocio Senger de Souza. 2018. Container-Based Performance Evaluation: A Survey and Challenges. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, Abhishek Chandra, Jie Li, Ying Cai, and Tian Guo (Eds.). IEEE Computer Society, 398–403. https://doi.org/10.1109/IC2E.2018.00075

[14] M. Bacis, R. Brondolin, and M. D. Santambrogio. 2020. BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 852–857. https://doi.org/10.23919/DATE48585.2020.9116333

[15] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[16] Ioana Baldini, Perry Cheng, Stephen J. Fink, and Nick Mitchell. 2017. The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017*. ACM, 89–103. https://doi.org/10.1145/3133850.3133855

[17] Bartosz Balis. 2016. HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Future Gener. Comput. Syst.* 55 (2016), 147–162. https://doi.org/10.1016/j.future.2015.08.015

[18] Christian Bargmann and Marina Tropmann-Frick. 2019. A Survey On Secure Container Isolation Approaches for Multi-Tenant Container Workloads and Serverless Computing. In *Proceedings of the Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2019, Ohrid, North Macedonia, September 22-25, 2019 (CEUR Workshop Proceedings, Vol. 2508)*, Zoran Budimac and Bojana Koteska (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-2508/paper-bar.pdf

[19] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and Alexandra Shulman-Peleg. 2016. Secure yet usable: Protecting servers and Linux containers. *IBM J. Res. Dev.* 60, 4 (2016), 12. https://doi.org/10.1147/JRD.2016.2574138

[20] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using application knowledge to reduce cold starts in FaaS services. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini (Eds.). ACM, 134–143. https://doi.org/10.1145/3341105.3373909

[21] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. 2012. Bi-criteria Workflow Tasks Allocation and Scheduling in Cloud Computing Environments. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*. IEEE Computer Society, 638–645. https://doi.org/10.1109/CLOUD.2012.83

[22] Nilton Bila, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. 2017. Leveraging the Serverless Architecture for Securing Linux Containers. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino

(Eds.). IEEE Computer Society, 401–404. https://doi.org/10.1109/ICDCSW.2017.66

[23] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 645–650. https://www.usenix.org/conference/atc18/presentation/boucher

[24] Mark Boyd. 2021. "Serverless: IOpipe Launches a Monitoring Tool for AWS Lambda". https://thenewstack.io/iopipe-launches-lambda-monitoring-tool-aws-summit/

[25] FRANK BUDINSKY. 2021. "Canary Deployments using Istio" about the Red-Black and the Blue-green deployment. https://istio.io/latest/blog/2017/0.1-canary/

[26] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, and Rodrigo N. Calheiros. 2019. A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. *ACM Comput. Surv.* 51, 5 (2019), 105:1–105:38. https://doi.org/10.1145/3241737

[27] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.* 25, 6 (2009), 599–616. https://doi.org/10.1016/j.future.2008.12.001

[28] James Cadden, Thomas Unger, Yara Awad, and Han Dong. 2020. SEUSS: skip redundant paths to make serverless fast. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 32:1–32:15. https://doi.org/10.1145/3342195.3392698

[29] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3419111.3421286

[30] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. 2019. In Search of a Fast and Efficient Serverless DAG Engine. *CoRR* abs/1910.05896 (2019). arXiv:1910.05896 http://arxiv.org/abs/1910.05896

[31] Israel Casas, Javid Taheri, Rajiv Ranjan, and Albert Y. Zomaya. 2017. PSO-DS: a scheduling engine for scientific workflow managers. *J. Supercomput.* 73, 9 (2017), 3924–3947. https://doi.org/10.1007/s11227-017-1992-z

[32] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. 2017. A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning. In *2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017*. IEEE, 1–6. https://doi.org/10.1109/GLOCOM.2017.8254046

