* * *

SURVEY ARTICLE

Journal Section

Efficiency in the Serverless Cloud Paradigm: A Survey on the Reusing and Approximation Aspects

Chavit Denninnart[1] | Thanawat Chanikaphon[1] | Mohsen Amini Salehi[1]

[1]High Performance Cloud Computing (HPCC) Laboratory, School of Computing and Informatics, University of Louisiana at Lafayette, Louisiana, 70503, USA

Correspondence
Mohsen Amini Salehi, High Performance Cloud Computing (HPCC) Laboratory, School of Computing and Informatics, University of Louisiana at Lafayette, Louisiana, 70503, USA
Email: amini@louisiana.edu

Funding information
CNS-2007209, CNS-2007209

Serverless computing along with Function-as-a-Service (FaaS) is forming a new computing paradigm that is anticipated to found the next generation of cloud systems. The popularity of this paradigm is due to offering a highly transparent infrastructure that enables user applications to scale in the granularity of their functions. Since these often small and single-purpose functions are managed on shared computing resources behind the scene, a great potential for computational reuse and approximate computing emerges that if unleashed, can remarkably improve the efficiency of serverless cloud systems—both from the user's QoS and system's (energy consumption and incurred cost) perspectives. Accordingly, the goal of this survey study is to, first, unfold the internal mechanics of serverless computing and, second, explore the scope for efficiency within this paradigm via studying function reuse and approximation approaches and discussing the pros and cons of each one. Next, we outline potential future research directions within this paradigm that can either unlock new use cases or make the paradigm more efficient.

[*]Equally contributing authors.

1 | INTRODUCTION

1.1 | Serverless Computing Paradigm

The first generation of cloud technology, established around 2010, mitigated the burden of system administration and maintenance via consolidating servers and forming centralized data centers. It is anticipated that the second generation of cloud technology focuses on mitigating the burden of developing cloud-native applications for programmers and solution architects via the serverless computing paradigm [1].

Serverless computing provides the developers with high-level software abstractions, such as functions, a.k.a. Function-as-a-Service (FaaS), and transparently deploying them, such that the user has the illusion of having no servers to manage [2]. Accordingly, modern software engineering methodologies, such as DevOps [3] and Continuous Integration Continuous Delivery (CI/CD) pipelines [4], have adopted the serverless computing paradigm to facilitate rapid cloud-native application development. These methodologies instruct splitting an application into several functions that are invoked periodically or in response to an event. Behind the scene, each function invocation leads to the execution of one or an ordered set of stateless microservice(s) [5].

As shown in Figure 1, the serverless computing paradigm can be defined as the combination of FaaS and BaaS (Backend-as-a-Service) subsystems (*i.e.*, Serverless = FaaS + BaaS [1]). While FaaS focuses on the front-end development of functions in a wide variety of programming languages, BaaS focuses on the transparent and isolated execution of the functions. BaaS is also in charge of data storage, scheduling, monitoring, and transparent elasticity of the functions. It is noteworthy that serverless computing is a loose term, and it does not strictly enforce the user's code to be based on FaaS. Moreover, Serverless solutions are sometimes abstracted from the user perspective. For instance, Amazon Athena [6] is an interactive SQL-like query processing service for Amazon S3 data. Although Athena operates based on serverless principles, its users may consider it as a Platform-as-a-Service (PaaS) instead.

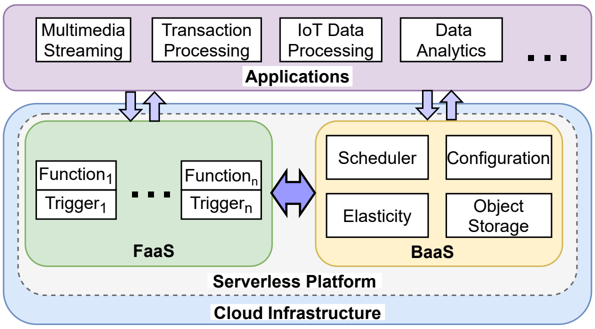

FIGURE 1 The serverless computing paradigm mitigates the burden of developing cloud-native applications via offering a high-level programming abstraction (FaaS) and transparently executing them (BaaS). Different applications can define and trigger functions with minimal configurations needed for each function.

A common approach to handle function calls (henceforth, called *users' requests* or *tasks*) in BaaS is to gather the requests from all triggering sources (*e.g.*, API calls, timer, and events) into a central queue. Then, a resource allocator maps these requests to scalable pools of computing resources. To isolate the requests from each other and to avoid the side effects of using shared resources, often, some forms of task sandboxing, such as containers or micro Virtual Machines (*e.g.*, Firecracker VM [7]), are employed.

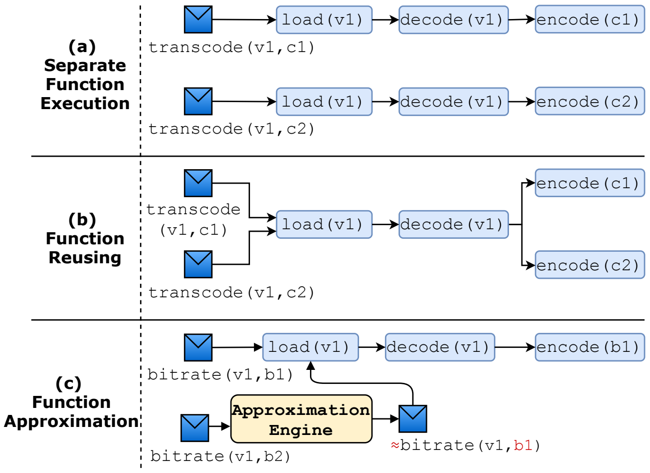

FIGURE 2 Comparison of three scenarios to handle similar function invocations: (a) separately executing functions; (b) reusing `load` and `decode` microservices of the function calls, and (c) approximating a function call and making it compatible with an existing one.

1.2 | Scope for Efficiency in the Serverless Computing Paradigm

The shared and transparent nature of serverless systems offers great potential for efficiency—both from the system and user perspectives. From the system end, metrics such as throughput, utilization, energy consumption, and carbon emission, and from the user end, Quality of Service (QoS) (*e.g.*, turnaround time), and the user's incurred cost can be potentially improved.

The potential efficiency improvement can be unleashed, primarily via smart resource allocation methods that can identify identical and/or similar tasks in the serverless system. As a motivating example, consider the case of a serverless cloud used for processing live video contents before streaming them to the viewers [8]. As shown in Figure 2, the system has `transcode(v,c)` function to change the codec of video segment `v` to `c`; and `bitrate(v,b)` function to change the bit-rate of video segment `v` to `b`. The figure shows possible scenarios of function execution in the system. Consider two invocations of `transcode(v1,c1)` and `transcode(v1,c2)` coexist in the system. Without merging, shown in Figure 2(a), the two invocations separately `load`, `decode`, and `encode` the video. Alternatively, by merging these invocations into one task, shown in Figure 2(b), the `load` and `decode` identical operations can be reused, and then `encode` operation into two different codecs is carried out individually. Because `transcode()` function cannot be approximated, consider `bitrate()` function to explain function approximation. In this case, shown in Figure 2(c), one invocation, `bitrate(v1,b2)`, can be approximated to `bitrate(v1,b1)`, hence, the whole execution chain can be reused. Accordingly, two main directions to improve the efficiency of serverless computing can be enumerated as follows:

1. *Computational reuse* that avoids redundant processing of identical or similar function requests. It focuses on reusing the whole or part of the execution, underlying platform (*e.g.*, container), and allocated resources of a process. A well-established reusing approach is based on caching [9] that can avoid the re-execution of a recent task. While caching is *retroactive* by nature and can only capture identical tasks, in serverless computing, there is a scope for *proactive* reusing. In this manner, similar (or identical) concurrent function calls can be aggregated to

one merged task to reuse a part of (or the whole) computation—even before an instance of the task is complete and cached. For example, the scheduler of a serverless system can detect two workflows that share the same sub-task (or use the same data) ahead of time, and schedule the sub-task or data to be reused.

2. *Approximate computing* that can be employed in contexts where lower quality (less accurate) results can be tolerated (*e.g.*, Machine Learning (ML), live-streaming, etc.). Using approximate computing, the cost, energy, and response time of the serverless cloud can be reduced. Some common approaches for approximate computing include scaling down the precision of the invoked function, downsampling its input data, skipping some computation steps in the function workflow, or approximating the function result from similar or recent invocations.

It is noteworthy that, in use cases where security is a concern, function reuse and approximation can be carried out on subsequent function invocations of the same user. Nevertheless, in other use cases where there is no security concern (*e.g.*, stateless functions performing mathematical operations [10]) reusing and approximation can be implemented more broadly—across users or organizations.

1.3 | Positioning of This Survey Study

To position this survey paper, we describe recent related studies from academia and industry that focus on serverless computing, and then, in Table 1, we summarize the comparison and highlight the topics covered in each work.

(i) Optimizing and extending serverless platforms by Nazari *et al.* [11]: This study focuses on the cold start and startup time optimizations, inter-function communication techniques, and possible extensions to the serverless platform. Although the serverless optimization aspects overlap with our work, we concentrate on the reusing and approximate computing techniques that are not covered in them.
(ii) Architectural design of serverless systems by Li *et al.* [12]: This work decouples the serverless architecture into four stacked layers namely, virtualization, encapsulation, orchestration, and coordination. The survey includes multiple techniques of implementation and efficiency improvements, such as pre-warming and scheduling strategies. In contrast, our work encompasses optimization techniques that are achieved at the intersection of the platform and application levels.
(iii) Eismann *et al.* [13] survey the state of serverless applications by providing a systematic study of serverless applications. They analyze 16 characteristics of 89 serverless applications collected from open-source projects and literature and compared the results with 10 related survey studies and datasets to analyze community consensus.
(iv) Raza *et al.* [14] measure various features and performance metrics of the commercial and open-source FaaS platforms. They particularly study the performance per cost benefits from the application developer's perspective. Several optimization techniques are discussed to present a developer's decision-making factor in choosing a FaaS platform.
(v) Serverless computing survey conducted by Mampage *et al.* [15]: This work identifies aspects of serverless resource management and proposes a taxonomy of elements that influence these aspects, encompassing characteristics of system design, workload attributes, and stakeholder expectations. However, the proposed taxonomy does not include efficiency improvement aspects such as approximate computing.
(vi) The survey of opportunities and challenges in serverless by Li *et al.* [16]: This paper collects papers reflecting the state of the art of serverless computing. They identify the serverless computing model's challenges and study how the existing works address them. Their work presents various areas that need further attention from the research community, on the contrary, our work particularizes on the scope for efficiency in the serverless paradigm.

Prior studies			Naz. *et al.* [11]	Li *et al.* [12]	Eism. *et al.* [13]	Raza *et al.* [14]	Mam. *et al.* [15]	Li *et al.* [16]	Chak. *et al.* [16]	CNCF [18]	Data-dog [19]	IBM [20]	This sur-vey
Platform aspects	Challenges	stateful serverless	✓	✓			✓	✓					✓
		cold start	✓	✓		✓	✓	✓					✓
		security		✓			✓	✓	✓			✓	✓
	Adoption	trend			✓	✓			✓	✓	✓	✓	
		compare solutions		✓	✓	✓	✓	✓	✓	✓			✓
Performance aspects	Reusing	deterministic	✓	✓		✓	✓	✓					✓
	Approx.	data level											✓
		instruction level											✓
	Reuse & approx.	semantic											✓
	Scheduling		✓	✓		✓	✓	✓	✓				✓
Future direction			✓	✓		✓	✓	✓	✓				✓

TABLE 1 Positioning of this survey study with respect to prior survey studies in the serverless computing area.

(vii) Chakraborty *et al.* [17] compare cloud frameworks on the basis of different paradigms. Cloud computing, fog computing, Cloud of Things (CoT), and Fog of things (FoT) paradigms are discussed with their respective properties and limitations. Our work is conducted with a comparable aspect, but dives deeper into the serverless paradigm.
(viii) CNCF annual survey [18]: It is a statistical report that contains a list of up-to-date technologies and serverless providers.
(ix) The state of serverless by Datadog [19]: The report analyzes the current trends of serverless applications on three leading cloud providers namely, AWS, Google Cloud, and Microsoft Azure.
(x) Enterprise-level serverless systems by IBM [20]: This survey was conducted by IBM Market Development & Insights (MD&I) and encompasses the perception of enterprises from the serverless paradigm, in areas like user experience, security, and CEO opinions.

A summary of the characteristics of these studies is compared in Table 1. In the table, we can see that several prior works focus on the platform and specification aspects of the serverless systems, whereas, our work concentrates on the potential to improve the performance of serverless systems via novel computational reuse and approximate computing techniques.

1.4 | Paper Structure

Before studying efficiency in serverless cloud computing, we need to learn about the nuts and bolts of serverless computing. Accordingly, in the rest of this survey, we first dive deep into the serverless computing details and study its anatomy. Next, we compare the serverless systems against other distributed computing paradigms and discuss why a separate study is required to make these systems efficient. Then, we concentrate on the efficiency of the serverless systems. The examples described in the previous part only show one possible scenario for reusing and approximating a function. We are to explore the potential for different forms of function reuse and approximation that can be unleashed, thereby, enabling efficient serverless cloud computing. An overview of the approaches that are studied in this work is shown in Figure 7.

We believe this survey study can help the research community to further develop these areas and build more efficient serverless computing platforms. The rest of the paper is organized as follows: Section 2 introduces the current state of commercial and research-based serverless computing platforms. Section 3 address the unique characteristics

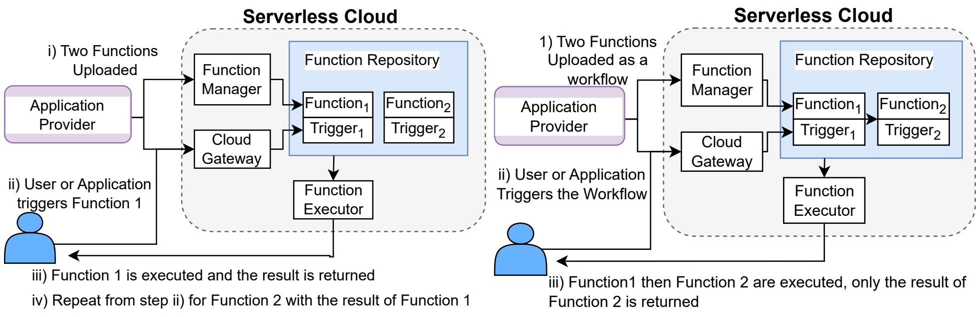
Serverless Cloud
i) Two Functions Uploaded
Application Provider
Function Manager
Cloud Gateway
Function Repository
Function$_1$
Trigger$_1$
Function$_2$
Trigger$_2$
Function Executor
ii) User or Application triggers Function 1
iii) Function 1 is executed and the result is returned
iv) Repeat from step ii) for Function 2 with the result of Function 1
Serverless Cloud
1) Two Functions Uploaded as a workflow
Application Provider
Function Manager
Cloud Gateway
Function Repository
Function$_1$
Trigger$_1$
Function$_2$
Trigger$_2$
Function Executor
ii) User or Application Triggers the Workflow
iii) Function1 then Function 2 are executed, only the result of Function 2 is returned

FIGURE 3 An example workflow with two functions uploaded by the application provider. In this example, the Function trigger is configured as an on-demand API request and the result of Function 1 serves as the input for Function 2. Scenario A (on the left) sets up two functions separately. Scenario B (on the right) chains two functions on the serverless platform.

of serverless that provide potential and burden to reusing and approximation techniques. Then, Section 4 discusses the potential of computational reuse on various parts of the serverless computing platform. Next, Section 5 discusses the approximate computing techniques that can be applied on serverless computing platforms. Section 6 lists some potential development directions to increase data and compute reusability in serverless computing platforms. Finally, we conclude this paper in Section 7.

2 | NUTS AND BOLTS OF THE SERVERLESS COMPUTING PARADIGM

2.1 | Introducing Serverless Computing

Serverless computing and FaaS abstract the users from both server maintenance and management. However, unlike PaaS, using serverless entails breaking the application into multiple functions that each one can potentially be developed in a different programming language. Then, the entire function execution management, such as resource allocation, scaling, scheduling, fail-over, and platform configurations, are transparently handled by the underlying serverless platform. As such, this paradigm simplifies the software development process and enables the users to become solution-oriented and focus on their business logic, rather than specific server configuration details.

2.2 | Functions Triggers in Serverless Computing

FaaS enables users to develop functions and define a way to enact their execution, called *function triggers*. FaaS platforms desire the user's application to be constructed as a set of single-purpose functions that receive a set of input parameters and yield a set of outputs [21]. Each function is managed and scaled independently by the BaaS platform.

Function triggers are typically based on API calls (*e.g.*, web requests), timers, and completion of other events (*e.g.*, completion of another task) [22,23]. As shown in Figure 3, two main ways to designate function triggers for serverless workflows are: (a) Defining the trigger for each function individually; and (b) Utilizing a *workflow* schema (*e.g.*, JSON workflow definition [24]). In scenario A of Figure 3, the application provider creates two functions separately (through the function manager) in the serverless platform. Upon triggering the workflow through the cloud gateway, Function 1 is executed first. Then, the result is returned to the application provider, and the provider feeds the result of Function 1 to trigger Function 2 and get the intended result. In scenario B, however, the application provider chains two functions as a workflow on the cloud. In this case, upon Function 1 completion, the serverless platform automatically triggers Function 2 with the result of function 1 before returning the final result to the user.

Although defining an individual trigger for each function is simpler, more flexible and more popular, defining multiple function triggers together in form of a workflow schema is more advantageous. First and foremost, the schema can provide useful metadata for the resource allocator in BaaS to identify parallelizable tasks and schedule them together, thereby, improving the resource utilization and users' QoS (*e.g.*, waiting time). Second, using the schema, complex function workflows can be defined that otherwise would be time-consuming and error-prone to build [25]. The third advantage is the portability and reusability that making use of the workflow schema offers. While per-function triggers are tied to each function, the workflow of functions is defined within a schema file with a specific syntax [25]. The schema can be used to effortlessly re-deploy the application on the same serverless platform. However, re-deploying the application on the different public serverless platforms is not achievable at this time, due to the lack of platform-agnostic standards.

2.3 | The Matter of "Function State" in Serverless Computing

Functions in serverless computing are originally designed to be stateless. That is, a function does not maintain (*i.e.*, memorize) any state data (*e.g.*, shared variables) between consecutive invocations, and its output is merely subject to its input arguments. Statelessness is, in fact, a primary practice in functional programming [26] that prevents side effects [27], thereby, improving software robustness and predictability. This implies that, for a given input, a stateless function (*e.g.*, mathematical operation, query/string preprocessing, etc.) always yields the same output, thus, the function results can be reused (*e.g.*, via caching). In addition, stateless functions mitigate the overhead of serverless platforms by relieving them from maintaining data consistency and synchronization in executing functions [2].

Despite the benefits of stateless functions, some applications naturally demand the state to be maintained. Refactoring the stateless version of these applications makes them prohibitively inefficient. For instance, a big data analytics workload (*e.g.*, for semantic search [28]) cannot afford to load the entire dataset for each function call, nor can it afford to forward the output to other functions along the workflow. A common approach to circumvent this situation is to persist the state on the external storage services [29]. However, Pu *et al.* [30] demonstrate that employing external storage to carry out serverless data analytics is up to 500$\times$ slower than using IaaS clouds. Note that, once the state of a function is persisted on the external storage, it behaves as a stateful function and its results are not reusable anymore (unless the state domain is small and cacheable).