[33] Liuhua Chen and Haiying Shen. 2017. Considering resource demand misalignments to reduce resource over-provisioning in cloud datacenters. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*. IEEE, 1–9. https://doi.org/10.1109/INFOCOM.2017.8057084

[34] Liuhua Chen, Haiying Shen, and Stephen Platt. 2016. Cache contention aware Virtual Machine placement and migration in cloud datacenters. In *24th IEEE International Conference on Network Protocols, ICNP 2016, Singapore, November 8-11, 2016*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICNP.2016.7784447

[35] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 107–120. https://doi.org/10.1145/3297858.3304005

[36] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA Useful for Hash Joins?. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf

[37] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-Based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/3431920.3439290

[38] Eli Cortez, Anand Bonde, and Alexandre Muzio. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 153–167. https://doi.org/10.1145/3132747.3132772

[39] CRIU: A utility to checkpoint/restore Linux tasks in userspace 2021. https://github.com/checkpoint-restore/criu

[40] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, Dilma Da Silva and Rüdiger Kapitza (Eds.). ACM, 356–370. https://doi.org/10.1145/3423211.3425690

[41] Docker 2021. https://www.docker.com/

[42] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R.

Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[43] Elastic Load Balancing Application Load Balancers. 2021. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/elb-ag.pdf

[44] Execute mode in Fission 2021. https://docs.fission.io/docs/usage/executor/

[45] Erwin Van Eyk, Lucian Toader, and Sacheendra Talluri. 2018. Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Comput.* 22, 5 (2018), 8–17. https://doi.org/10.1109/MIC.2018.053681358

[46] Fission Workflows: Fast, reliable and lightweight function composition for serverless functions 2021. https://github.com/fission/fission-workflows

[47] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[48] Cloud Native Computing Foundations. 2021. "Deployment Strategies on Kubernetes", Software engineer at Container Solutions. https://www.cncf.io/wp-content/uploads/2020/08/CNCF-Presentation-Template-K8s-Deployment.pdf

[49] Google container runtime sandbox 2021. https://github.com/google/gvisor

[50] Xinjie Guan, Xili Wan, Baek-Young Choi, Sejun Song, and Jiafeng Zhu. 2017. Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers. *IEEE Commun. Lett.* 21, 3 (2017), 504–507. https://doi.org/10.1109/LCOMM.2016.2644658

[51] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. USENIX Association, 181–195. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[52] Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. 2021. Survey on serverless computing. *J. Cloud Comput.* 10, 1 (2021), 39. https://doi.org/10.1186/s13677-021-00253-7

[53] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[54] Joseph M. Hellerstein, Jose M. Faleiro, and Joseph Gonzalez. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[55] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. *login Usenix Mag.* 41, 4 (2016). https://www.usenix.org/publications/login/winter2016/hendrickson

[56] Honeycomb: DevOps tool for code inspection 2021. https://www.honeycomb.io/

[57] M. Reza HoseinyFarahabady, Albert Y. Zomaya, and Zahir Tari. 2018. A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform. *IEEE Trans. Parallel Distributed Syst.* 29, 7 (2018), 1442–1455. https://doi.org/10.1109/TPDS.2017.2779502

[58] Hyper-V for Windows containers. 2021. https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container

[59] Shigeru Imai, Thomas Chestna, and Carlos A. Varela. 2013. Accurate Resource Prediction for Hybrid IaaS Clouds Using Workload-Tailored Elastic Compute Units. In *IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013, Dresden, Germany, December 9-12, 2013*. IEEE Computer Society, 171–178. https://doi.org/10.1109/UCC.2013.40

[60] Shigeru Imai, Stacy Patterson, and Carlos A. Varela. 2018. Uncertainty-Aware Elastic Virtual Machine Scheduling for Stream Processing Systems. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, Esam El-Araby, Dhabaleswar K. Panda, Sandra Gesing, Amy W. Apon, Volodymyr V. Kindratenko, Massimo Cafaro, and Alfredo Cuzzocrea (Eds.). IEEE Computer Society, 62–71. https://doi.org/10.1109/CCGRID.2018.00021