The matter of state is still an open challenge in the serverless paradigm. Several research works have been undertaken to offer a built-in stateful serverless solution [31]. Such solutions often employ some forms of key-value and/or file-based storage. Sreekanti *et al.* [32] develop a stateful serverless platform, called Cloudburst, using Anna [33], which is auto-scaling key-value storage, to persist the state. Pu *et al.* propose Shuffling [30], a stateful domain-specific serverless platform for data analytics, with a hierarchical state persistence—a fast layer on the memory and a slower one on the device storage. Schleier-Smith *et al.* develop a dedicated POSIX-like file-storage system to enable stateful serverless computing, called FAASFS [34]. It tackles multiple challenges of providing a shared file system across functions, such as cache and transactional consistency [35, 36]. Shillaker and Pietzuch evaluate stateful functions within the FAASM platform [37] via sharing the state in form of both memory segments and files. Kraft *et al.* propose Apiary [38], a serverless framework for data-centric functions. It compiles application logic into a database stored procedures to improve performance. On the other hand, some solutions for the function state operate based on the actor model [39]. Azure Functions provide support for the Entity functions [40]. Kalix [41] and Apache Flink [42] offer an implementation of the actor model for stateful serverless.

2.4 | Function Isolation in Serverless Computing

In principle, virtualization is not a must for FaaS and serverless cloud offerings. A user can essentially call a function using a command in the general form of `client.invoke(FunctionName='F',Payload=Data)` [43,44]. Upon invocation, the FaaS engine can interpret the function and form a task that can be then directly executed on the host machine (*i.e.*, bare-metal resource provisioning). However, lack of isolation in bare-metal raises security concerns, particularly, when there are coexisting tasks from multiple users on shared computing resources. Therefore, some form of sandboxing is required to isolate the execution environment of each function call. Broadly speaking, such isolation can be provided at the following levels: application-level runtime frameworks (WebAssembly [45] and language runtime [46]), Operating system-level (containerization), and hardware-level (virtualization). The layer-view of each isolation platform for the serverless functions is provided in Figure 4 [47]. The software stack of each platform implies the overhead imposed by that platform. In this figure, the left-most boxes serve as the legend—dedicating a color for each layer. The white space(s) in each isolation platform express the absence of the corresponding layer(s), represented on the left-most side. For example, On the Hypervisor (blue color) and Host OS Kernel (dark gray color) level, the existence of these two colors on the VM column implies that both of these layers exist in the VM technology. Similarly, for Micro-VM and Uni-Kernel, it is blue only (without gray) meaning that only the Hypervisor exists (without Host OS Kernel).

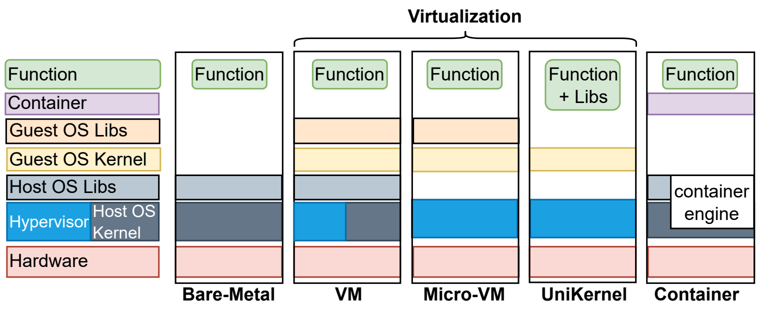

FIGURE 4 A bird-eye view of the underlying layers of various isolation platforms for serverless functions. From left to right, respectively, there are functions on bare-metal (*e.g.*, via WebAssembly), various forms of VMs, and a container. The number of layers implies the overhead of each isolation platform.

WebAssembly: WebAssembly (a.k.a. Wasm) is an open standard that enables the generation of portable binary code from various high-level programming languages and interfaces the binary code with the underlying host environment. The binary can be executed both as a standalone code or within a Web browser. WebAssembly and, particularly, its software-fault isolation (SFI) feature [37] provides software-level isolation that can be used by serverless function solutions. FAASM [37] is an instance of a serverless framework that executes functions on WebAssembly. Also, in Kruslet [48], containers are replaced with WebAssembly in the context of the Kubernetes orchestrator [49]. Kruslet listens to the Kubernetes event stream and upon receiving a task request, it executes the task on WebAssembly runtime [2], instead of creating (container) pods.

Using Wasm in a WebAssembly-based serverless platform eases function migration across devices, edge, and cloud. Since Wasm runtime is available on web browsers, the same packaged serverless function not only can be executed on the cloud server, but also on the end-users' web browsers. This allows the web application to have an adaptive function scheduling such that in a certain scenario (*e.g.*, user device has low battery or limited computing power), the function is executed on the cloud, whereas, in other scenarios (*e.g.*, low bandwidth) the function is executed locally on the device.

Although WebAssembly can satisfy the needs of small-size functions, in many use cases, a specific environmental setup, such as software packages and libraries, is required. Just-in-time preparation of these dependencies for each function execution is time-consuming. Moreover, making use of WebAssembly implies compiling the functions to WebAssembly which curbs the generality of the serverless solution. Therefore, for the sake of generality and to maintain cost- and time-efficient preparation of functions, container or lightweight virtual machine (VM) technologies are more commonly used in the serverless domain.

Language runtime: Similar to using WebAssembly, the language runtime can also be used as a lightweight execution environment. Dukic *et al.* [50] demonstrate overhead reduction via co-locating concurrent instances of the same function within the same runtime. However, it limits supported language and requires application modifications. Bruno *et al.* [51] develop a virtualized polyglot language runtime for serverless applications, called Graalvisor, using Truffle [52], an interpreter-writing framework, to relax language limitations.

Containerization: The most common way to package and isolate the function is through containerization [53]. In this technology, a function is encapsulated within a widely-accepted container standard, termed Open Container Initiative (OCI) format [54], which is supported in all modern containerization solutions. Any programming language and/or software dependency can be supported, thus, the desired generality of serverless is accomplished. Unlike VMs that emulate the entire operating system stack, containers share the host kernel, thereby, both the memory and storage footprints are reduced. This pattern of reusing the kernel is extended to other layers within the container image. Specifically, container images have a layered structure that encourages reusing software packages across these images. Container engines use a method, called union mounting, through which a container is formed dynamically (*i.e.*, on-demand) via fetching its (read-only) layers at the runtime. Further details about union mounting are discussed in Section 4.3.

Virtualization: Virtualization is the traditional method of providing strong isolation in cloud computing [55]. Due to including the whole operating system and application stack in the VM image, in general, VMs suffer from high memory and storage footprints. In addition, VMs introduce a high startup delay [56] to boot up, hence, they are not a perfect fit for frequent starts and terminations inherent to the serverless functions [23]. As such, VMs are usually employed as the underlying platform of the serverless frameworks, rather than an isolator of each function. In this case, function isolation within the VM is offered either via application-level solutions (*e.g.*, WebAssembly) or containers.

Micro-VM: Although VMs are generally not ideal isolators for functions, there are some notable efforts to customize VMs for functions. Firecracker [7] is an AWS open-source project that provides a lightweight VM (a.k.a. micro-VM) with high isolation and low startup delay. It is being used by the AWS FaaS and serverless platforms (*e.g.*, AWS Lambda and AWS Fargate). Firecracker works similar to other full VM technologies that offer an isolated operating system environment to the user. However, unlike other KVM-based VMs [57] that sit in the user space on top of QEMU [58] (a machine emulator that enables VMs), Firecracker directly communicates with the KVM layer via a customized emulation stack. While such a highly simplified emulation stack is sufficient for ordinary tasks, it lacks some notable features, such as libraries to support GPU and specialized CPU instructions. As a result, serverless platforms that use Firecracker, such as AWS Lambda, are currently unable to provide services demanding advanced facilities, e.g., those needed for GPU-based machine learning algorithms [59]. Another downside of Firecracker is its deployment

	Memory Cost	Startup Delay	Security	Main Pro	Main Limitation
WebAssembly	Low	Short	Software-fault Isolation	Wasm runtime available on web browsers	Limited language and library support
Language runtime	Low	Short	Software-fault Isolation	Ultra lean for an application-level isolation	Limited language and library support
Containerization	Low	Short	Containerization	Built-in container image partial reusing	Container platform security threats
VM	High	Long	VM Memory Isolation	Accelerator access; Proven isolation	High startup overhead
Micro-VM	Low	Short	VM Memory Isolation	Ultra lean for a VM-based isolation	*Unsuitable for task that require large software dependencies
Unikernel	Low to Medium *	Depends *	VM Memory Isolation	Better compatibility than Micro-VM	* Stores library within the function image, thus, memory footprint may increase

TABLE 2 Comparison of isolation techniques with respect to their memory cost, startup latency, security, and their main pros and limitations for serverless platform.

inflexibility because it only has to be deployed on top of the KVM hypervisor.

Unikernel [60] is another lightweight VM-based technology. Similar to micro-VMs (see Table 2 which compares all the aforementioned isolation techniques), it bypasses the user space of the hypervisor. Moreover, it bypasses the user space of the guest operating system too. Thus, to run a function inside Unikernel, required guest OS libraries have to be incorporated at the application level. That is, the applications that are deployed within Unikernel have to encapsulate all their required libraries. Storing libraries on each function image causes substantial data redundancy and overhead. As such, Unikernel utilization becomes limited to small functions that do not require large dependencies.

2.5 | Cold Start vs Warm Start Functions

In a container-based serverless platform, a functioning container that resides in the memory to be launched rapidly is generally referred to as a *warm start* container. In contrast, a function that must be loaded from the storage system in an on-demand manner is referred to as a *cold start* container [5]. The cold start function involves loading the container image that (depending on the container size) imposes a nontrivial time overhead and can potentially dominate the function execution time [5]. The overall cold start overhead of a function can be approximated by multiplying the function invocation frequency and the cold start overhead.

Note that, calculating the cold start overhead can be further complicated when other system factors, such as elasticity and storage location, are taken into consideration. Importantly, there can be a lower-level cold start in which a function has to undergo the elasticity overhead and wait for the underlying VM (or hardware) to be made available before it can be loaded into its memory. Depending on the storage location, the cold start overhead can be subdivided into multiple tiers of cold, namely *local storage cold* and *repository cold*. In the former, the function container can be retrieved from the local storage, whereas, in the latter, the function must be retrieved from remote storage (*e.g.*, on a central cloud), which implies a more substantial overhead. These factors show that in an efficient serverless system, the BaaS subsystem must handle the cold/warm start of each function based on its characteristics. In Section 5.6.2, we discuss multiple approaches to strategically manage the containers and mitigate the cold start frequency.

2.6 | Serverless Cloud Solutions

In this part, we first survey various commercial and open-source serverless cloud systems, and then, compare them (in Table 3) based on the aspects described in the previous sections. We note that, in addition to the platforms listed in Table 3, there have been several other serverless computing projects (*e.g.*, Fission Workflow [61], Kubeless [62], and

Serverless System	Open-Source	Container Feed Format	Supported Programming Languages in Function Feed Format	Underlying Platform	Stateful	Function Start	Workflow Support
AWS Lambda	✗	✓	JS, Python, Go, Java, Ruby, .NET, PowerShell	Firecracker	✗	Cold/ Warm	✓
AWS Fargate	✗	✓	Function Feed Format not supported	Container on Firecracker	✓	Warm	✓
Microsoft Azure Functions	✗	✓	JS, Python, Java, .NET, PowerShell	Kubernetes /Azure Arc	✗	Cold/ Warm	✓
Microsoft Durable Function	✗	✓	JS, Python, Java, .NET, PowerShell	Kubernetes /Azure Arc	✓	Warm	✓
Google Cloud Functions	✗	✗	JS, Python, Go, Java, PHP, Ruby, .NET	Google App Engine	✗	Cold/ Warm	✗
Google Cloud Run	✗	✓	JS, Python, Go, Java, PHP, Ruby, .NET, Kotlin	Kubernetes	✓	Cold/ Warm	✓
IBM Cloud Functions	✗	✓	JS, Python, Go, Java, PHP, Ruby, .NET,	OpenWhisk	✗	Cold/ Warm	✗
OpenFaaS	✓	✓	JS, Python, Go, Java, PHP, Ruby, .NET,	Kubernetes/ OpenShift	✗	Cold/ Warm	✗
Apache OpenWhisk	✓	✓	JS, Python, Go, Java, PHP, Ruby, .NET,	Kubernetes/ OpenShift/ Docker	✗	Cold/ Warm	✗
Platform9 Fission	✓	✗	JS, Python, Go, Java, PHP, Ruby, .NET, Rust, Swift	Kubernetes	✗	Cold/ Warm	✗
Oracle Fn	✗	✓	JS, Python, Go, Java, Ruby, .NET	Container	✓	Cold/ Warm	✓
Knative	✓	✓	JS, Python, Go, Java, Rust	Kubernetes	✓	Cold/ Warm	✓
Nuclio	✓	✓	JS, Python, Go, Java, .NET, Shell	Kubernetes/ Docker	✓	Cold/ Warm	✓
funcX	✓	✗	Python	Kubernetes/ Docker	✗	Cold/ Warm	✗

TABLE 3 Comparison of the major serverless computing platforms.

Iron Function [63]) that were discontinued, thus, we have excluded them from the comparison table. Because programming languages have an impact on function efficiency [64,65], in Table 3, we include the list of language runtimes that are officially supported by each serverless system to help function developers in choosing a feasible system and language for their function. Lastly, in Section 2.6.3, we leverage our observations from the studied serverless systems and design a generic architecture that includes the main components of the serverless systems.

2.6.1 | Public Serverless Cloud Platforms

FaaS and serverless computing have commercially been made available via AWS Lambda service [66] for the first time in 2014. AWS Lambda executes each function based on a user-defined trigger and charges the user only for the actual resource usage time (*i.e.*, the function execution time). The Lambda service arguably pioneered and shaped other FaaS services. Nowadays, Amazon also offers other serverless computing services—most notably AWS Step Functions [67] and AWS Fargate [68]. AWS Step Functions is a workflow service that can chain a sequence of Lambda functions and other AWS services to build a serverless application. It manages the workflow in terms of scheduling, failure, and parallelization so that the users can focus on the higher-level business logic. Alternatively, AWS Fargate operates based on containers rather than functions. It is an example of a serverless service that is not built from a FaaS platform.

After AWS, multiple competing serverless computing cloud services, such as Azure Functions [69], Google Cloud

Functions [70], and IBM Cloud Functions [71] have emerged. These services are consistently evolving with different sets of features. Notably, recently, Microsoft Azure released Durable Functions [72] to extend the Azure Function service to support stateful workflows.

2.6.2 | Private Serverless Cloud Platforms

Although public serverless platforms are increasingly popular, they come with the vendor lock-in risk and the trustworthiness issue that is inherent to public clouds. Therefore, multiple open-source projects have been developed to allow serverless deployment on self-hosted servers [73]. The details of stateful serverless platforms are described in Section 2.3. OpenFaaS [74] and Apache OpenWhisk [75] are two popular open-source serverless platforms that dominate the private serverless cloud market.

OpenFaaS handles each function as a container that is deployed through Kubernetes. Therefore, a user can develop functions in the programming language of her choice. A packaging script creates a container image with the user's function encapsulated in it. Each function container is stored and managed in the Docker Registry [76] and also in the function store. While OpenFaaS is open-source and free to use, OpenFaaS PRO [74] is developed for commercial purposes.

OpenWhisk is another popular open-source serverless cloud platform backed by the Apache foundation [77]. In comparison with OpenFaaS, OpenWhisk has a bigger developer community and many more features. Similar to OpenFaaS, the OpenWhisk project is also based on Kubernetes. It also utilizes many features from other open-source products, such as Kafka [78], CouchDB [79], Nginx [80], Redis [9], and Zookeeper [81]. This allows the OpenWhisk to be very scalable and, at the same time, feature-rich. However, this makes the learning curve of deploying and managing OpenWhisk steeper. OpenWhisk is also offered commercially on IBM Cloud Functions [71]. Fission [82] is another serverless platform heavily based on Kubernetes. However, in comparison with Kubeless, Fission has less dependency on Kubernetes as they implement their own function management components, whereas, Kubeless relies on Kubernetes components and extends only the required features that are missing to become a serverless platform.

Fn Project [83] is an open-source platform that works with Docker containers in its underlying layer. It supports API-based event triggers (*e.g.*, in form of web requests). Fn Flow [84] is an alternative version of Fn that can support workflows. Although both projects have attracted limited users from the open-source community, Oracle still offers them commercial serverless solutions.

Knative [85] is a fast-growing open-source project led by IBM and Google. Similar to Kubeless [62], which has been discontinued, Knative sits on top of Kubernetes and enables it to handle serverless workloads. According to the Cloud Native Computing Foundation (CNCF) survey [86], Knative has been the top installable serverless solution in 2020. Knative project includes three main components, namely *Build*, *Serve*, and *Event*. Build is in charge of source code management, containerization, and making it deployable by Kubernetes. Serve deals with service deployment, managing microservice revisions, routing requests to different versions of microservices, automatic scaling, and scaling to zero. Finally, Event takes care of creating function triggers and forming workflow pipelines. Knative is one of the backends supported by KNIX MicroFunctions [87] that is a well-known serverless platform in the academic area (formerly known as SAND [88]).

Nuclio [89] is another platform that works on top of Kubernetes. It offers popular data science tools integration and GPU-based machine support. Lastly, the aforementioned solutions that leverage centralized management can potentially lead to some form of vendor lock-in. Hence, frameworks for decentralized function execution are proposed [90, 91]. *f*uncX [92] is another open-source FaaS platform for federated ecosystems. It supports function execution on remote heterogeneous computers and clouds.

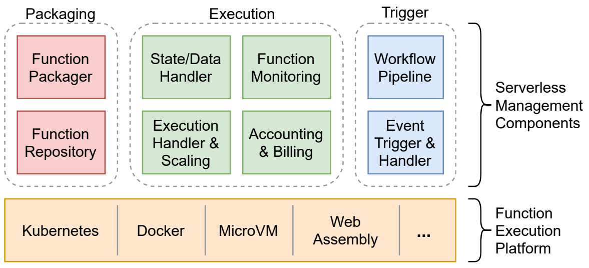

FIGURE 5 Common architectural components of a serverless framework. Note that not all these components exist in every framework. For example, Function Packager exists only on the frameworks that do not run functions straight from the source code. The Function Execution Platform shows the list of common execution platforms for functions. However, this list is not inclusive, as there can be other execution platforms emerging or already deployed in the proprietary systems.

2.6.3 | General Architecture of Serverless Clouds

Considering the common architectural components of the studied serverless platforms, we can design a generic architecture, depicted in Figure 5, that is composed of an underlying function execution framework and eight components on top that take care of different aspects of serverless management. The “Packaging” component contains the Function Packager that wraps the function code into isolated units, such as container image, MicroVM image, and WASM compiled code [37]. The “Execution” component is considered the engine of the serverless systems and is in charge of executing and monitoring functions. It also assures that the functions remain scalable and are accessible with low latency (via warm-starting functions). Particularly, it supports scale-to-zero when the function is unused. The State Handler module only exists to support stateful functions and is not used in stateless serverless platforms. The “Trigger” component on one end is used by the developer to declare the conditions for initiating a function call. On the other end, it connects with the Execution component to execute the task(s) based on the user requirements. The Workflow Pipeline is a high-level tool (*e.g.*, AWS Step functions [67]) that enables developers to create new services by defining a function chain.

Note that, a given serverless platform may only have a subset of these components, and it may categorize them sightly differently. For instance, Knative’s main components (Build, Serve, and Event) can be mapped to *Packaging*, *Execution*, and *Trigger* group of components. Function Packager is only needed if the framework requires function code to be transformed into the container or other forms of the compiled function. Workflow Pipeline only exists in the frameworks that support function workflows.

3 | SERVERLESS COMPUTING VS OTHER DISTRIBUTED COMPUTING PARADIGMS

Many characteristics of serverless systems, including computational reuse and approximate computing, share similarities with other distributed computing paradigms, such as High-Performance Computing (HPC) systems, Grid computing, and various forms of cloud computing. However, the serverless paradigm exposes characteristics and obstacles that call for solutions specifically developed for them. In this section, we compare these various distributed computing paradigms (in Table 4) and discuss those aspects of the serverless demanding new solutions.