[61] Vitalii Ivanov and Kari Smolander. 2018. Implementation of a DevOps Pipeline for Serverless Applications. In *Product-Focused Software Process Improvement - 19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28-30, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11271)*, Marco Kuhrmann, Kurt Schneider, Dietmar Pfahl, Sousuke Amasaki, Marcus Ciolkowski, Regina Hebig, Paolo Tell, Jil Klünder, and Steffen Küpper (Eds.). Springer, 48–64. https://doi.org/10.1007/978-3-030-03673-7_4

[62] David Jackson and Gary Clynch. 2018. An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion,*

UCC Companion 2018, Zurich, Switzerland, December 17-20, 2018, Alan Sill and Josef Spillner (Eds.). IEEE, 154–160. https://doi.org/10.1109/UCC-Companion.2018.00050

[63] Jenkins: DevOps CI tool 2021. https://www.jenkins.io/

[64] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 445–451. https://doi.org/10.1145/3127479.3128601

[65] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[66] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 158–164. https://doi.org/10.1145/3357223.3362709

[67] Kata Containers: an open source container runtime, building lightweight virtual machines. 2021. https://katacontainers.io/

[68] Alireza Keshavarzian, Saeed Sharifian, and Sanaz Seyedin. 2019. Modified deep residual network architecture deployed on serverless framework of IoT platform based on human activity recognition application. *Future Gener. Comput. Syst.* 101 (2019), 14–28. https://doi.org/10.1016/j.future.2019.06.009

[69] Asif Khan. 2017. Key characteristics of a container orchestration platform to enable a modern application. *IEEE cloud Computing* 4, 5 (2017), 42–48. https://doi.org/10.1109/MCC.2017.4250933

[70] Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, and Albert Y. Zomaya. 2020. Automated Fine-Grained CPU Cap Control in Serverless Computing Platform. *IEEE Trans. Parallel Distributed Syst.* 31, 10 (2020), 2289–2301. https://doi.org/10.1109/TPDS.2020.2989771

[71] Ana Klimovic, Yawen Wang, Patrick Stuedi, and Animesh Trivedi. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[72] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101. https://doi.org/10.1145/3399742

[73] Ricardo Koller and Alan Dawson. 2021. Vulnerability Advisor – Secure your Dev + Ops across containers. https://www.ibm.com/blogs/cloud-archive/2016/11/vulnerability-advisor-secure-your-dev-ops-across-containers/

[74] Kubeless 2021. https://kubeless.io/

[75] Kubernetes CronJob 2021. https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/

[76] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. 2019. HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 80–90. https://doi.org/10.1109/ICDCS.2019.00017

[77] Hyungro Lee, Kumar Satyam, and Geoffrey C. Fox. 2018. Evaluation of Production Serverless Computing Environments. In *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE Computer Society, 442–450. https://doi.org/10.1109/CLOUD.2018.00062

[78] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *J. Syst. Softw.* 149 (2019), 340–359. https://doi.org/10.1016/j.jss.2018.12.013

[79] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-Level Image Service for Agile and Elastic Application Deployment. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 727–740. https://www.usenix.org/conference/atc20/presentation/li-huiba

[80] Wubin Li and Ali Kanso. 2015. Comparing Containers versus Virtual Machines for Achieving High Availability. In *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*. IEEE Computer Society, 353–358. https://doi.org/10.1109/IC2E.2015.79

[81] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 615–632. https://doi.org/10.1109/TPDS.2020.3028841

[82] W. Ling, L. Ma, C. Tian, and Z. Hu. 2019. Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 1416–1421. https://doi.org/10.1109/CSCI49370.2019.00265

[83] David Lion, Adrian Chu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2017. Don't Get Caught in the Cold, Warm Up Your JVM. *login Usenix Mag.* 42, 1 (2017). https://www.usenix.org/publications/login/spring2017/lion