Considering Table 4, serverless tasks are usually user-defined (unlike pre-defined tasks in P2P, IaaS, PaaS and SaaS),

	Task latency sensitivity	Request nature	Task types	Approx. hardware accessibility	Stateless
HPC	Low	Reservation	User-defined	Direct access	No
Grid	Low	Reservation	User-defined	Direct access	No
P2P	Can be high	On demand task	Predefined	Direct access	No
IaaS	Low	VM Lease	Predefined	Virtualized	No
PaaS	Can be high	On demand task	Predefined types	Abstracted	No
SaaS	Generally high	Interactive task	Predefined	Abstracted	Can be
Edge-to-cloud	Definitely high	Interactive task	Generally predefined	Abstracted	Generally not
Serverless	Generally high	On demand/ Interactive task	User-defined	Abstracted Originally	Yes

TABLE 4 Characteristics of the tasks and platforms across different distributed system paradigms.

are requested upon the user's demand (as opposed to time/resource reservation or interactive task), and then are interactively served; as opposed to HPC systems that are reservation-based and offline. Unlike HPC and Grid computing paradigms, serverless tasks are often latency-sensitive. These characteristics entail having low-latency solutions to detect reusable tasks in the serverless system and acting upon them, whereas, existing solutions for HPC/Grid systems (*e.g.*, [93–95]) are designed for large offline tasks. Although the serverless functions are user-defined, they are stateless and fine-grained (*i.e.*, they are typically single-purpose), rather than complex stateful applications in other paradigms. These characteristics make the serverless systems potent for task duplication and reusing with proven performance gains [96,97]. While serverless platforms can execute tasks in a pipeline to complete a workflow, each task in the workflow is often executed on the same unified execution engine. This is different from what commonly occurs across edge-to-cloud continuum, where the edge generally takes care of the pre-processing and the core processing happens on the cloud.

The user-defined nature of functions in the serverless paradigm implies that the context-specific solutions have limited applicability. Moreover, the high-level abstractions offered by the serverless paradigm imply that the resource allocation and its related optimizations are accomplished by the platform, whereas, approximate computing techniques often require direct access to the hardware resources. As such, while users in the HPC, Grid, and IaaS systems can tweak their tasks to exploit the approximator hardware [98] or use Dynamic Voltage Frequency Scaling (DVFS) techniques [99] with some control over the underlying hardware, the serverless paradigm abstracts these aspects from the user. Hence, if such capability is to be offered, it'd be the serverless platform's responsibility to provide them as an abstracted platform feature.

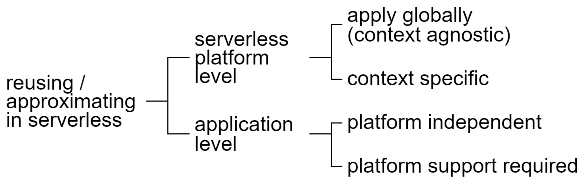
reusing / approximating in serverless
serverless platform level
application level
apply globally (context agnostic)
context specific
platform independent
platform support required

FIGURE 6 Categorizing reusing and approximation techniques that can be used in the serverless paradigm.

Generality and supporting user-defined task-type are the main selling points of the serverless paradigm. However, these features also makes it challenging to apply reusing and approximation techniques on a serverless system with a wide variety of task types. Such techniques, in theory, can be applied either from the serverless platform side, from the function (application) side, or collaboratively from both sides. As shown in Figure 6, we can categorize the reusing and approximation techniques of the serverless systems into four types: (1) Global techniques applied at the platform level, regardless of the application context. For instance, container or container image reusing techniques [100–102] that can universally improve the performance, regardless of the task types; (2) Selective techniques applied to certain tasks (contexts) at the platform level. For instance, consider a result caching system [103] that reuses the results of recently executed tasks. Such techniques are only safe to apply to functions that are truly stateless, such that the function execution does not have side effects on the internal/external states; (3) Application-specific techniques independent of the serverless platform. For instance, consider an application that can approximate the output precision according to the input and context, regardless of the underlying platform; and (4) Application-specific techniques with platform support. For instance, consider a function that can do approximate computing via ASICs. The function must be developed to allow such approximation, and the serverless platform must recognize when to assign such function to the ASICs.

In this survey study, we not only target the first two types of techniques that directly involve the serverless computing platform, but we also discuss the fourth type, which is application-specific techniques that require support from the serverless framework.

4 | REUSING OPPORTUNITIES IN THE SERVERLESS CLOUDS

Reusing is defined as a way(s) to reduce resource usage and increase efficiency via deduplicating data or computations that share a certain level of similarity. Historically, reusing (*e.g.*, in form of caching) has been a fundamental approach to achieving software and hardware efficiency. In this section, we study reusing in the context of serverless computing and describe how it can be potentially advantageous for both cloud providers and users. Then, we explore a wide variety of techniques to carry out computational reuse in the serverless context. A summary of these techniques is shown in Figure 7.

4.1 | Deterministic Versus Semantic Reusing

Deterministic reusing refers to the set of techniques that can detect reusable computation or data in a definitive manner and perform the reusing without altering the results of the involved tasks. That is, these techniques do not require inferring the semantic similarity between two data or two computations. An example of deterministic reusing is when a user requests for the re-execution of a stateless video encoding function with identical specifications on the same video. As the stateless function output depends entirely on its input, a re-execution of a task executed with the same input arguments yields the same result. Thus, the video result of such task execution can be cached and reused. The majority of the deterministic reusing techniques operate based on detecting frequently-used computation and/or data, then caching them to be reused at a later time [104]. The cached data can be stored in different locations, such as platform-level storage [105], node-level storage [32], or inside the container in case durable containers are utilized (as described in Section 5.6.2). Accordingly, deterministic reusing techniques can be categorized into *data reusing* (a.k.a. caching) and *process reusing* that are elaborated in the next subsections.

Alternatively, *semantic reusing* aims to find a semantic relationship between similar (*i.e.*, non-identical) data [106] and perform reusing on them. In a sense, semantic reusing can be viewed as an approximation approach, due to

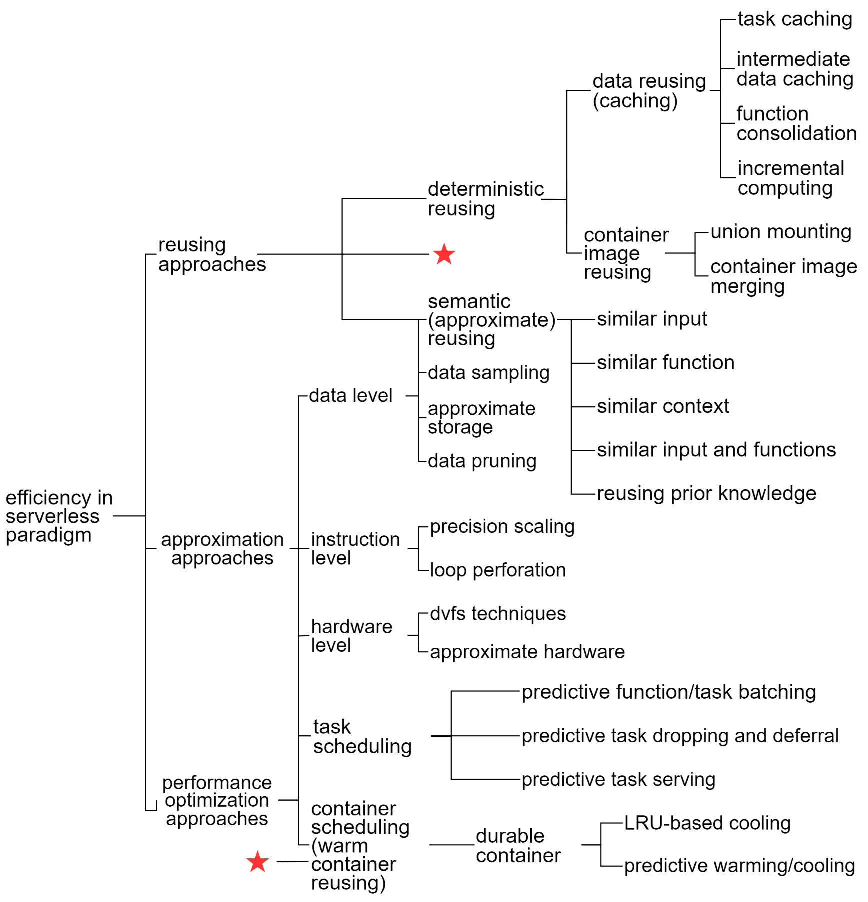

FIGURE 7 Taxonomy of approaches for efficiency in serverless computing platforms. The approaches are classified into two categories that are based on the *computational reuse*, and then based on *approximate computing*. There are approaches in the intersection of these two, known as *approximate reusing*.

similarity detection and the fact that the execution results of the involved tasks are not deterministic. However, the semantic similarity detection is uncertain and prone to misinterpretation, thus, can potentially lead to incorrect results.

An example of semantic reusing can be in an application that provides ambient perception for blind and visually impaired users by detecting obstacles in their environment (*e.g.*, [107,108]). These users often visit repeated locations and interact with objects they have previously encountered during their day-to-day activities. The captured pictures of these objects are *digitally* different because they are captured from different angles or under different light conditions. However, these pictures are *semantically* representing the same objects [94,109]. Accordingly, a reusing mechanism that can pre-process incoming images and detect whether a semantically similar one has been (or is being) processed can be helpful.

Further details about different approaches to performing semantic (approximate) reusing are discussed in Section 4.4

4.2 | Data Reusing

Data reusing is defined as the act of saving certain data to be reused at a later time. It is an integral part of different levels of modern computing systems—from the hardware level to the compiler and execution levels. In the particular case of serverless computing and FaaS, the fact that tasks are fine-grain (function level) and stateless provides an ideal opportunity for data reusing via saving (caching) and using the results of function execution again.

Data reusing is predominantly achieved via caching operation. Caching is an optional, but very popular, operation in the computing systems to mitigate the slowdown resulting from accessing the storage systems, hence, accelerating the task execution. That is, the system can still function correctly, even if it misses the cached data and retrieves it from the storage. Since caching is limited and often costly, establishing a trade-off between cost and efficiency in the caching scheme is of paramount importance. An extensive cache space imposes a significant cost, whereas, an inadequate cache space leads to missing reusable function results that, in turn, increase the re-computation cost and the response time [110, 111].