[84] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56. https://doi.org/10.1145/3357033

[85] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2020. Triggerflow: Trigger-Based Orchestration of Serverless Workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/3401025.3401731

[86] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 461–472. https://doi.org/10.1145/2451116.2451167

[87] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[88] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. 2019. Optimizing serverless computing: introducing an adaptive function placement algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*, Tima Pakfetrat, Guy-Vincent Jourdan, Kostas Kontogiannis, and Robert F. Enenkel (Eds.). ACM, 203–213. https://dl.acm.org/doi/abs/10.5555/3370272.3370294

[89] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Gener. Comput. Syst.* 110 (2020), 502–514. https://doi.org/10.1016/j.future.2017.10.029

[90] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 218–233. https://doi.org/10.1145/3132747.3132763

[91] Mohammad Masdari, Sima ValiKardan, Zahra Shahi, and Sonay Imani Azar. 2016. Towards workflow scheduling in cloud computing: A comprehensive analysis. *J. Netw. Comput. Appl.* 66 (2016), 64–82. https://doi.org/10.1016/j.jnca.2016.01.018

[92] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. 2015. Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*. IEEE, 559–567. https://doi.org/10.1109/CNS.2015.7346869

[93] Sean McDaniel, Stephen Herbein, and Michela Taufer. 2015. A Two-Tiered Approach to I/O Quality of Service in Docker Containers. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*. IEEE Computer Society, 490–491. https://doi.org/10.1109/CLUSTER.2015.77

[94] M. Garrett McGrath and Paul R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino (Eds.). IEEE Computer Society, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

[95] mirage-skeleton with simple MirageOS applications 2021. https://github.com/mirage/mirage-skeleton

[96] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[97] Anup Mohan, Harshad Sane, Kshitij Doshi, and Saikrishna Edupuganti. 2019. Agile Cold Starts for Scalable Server-less. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*, Christina Delimitrou and Dan R. K. Ports (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[98] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *J. Parallel and Distrib. Comput.* 139 (2020), 32 – 42. https://doi.org/10.1016/j.jpdc.2020.01.004

[99] Hylson Vescovi Netto, Lau Cheuk Lung, Miguel Correia, Aldelir Fernando Luiz, and Luciana Moreira Sá de Souza. 2017. State machine replication in containers managed by Kubernetes. *J. Syst. Archit.* 73 (2017), 53–59. https://doi.org/10.1016/j.sysarc.2016.12.007

[100] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, Jeffrey O. Kephart, Calton Pu, and Xiaoyun Zhu (Eds.). USENIX Association, 69–82. https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen

[101] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[102] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Providence, RI, USA, April 14, 2019*, Jennifer B. Sartor, Mayur Naik, and Chris Rossbach (Eds.). ACM, 59–73. https://doi.org/10.1145/3313808.3313817

[103] OpenWhisk: Serverless functions platform for building cloud applications 2021. https://github.com/apache/openwhisk

[104] Prewarm in Apache OpenWhisk 2021. https://github.com/apache/openwhisk/blob/master/docs/actions-python.md

[105] Prewarm in Azure Functions 2021. https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan

[106] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[107] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 401–417.

[108] Josep Sampé, Marc Sánchez Artigas, Pedro García López, and Gerard París. 2017. Data-driven serverless functions for object storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch (Eds.). ACM, 121–133. https://doi.org/10.1145/3135974.3135980

[109] Josep Sampé, Pedro García López, and Marc Sánchez Artigas. 2016. Vertigo: Programmable Micro-controllers for Software-Defined Object Storage. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. IEEE Computer Society, 180–187. https://doi.org/10.1109/CLOUD.2016.0033

[110] Joel Scheuner and Philipp Leitner. 2020. Function-as-a-Service performance evaluation: A multivocal literature review. *J. Syst. Softw.* 170 (2020), 110708. https://doi.org/10.1016/j.jss.2020.110708