Based on the way data is stored and reused, the data reusing techniques fall into the following four categories that are elaborated in the next parts: (1) Task caching; (2) Intermediate data caching; (3) Function consolidation; and (4) Incremental computing solution.

Task caching

Task caching is the act of capturing and reusing the result of a task (function) execution. For stateless functions, this caching technique can be deployed transparently from the user's perspective. The cached data can be quickly identified by making use of the hash value of the function call signature that is composed of the function name and its arguments [112]. The cache table can be either shared across users (*i.e.*, public) or maintained separately for each user (*i.e.*, private). Unarguably, a public cache table maximizes the likelihood of data reusability across functions of all users. However, it can be vulnerable to cache poisoning attacks [113]. Alternatively, a private cache table offers better security via segregating the cache table either based on the user or the function [103].

Intermediate data caching

Intermediate data caching maintains partial results of the execution, rather than the final result. The technique is usually suited to a workflow with multiple computing steps, in which caching the intermediate result offers a higher chance of reusability than caching the final result [32, 33, 111]. For example, consider a two-stage calculation where the result of the first stage is fed to the second stage. If the first stage only has a few possible input parameters that take a long time to compute, and the second stage is fast, but has a large domain of possible input parameters, then, caching the result of the second stage yields a poor cache-hit, whereas, caching the result of the first stage (intermediate result) leads to a more reusable caching. In a serverless platform, intermediate result caching can be carried out via a key-value storage for the (stateful) functions [9]. To achieve reusability, this caching technique requires the function code to explicitly store and retrieve partial results from the caching system. Thus, it is not transparent from the user's perspective.

Function consolidation

Another variation of data reusing occurs in the function workflows [93, 114, 115] that includes multiple functions with data dependencies between them. In such workflows, because each function can be potentially allocated on a different machine, the overhead of transferring the output of one function to be used as the input of another function can be significant [114]. Such overhead can be mitigated by fusing functions. That is, two or more functions can be consolidated to form one function, such that the whole function is executed together and the data transfer overhead is eliminated. Let `A+B` represent a consolidated version of two functions `A` and `B`. In `A+B`, the output data of `A` can be reused by being directly fed as the input of `B`. We note that function consolidation has another benefit of scheduling one task as opposed to two individual tasks, hence, in Figure 7, it can be also considered under the "task scheduling" category. However, given the increasing popularity of function workflows in modern software engineering paradigms and the substantial overhead of data transfer across data centers, we believe that it better fits the "data reusing" category.

Function consolidation can be employed at the programming level via defining less granular functions, however, doing so is against the microservice-based software engineering methodology and makes the function maintainability cumbersome. A more efficient approach for function consolidation is to let users maintain their fine-grain functions and let a framework in the back-end automatically carry out the fusing [50,88,114] process without any user intervention. The main challenge in this approach is how to balance reducing the data transfer overhead against the resource inefficiency potentially caused by forming coarse-grain (consolidated) functions [114]. The reason for inefficiency (and potentially resource wastage) is that coarse-grain functions limit the ways tasks can be allocated, thereby, reducing the flexibility of resource allocation methods. For instance, consider a workflow with two chained functions where the first one, denoted $f_m()$, is memory-intensive and the second one denoted $f_c()$, is CPU-intensive. These functions can be efficiently allocated on different machines in a heterogeneous system—$f_m()$ on a memory-rich and $f_c()$ on a CPU-rich machine. However, consolidating the two functions requires a machine that is both memory- and CPU-rich. This can potentially lead to inefficient resource utilization or the incurred cost of accessing such a high-end machine can be more than the benefit of reducing the data transfer overhead. In summary, we note that an efficient function consolidation framework must consider the function's characteristics and bases its decisions on comprehensive function profiling [23].

Incremental computing

The fourth category of data reusing is the incremental computing technique [106]. Similar to task caching, this technique also caches the task result. However, the cached content is reusable beyond the tasks with identical input arguments. Incremental computing utilizes a *correction function* to adapt (*i.e.*, prune and expand) the cached results based on the new input. A common use case of incremental computing is in data analytics [106, 116, 117]. For instance, consider a repetitive function (query) that is regularly applied against a database with minor daily changes (*e.g.*, average number of active users in the past 365 days). A naïve way to handle the query is to thoroughly search the database every time. Alternatively, an incremental reusing technique retrieves the results of the prior period (*e.g.*, 365-days result calculated yesterday) and corrects them by pruning the invalid records (*e.g.*, data from 366 days ago) and adding new results (*e.g.*, data from today) via searching only within the updates in the database since the prior period. It is noteworthy that incremental reusing is a highly context-specific technique, and currently, it is not implemented within the general-purpose serverless platforms. For instance, Zhang *et al.* [118] propose a serverless and FaaS-based platform that takes advantage of incremental computing in the video analytics context. Their use case employs a deep neural network (DNN) model for video object classification. The model requires frequent updates to its weights to gain maximum accuracy with the ever-changing datasets. To avoid the excessive cost of re-training the model frequently, their platform deploys the incremental machine learning [119] technique to keep up with the gradual changes in the input datasets.

One way to enable incremental computing in future serverless platforms is to allow users to define multiple auxiliary functions, in addition to the main function. The auxiliary functions can include: a *subtract function* (to remove part of the existing results that are not valid for the input argument) and an *addition function* (to include new results to the existing ones and adapt them based on the new function arguments).

4.3 | Container Reusing

Apart from the data reusing, serverless efficiency can be gained via reusing at the sandboxing platform (*i.e.*, container) level. More specifically, container reusing can be carried out at the *container image* level that is described in the following subsection or at the *container instance* level that is described in Section 5.6.2.

4.3.1 | Container Image Reusing

Union mounting

Union mounting [120] is a well-established approach to reduce the container image footprint and, subsequently, the container start-up overhead. In union mounting, a container image is stored as multiple separate layers that can mount together to form an image. In this manner, the same layer can be utilized, *i.e.*, reused, in multiple images that share a module, thus, the storage overhead of container images is reduced. For instance, two machine learning functions can share the same operating system layer and the same machine learning framework (*e.g.*, TensorFlow [121]) layer. Then, they only differ on the application and model layers. Although deduplicating redundant layers is already widely used by the runtimes, further research works have recently been undertaken to improve the efficiency of deduplicating [100, 101] and to extend the deduplication idea to reuse *similar* layers [122]. Notably, Zhao *et al.* 's Duphunter [100] is a replacement layer loading module for the Docker platform. The architecture is more effective in deduplicating similar layers across multiple docker images with less deduplication overhead than prior designs.

Container image merging

When two or more functions whose container images have only minor differences are launched from a cold start, union mounting can capture and reuse several parts of their images. However, from the serverless platform perspective, these two are separate functions, hence, are treated independently. If these functions are infrequently invoked, they can get evicted from the memory. To encourage the system to keep these infrequently-used functions in the memory, a.k.a. *warm*, one approach is to merge these functions such that they share the same container image. In this case, the collective invocation frequency of these functions is increased that can avoid the memory eviction for them.

4.4 | Semantic (Approximate) Reusing

While deterministic reusing is only built on the identical data, semantic reusing aims to achieve reusing where the data are not digitally identical, but the base for reusing is some form of *semantic similarity* between the data. Semantic reusing can maintain semantic correctness by producing approximately the same results, while remarkably avoiding resource wastage and improving users' satisfaction. That is why this category of reusing is also considered a type of approximation that is discussed in Section 5. However, the mechanism to detect semantic similarity is not infallible and, similar to other approximation approaches, can potentially lead to inaccurate results that in certain contexts (*e.g.*, video streaming) could be tolerable and still useful. The semantic similarity detection and reusing functions are often application-specific. However, similarity detection is more efficiently performed at the scheduler level rather than the application (container) level. To enable wide adoption of semantic reusing, one potential way in the future serverless platforms is to provide a standardized API for users to define the semantic similarity detection for their function to be utilized by the platform scheduler.

In the serverless context, there are four types of similarities that can be leveraged for semantic reusing:

(1) Similar input:
For stateless functions, the result of execution depends only on the input argument. The input is often composed of multiple parameters. For instance, a video encoding function has the video segment, resolution, frame rate, bit rate, and codec as its input parameters. By designating certain parameters of the function as approximable, function reusability can be enhanced [94]. For instance, consider two users who call the `transcode` function to process the same video with two different (but approximable) resolutions. In this case, the system can approximate the resolution arguments and process the function once, instead of twice, and send the results to both users. Such scenarios can be particularly useful under certain circumstances, such as when the system is oversubscribed or when some users can tolerate lower QoS, such as free subscribers of a video streaming service.

(2) Similar function:
Following the microservice-based software development principles [4], generally, functions are developed to be short and single-purpose to ease the continuous deployment (CD) process. Therefore, it is likely that multiple users define similar functions that try to achieve the same purpose. These functions are semantically the same while having distinct source codes. Let functions A and B be semantically the same and x be an arbitrary input argument. In this case, the serverless computing system with the function similarity detection mechanism in place can reuse the result of $A(x)$ for $B(x)$ too. Moreover, since these functions are similar, one of the functions can be replaced by the other one, rather than keeping both functions available. This saves the number of functions that have to be kept active for rapid execution.

(3) Similar context:
In this category, the state, a.k.a. context, of a stateful function is to be reused. This is particularly helpful in the circumstances that the state data is not sensitive and can tolerate minor differences. That is, minor changes in the state do not significantly impact the results. For instance, consider a function for online learning of an image classification machine learning (ML) model [123] where the state data is the weights of the ML model. Other functions that intend to train the same model can reuse the state (weights) of the model from the earlier function, and train it further with their data. Such reusability makes the ML model converge faster and is more cost- and energy-efficient. A similar type of reusing can be considered in a serverless federated learning system [124] where the workers reuse a central model and train it further with their local data.

(4) Reusing prior knowledge:
In a serverless system, function (task) profiling data, collected and summarized from prior executions via automatic task profilers [125], can be supplied to the task scheduling module to maximize the resource allocation efficiency of the system [126]. Moreover, for a new user-defined function that lacks prior profiling information, the serverless platform can reuse prior knowledge of semantically similar functions to estimate the execution time of the new function on different machine types available in the system. Otherwise, lack of such information causes uninformed scheduling decisions that negatively impact the users' perceived QoS. Unlike other forms of semantic reusing that directly impact the quality of the results, reusing prior knowledge deals with the system parameters, such as utilization and QoS, and is used to improve them.

Transfer learning is a technique to reuse the knowledge gained while solving one problem and applying it to a different but related problem [127,128]. Accordingly, transfer learning can be employed to predict the execution time of a new function on a given machine type based on the trained networks of other functions on the same machine type. Moreover, methods can be explored to measure the semantic similarity between the new function and each one of the existing functions based on the dependencies and libraries shared between them. Then, the weighted average similarity of the new function with other functions can be used to estimate the execution time of the new function on that particular machine type. It is worth noting that, once enough execution time information for the new function is collected, a model specific to that task type can be trained to infer its execution time independent of other task types. Similarly, when a new machine type is added to a heterogeneous serverless system, the prior profiling information of functions on other machine types can be leveraged to estimate the expected execution time of the functions on the new machine type, thereby, utilizing it more efficiently.

5 | APPROXIMATE COMPUTING IN THE SERVERLESS CLOUDS

Approximate computing allows functions (tasks) with unaffordable response time, energy, or cost constraint(s) to be completed within its constraint(s) [129]. Even the tasks with affordable constraints that can tolerate approximate results, a.k.a. multi-fidelity tasks [130], can use approximate computing to bring about further resource-saving in the system. Since approximate computing compromises the precision and/or accuracy of the results, we envision that in comparison to reusing, employing approximation has less scope in the serverless platforms and is only to meet the tasks' constraints that are otherwise unattainable. Some notable use cases of approximate computing in the serverless systems include:

1) Improving the *response time* via fast approximate results, before confirming or correcting the results by the exact computing.
2) Providing an approximate result to reduce the incurred *cost*.
3) Providing an approximate result only if the system is *oversubscribed* and cannot perform exact computing in time.

There are various approaches for approximate computing that can improve the efficiency of serverless computing platforms. We can categorize these approaches into four classes, as shown in the lower part of Figure 7. In this section, we first position approximation approaches in comparison to the reusing-based approaches and then, discuss the general requirements for function approximation in the serverless context. Next, we elaborate on the four categories of approximation in Sections 5.3—5.6.

5.1 | Approximation Versus Reusing

The main difference between approximation and computational/data reusing is the impact on the result accuracy and precision. Computational reuse accelerates the turnaround time or saves the computing resource without influencing the task result. Conversely, approximate computing compromises the result accuracy and/or precision in favor of time- and/or resource-saving.

While approximate computing can be applied to tasks independently, many of the approximation techniques benefit from reusing information gathered from prior tasks. Such information can be either predefined ahead of time, *e.g.*, trained ML models, or collected and applied dynamically at the run time, *e.g.*, caching the results of similar tasks. Approximate computing techniques that directly reuse the result of other similar tasks are also known as approximate reusing which is discussed in the previous section.

5.2 | Approximate Computing Requirements

Approximate computing exploits the *resilient* property of the system by producing inexact but acceptable results at a lower cost. A resilient system [129] or application should be able to tolerate a certain amount of deviation from the ideal result [131]. Specifically, an approximate computing technique should not cause deviations that exceed the application resiliency in both error magnitude and likelihood.

Error magnitude

Error magnitude is defined and measured based on the variation of the obtained result from the ideal result. Applications' tolerance to the error magnitude varies based on the context. For example, video processing for live video conferences can tolerate a higher error magnitude than the video processing for traffic cameras that has to perform vehicle identification.

Error chance

Another dimension of error quantification is the likelihood of error occurrence. Formally, the likelihood of getting an overly inaccurate approximation is called the *error chance*. Certain approximation techniques (*e.g.*, DVFS [99]) produce mostly accurate results, however, there is a chance that the error occurred, and the result accuracy is off by far beyond the acceptable error magnitude. In such approximation techniques, upon detecting an error by a validation function, a correction function [132] is employed to fix the error retroactively. If the chance of getting an error is considerably high, then the overhead of correcting the results frequently can exceed the benefits of the approximation.

5.3 | Data-Level Approximation Approaches in Serverless Computing

5.3.1 | Approximate Reusing

Approximate reusing is the same as semantic reusing and is performed via identifying potentially reusable tasks and using their data to approximate other tasks [133]. Allowing repeated function calls to reuse the result of similar, but not strictly the same, tasks promotes reusability. The main challenge in approximate reusing is detecting the semantic task similarity. Applying this type of computational reuse can improve both the user and system metrics unless the users opt-out of it due to privacy concerns. The feasibility of this approach entirely depends on the feasibility of the semantic similarity detection system. As such, they are mostly applicable in domain-specific serverless systems, for example in video streaming [134].

5.3.2 | Data Sampling

For functions that work on a batch of data, such as data analysis works [30,106], it is possible to reduce the input data size by sampling from the dataset. Various techniques have been explored for data sampling, performing computation on the samples, and then analyzing the variability and the error rate in comparison to the complete data analysis on the whole data. For example in the IoT context, ApproxIoT [135] provides a method to sample data from a stream of unknown data sizes. Sampled data are stored in a size-limited reservoir. New data can randomly replace the existing ones in the reservoir. We believe this approach of approximate computing can be offered as an optional service for various special-purpose stateful serverless functions.

5.3.3 | Approximate Data Storage and Data Pruning

Approximate data storage can be achieved via either persisting only a portion of the data or scrambling multiple data points together. For instance, in a serverless multimedia cloud [8, 96] only base video formats can be persisted, and other less popular formats can be transcoded lazily—upon receiving a user request. For the content types that are error sensitive, similar data points can be stored together, *i.e.*, merged, via lossless data compression methods [136], whereas, for the content types that can tolerate minor errors, *e.g.*, multimedia and image, lossy compression methods [137] can be employed to approximate similar data, thereby, carry out a more effective compression.

Instead of involving the user in handling data storage and reusing, the serverless platform can be made aware of the services that store and deduplicate similar data [138]. By performing the data management at the platform level, the platform can maximize the likelihood of detecting similar data and performing deduplication. In such a storage mechanism, the process of retrieving data can also be approximated to reduce the data retrieval overhead. Eventual consistency [139] can be utilized on the data that is accessed by multiple tasks. Such a relaxed consistency control incurs a lower cost than a strict data consistency at the price of allowing the task to start with inconsistent data. Serverless functions are generally short-lived and are easily undoable. Therefore, tasks that start with inconsistent data can be canceled and restarted with minimal overhead.

5.4 | Instruction-Level Approximation in Serverless Computing

While instruction-level approximation techniques seem to be very context-specific and are applied at the application level, they can benefit from the metadata provided by the serverless platform. For example, the serverless platform can leverage resource utilization information to determine the normal or approximate processing of a given function.

5.4.1 | Precision Scaling and Stochastic Computing

The earliest forms of approximate computing were built by necessity in computer storage systems that could not store infinite decimal points [140]. Hence, the numbers had to be approximately stored and represented by a close-enough value. Then, the concept was further developed to a more deliberate dynamic precision scaling [141] where calculating the precision is scaled based on multiple factors, including the trade-off between computing precision and energy or turnaround time requirement.

Stochastic computing [142] is a popular collection of techniques to achieve precision scaling via representing a continuous value in form of a stream of bits. In this case, calculating the precision can be scaled by altering the number of bits in the bit stream. Making use of stochastic computing often implies designing a domain-specific processing unit (a.k.a. ASICs) hardware. Since precision scaling requires a specific framework and stochastic computing requires specialized hardware, these techniques have not yet gained wide adoption in the cloud computing industry. However, with the increasing prevalence of ASICs and, particularly, the trend in using precision scaling for the ML inference on the energy-limited edge devices [143], we envisage that these solutions will eventually carry over to the cloud systems too. As such, serverless solutions to support domain-specific processors that achieve instruction-level approximation will be demanded shortly to hide the complexity of deploying ASICs across the edge-cloud continuum from the user perspective. Such solutions will help the users to become solution-oriented and focus on their application logic, rather than dealing with the operational details of different hardware systems.

5.4.2 | Loop Perforation and Instruction Replacement

In a serverless computing platform, functions can either be provisioned as containers or, more popularly, as functional code blocks. The serverless platform can analyze the user code and optimizes them to save the computing resources. We believe that approximating frameworks such as Approxilyzer [144], proposed by Venkatagiri *et al.* can be deployed as an optional service to achieve such optimizations in serverless platforms. Approxilyzer analyzes the machine code of the function and dynamically replaces parts of the instruction with the approximated version. The aggressiveness of the approximation can be tuned based on the demanded quality, resiliency, and the approximation overhead.

5.5 | Hardware-Level Approximation In Serverless Computing

When heterogeneous computing is supported in a serverless computing system, specialized hardware that allows approximate computing at the hardware level can be offered as one of the resource types. The offering can be especially attractive for use cases, such as big data and machine learning, that are data-intensive and can benefit from domain-specific machines to offer low latency and real-time services [145]. Moreover, making use of specialized hardware to accomplish approximate computing can effectively reduce the energy consumption and footprint of cloud datacenters [145].

Two main hardware-level approximation approaches are namely, Dynamic Voltage Frequency Scaling (DVFS) [99] and approximator hardware. DVFS is a technique that strategically under-volt common hardware systems. Although such Undervoltage induces errors in the computation, applying it in a controlled manner, such that the error rate is tolerable by the applications, can improve the efficiency of the serverless system. For instance, Rahimi *et al.* [99] propose to strategically under-volt the GPU in favor of energy efficiency, while employing Hamming distance [146] to allow more error tolerance at the application level. On the approximator hardware side, certain computations are common and can greatly benefit from the approximator hardware. A few notable examples of such tasks include stochastic computing that can greatly accelerate computing by its approximator hardware [98], as explained in the previous section; DNN approximate inference that uses specially designed inference hardware [143]; and image encoding using the approximate encoder hardware [147].