[111] Joel Scheuner and Philipp Leitner. 2020. The State of Research on Function-as-a-Service Performance Evaluation: A Multivocal Literature Review. *CoRR* abs/2004.03276 (2020). arXiv:2004.03276 https://arxiv.org/abs/2004.03276

[112] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja Jayant Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM* 64, 5 (2021), 76–84. https://doi.org/10.1145/3406011

[113] Florian Schmidt. 2017. uniprof: A Unikernel Stack Profiler. In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 31–33. https://doi.org/10.1145/3123878.3131976

[114] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 753–768. https://doi.org/10.1145/3319535.3354252

[115] Srinath T. V. Setty, Chunzhi Su, and Jacob R. Lorch. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 501–516. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty

[116] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2021. Serverless Computing: A Survey of Opportunities, Challenges and Applications. arXiv:1911.01296 [cs.NI]

[117] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 1063–1075. https://doi.org/10.1145/3352460.3358296

[118] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, and Gohar Chaudhry. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[119] Vaishaal Shankar, Karl Krauth, and Qifan Pu. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679

[120] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. SNF: serverless network functions. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 296–310. https://doi.org/10.1145/3419111.3421295

[121] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, and Nam Sung Kim. 2020. BabelFish: Fusing Address Translations for Containers. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 501–514. https://doi.org/10.1109/ISCA45697.2020.00049

[122] Sonarqube: Code quality and security platform 2021. https://www.sonarqube.org/

[123] Sparta: A Go framework for AWS Lambda microservices 2021. http://gosparta.io/

[124] Vikram Sreekanti, Chenggang Wu, and Saurav Chhatrapati. 2020. A fault-tolerance shim for serverless computing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 15:1–15:15. https://doi.org/10.1145/3342195.3387535

[125] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, and Johann Schleier-Smith. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[126] Satish Narayana Srirama and Alireza Ostovar. 2018. Optimal cloud resource provisioning for auto-scaling enterprise applications. *Int. J. Cloud Comput.* 7, 2 (2018), 129–162. https://doi.org/10.1504/IJCC.2018.10014880

[127] Amoghavarsha Suresh and Anshul Gandhi. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing, WOSC@Middleware 2019, Davis, CA, USA, December 09-13, 2019*. ACM, 19–24. https://doi.org/10.1145/3366623.3368136

[128] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. 2017. Understanding security implications of using containers in the cloud. In *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*. 313–319.

[129] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: enabling quality-of-service in serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 311–327. https://doi.org/10.1145/3419111.3421306

[130] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 199–212. https://www.usenix.org/conference/atc18/presentation/thalheim

[131] Raúl Gracia Tinedo, Pedro García López, Marc Sánchez Artigas, and Josep Sampé. 2016. IOStack: Software-Defined Object Storage. *IEEE Internet Comput.* 20, 3 (2016), 10–18. https://doi.org/10.1109/MIC.2016.46

[132] Raúl Gracia Tinedo, Josep Sampé, and Edgar Zamora-Gómez. 2017. Crystal: Software-Defined Storage for Multi-Tenant Object Stores. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, Geoff Kuenning and Carl A. Waldspurger (Eds.). USENIX Association, 243–256. https://www.usenix.org/conference/fast17/technical-sessions/presentation/gracia-tinedo

[133] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. 2020. Adaptive AI-based auto-scaling for Kubernetes. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*. IEEE, 599–608. https://doi.org/10.1109/CCGrid49817.2020.00-33

[134] Using custom Docker images as the action runtime in OpenWhisk 2021. https://github.com/apache/openwhisk/blob/master/docs/actions-docker.md

[135] Alexandre Verbitski, Anurag Gupta, and Debanjan Saha. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1041–1052. https://doi.org/10.1145/3035918.3056101

[136] Jaagup Viil and Satish Narayana Srirama. 2018. Framework for automated partitioning and execution of scientific workflows in the cloud. *J. Supercomput.* 74, 6 (2018), 2656–2683. https://doi.org/10.1007/s11227-018-2296-7

[137] Muhammad Wajahat, Anshul Gandhi, Alexei A. Karve, and Andrzej Kochut. 2016. Using machine learning for black-box autoscaling. In *Seventh International Green and Sustainable Computing Conference, IGSC 2016, Hangzhou, China, November 7-9, 2016*. IEEE Computer Society, 1–8. https://doi.org/10.1109/IGCC.2016.7892598

[138] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[139] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*. IEEE, 1288–1296. https://doi.org/10.1109/INFOCOM.2019.8737391

[140] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. [n.d.]. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019) *(EuroSys '19)*. Association for Computing Machinery. https://doi.org/10.1145/3302424.3303978

[141] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[142] Stephanie Wang, John Liagouris, and Robert Nishihara. 2019. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 338–352. https://doi.org/10.1145/3341301.3359653

[143] Jake Wires and Andrew Warfield. 2017. Mirador: An Active Control Plane for Datacenter Storage. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, Geoff Kuenning and Carl A. Waldspurger (Eds.). USENIX Association, 213–228. https://www.usenix.org/conference/fast17/technical-sessions/presentation/wires

[144] Mingyu Wu, Zeyu Mi, and Yubin Xia. 2020. A Survey on Serverless Computing and Its Implications for JointCloud Computing. In *2020 IEEE International Conference on Joint Cloud Computing*. 94–101. https://doi.org/10.1109/JCC49151.2020.00023

[145] Yulai Xie, Dan Feng, Yan Li, and Darrell D. E. Long. 2016. Oasis: An active storage framework for object storage platform. *Future Gener. Comput. Syst.* 56 (2016), 746–758. https://doi.org/10.1016/j.future.2015.08.011

[146] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. 2019. Adaptive Function Launching Acceleration in Serverless Computing Platforms. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019*. IEEE, 9–16. https://doi.org/10.1109/ICPADS47876.2019.00011

[147] Kejiang Ye, Zhaohui Wu, Chen Wang, Bing Bing Zhou, Weisheng Si, Xiaohong Jiang, and Albert Y. Zomaya. 2015. Profiling-Based Workload Consolidation and Migration in Virtualized Data Centers. *IEEE Trans. Parallel Distributed Syst.* 26, 3 (2015), 878–890. https://doi.org/10.1109/TPDS.2014.2313335

[148] Tianyi Yu, Qingyuan Liu, and Dong Du. 2020. Characterizing serverless platforms with serverlessbench. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 30–44. https://doi.org/10.1145/3419111.3421280

[149] Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2018. SP-cache: load-balanced, redundancy-free cluster caching with selective partition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 1:1–1:13. http://dl.acm.org/citation.cfm?id=3291658

[150] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[151] Haoran Zhang, Adney Cardoza, and Peter Baile Chen. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 1187–1204.

[152] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 1–12. https://doi.org/10.1145/3357223.3362723

[153] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: a programming framework for serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 328–343. https://doi.org/10.1145/3419111.3421277

[154] Ge Zheng and Yang Peng. 2019. GlobalFlow: A Cross-Region Orchestration Service for Serverless Computing Services. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, Elisa Bertino, Carl K. Chang, and Peter Chen (Eds.). IEEE, 508–510. https://doi.org/10.1109/CLOUD.2019.00093

[155] Wenjia Zheng, Michael Tynes, Henry Gorelick, Ying Mao, Long Cheng, and Yantian Hou. 2019. FlowCon: Elastic Flow Configuration for Containerized Deep Learning Applications. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*. ACM, 87:1–87:10. https://doi.org/10.1145/3337821.3337868

[156] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1543–1552. https://doi.org/10.1109/TPDS.2013.111