The list of tasks that has approximator hardware support is still expanding, as more use cases are found to benefit from the approximator hardware and more tools to aid approximator hardware emerges [148]. Accordingly, we envision that the approximator hardware will find its way to the serverless systems in the future and their platform should be able to accommodate them transparently and efficiently.

5.6 | Scheduling-Level Approximation in Serverless Computing

5.6.1 | Task Scheduling Approaches

Predictive function (task) batching

Although the request turnaround time is one of the main criteria in measuring the performance of serverless clouds, not every application needs the function to complete as soon as possible. Moreover, even deadline-constraint tasks often can tolerate some delay, a.k.a. slack, before missing their deadlines. A recent study conducted by Eismann *et al.* [149] demonstrates that around 38% of their surveyed serverless applications have no latency requirement and another 28% of the applications have a few latency-sensitive functions. Only 2% of the applications are real-time with rigid latency constraints.

To maximize the efficiency of serverless systems, the user or the system should have a way to declare the task urgency, possibly in multiple tiers (*e.g.*, urgent vs non-urgent) or as a continuous number, such as a deadline value. In this case, highly urgent tasks can be scheduled to complete with the minimum turnaround time via warmed containers or by some form of approximation, whereas, the less urgent ones can potentially wait to aggregate with other similar arriving requests, thereby, maximizing the container reuse and reducing the incurred cost.

The scheduler must have the ability to predict the cost-benefit of delaying tasks in favor of batching them, such that each task waits as long as possible without missing its deadline to share the function container and other resources with other tasks. Predictive function batching is a viable technique that can be implemented in the existing serverless systems and multiple research works, though not directly targeting serverless cloud, can be applied to serverless as well. For instance, Grandslam [150] scheduler dynamically reorders tasks to maximize the task batching and minimize Service Level Objective violations in an oversubscribed system. Fifer [151] includes a scheduler with mechanisms to batch tasks and reduce the amount of container usage and cold start overhead within a given latency budget. Unlike Grandslam, Fifer tries to minimize the resource usage in an underutilized system, rather than trying to meet the tasks' deadlines in an oversubscribed system. The core idea of these two works can be applied to the serverless engine's task scheduler.

Task dropping and deferral

In a serverless system, each task request can be part of a bigger workflow. In some use cases, the workflow includes optional steps (tasks) whose loss can be tolerated [152]. Such a feature can be exploited at the scheduling level, particularly, to mitigate resource oversubscription [153] via dropping the optional tasks [154] or deferring their execution to a later time when the system is less busy [95, 153]. One use case that can take advantage of such workflow level approximation is in video conferencing where the voice quality enhancement task on the received video segments can be skipped, *i.e.*, dropped, to keep up with the liveness of the streamed video contents [8]. While this technique is feasible, it is currently not widely adopted on cloud platforms because: (a) dropping tasks intentionally can impact SLO compliance; and (b) on the cloud, it is commercially viable to scale out and cover the surge in demand, rather than dropping tasks to fit the resources. However, in an emergency, such as disaster recovery, or the resource- and energy-limited edge systems, task dropping and deferral techniques can be instrumental to improve the overall performance of the system. Due to the nature of user-defined and time-sensitive task types (with deadline), we believe that such desperate techniques are not yet practical to be offered on a general-purpose serverless framework.

Predictive task serving

The scheduler of a serverless computing platform can operate proactively and approximately anticipate the arrival of the user requests. For instance, Roy *et al.* [155] proposed an approach to predict the function calls via fitting the invocation trend to a statistical distribution. Such prediction can help the platform to pre-warm function containers and load their required data in speculation of upcoming task requests. Predictive task serving is done internally on the large-scale cloud providers [23]. However, the exact details on how they carry it out are often not publicly available.

5.6.2 | Container Scheduling Approaches

Durable container

Prior studies [23, 156] express that, in a serverless system, a certain percentage of functions are invoked frequently. If warm start execution of these functions incurs a significant loading overhead, then the frequent warm starts can compound a substantial overhead for the system. A common example of a function with high warm start overhead is the one that involves an online machine learning model as its state [157]. To solve such inefficiency, certain serverless computing platforms allow functions to run as a “durable container”, a.k.a. non-transient container, which means the container does not terminate after the task completion. These durable containers can maintain the state across function calls (*e.g.*, updating the machine learning model in the above example) that help to process subsequent tasks without paying repeated start-up overhead. In the event that there are multiple requests for the same function, the requests are queued to receive the service. Microsoft Durable Functions [72] and Oracle Fn Flow [84] support this capability. We envision that the future serverless platforms will auto-detect frequently-used functions and make their containers permanent without any user intervention.

Predictive warming and cooling

Due to the memory limitations of the servers, not all function containers can be maintained in the memory to rapidly start the functions’ execution. Infrequently used functions have to be offloaded to the storage to make room for the frequently-used ones. Such a memory contention across function containers is one of the challenges in the serverless domain and resolving it entails dealing with multiple problems: (1) how to reduce the cold start time overhead? (2) how to keep more containers in a given memory space? (3) how to minimize the number of cold starts through efficient memory allocation?

The main approach to mitigate the cold start overhead is to alter the transient nature of containers and keep them in the memory even after the function execution. Another approach is to proactively load the container, *i.e.*, before the function invocation [158]. In this direction, research has been conducted by Shahrad *et al.* [23] who studied 14 days of function invocation patterns in Azure cloud and leveraged that to develop a container memory allocation strategy. They propose to reduce the number of cold starts via categorizing the functions where each category has its pattern of *pre-warm* and *keep-alive* periods. Right after finishing an invocation, the function is removed from the memory for the *pre-warm* period, because the system does not expect to get another invocation of the function shortly. Then, once the pre-warm period expires, the container is loaded back into the memory (*i.e.*, warmed) to get ready for the next warm start function invocation. If the function stays in the memory for more than the keep-alive period without any invocation, then the function is removed from the memory. Such a strategy reduces the number of cold starts for the majority of the functions, however, there are still some functions whose invocation pattern is unpredictable, hence, cannot benefit from the predictive memory allocation. Nevertheless, the benefit of correct predictions and delivering a warm start remarkably exceeds the cold start misprediction plus the solution overhead.

6 | POTENTIAL FUTURE RESEARCH DIRECTIONS

Serverless computing and FaaS are considered as the second generation of cloud computing systems. As such, it is crucial to know the directions that which this area of distributed computing is evolving. In this section, we discuss the research directions that have the potential for further exploration and are deemed as the enablers of the next-generation cloud computing systems. Figure 8 categorizes and summarizes these research directions into the four following thrusts: (A) providing higher-level abstractions to streamline and accelerate serverless application development; (B) improving the performance of serverless systems; (C) extending the serverless paradigm to encompass the edge-to-cloud continuum; and (D) improving the security aspects of the serverless systems.

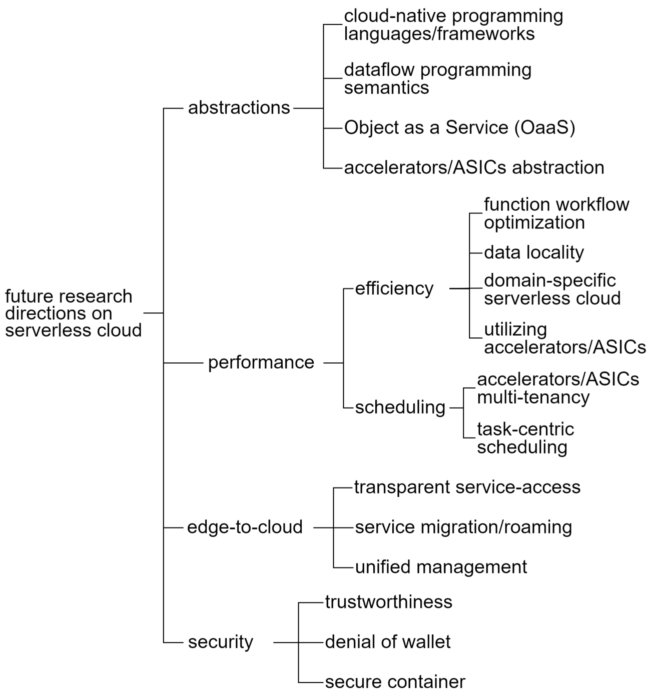
future research directions on serverless cloud
abstractions
cloud-native programming languages/frameworks
dataflow programming semantics
Object as a Service (OaaS)
accelerators/ASICs abstraction
performance
efficiency
function workflow optimization
data locality
domain-specific serverless cloud
utilizing accelerators/ASICs
scheduling
accelerators/ASICs multi-tenancy
task-centric scheduling
edge-to-cloud
transparent service-access
service migration/roaming
unified management
security
trustworthiness
denial of wallet
secure container

FIGURE 8 Topics for future research directions on the serverless cloud.

6.1 | High-level Abstractions for Serverless Computing

6.1.1 | Cloud-Native Programming Language/Framework

As the second generation of cloud technology is evolving based on the serverless paradigm and mitigating the programmers' job, we envisage that the technology is approaching the *cloud-native programming languages* era. In such programming languages, FaaS concepts will be integrated into the programming languages and compilers can natively support them. Like that, functions can be defined as `cloudable` and seamlessly deployed on the cloud by the compiler or auto-migrates to the cloud by the run-time system to overcome the resource shortage or to achieve reusing and approximation. Moreover, these languages can integrate the "programming code" with the "deployment code", a.k.a. infrastructure in code [159], that are currently developed separately by different people. In this manner, the program is analyzed to automatically identify the infrastructure demands and deploy them on the cloud, such that the likelihood of reusing and approximation are maximized.

These features of the forthcoming cloud-native programming languages will democratize cloud programming, such that people without cloud knowledge can utilize them in their programs.

6.1.2 | Dataflow Programming Semantics

Although Function-as-a-Service (FaaS) abstraction relieves users from the burden of resource management (*e.g.*, load balancing and elasticity), it is not truly serverless, because it falls short on abstracting data management and the users still have to get involved with other services, such as AWS DynamoDB [160] or AWS SAM [161], to serve desires of the functions and/or applications [30, 34, 162]. In particular, in some use cases, the data can represent the function state. For instance, the state of an online-learned Machine Learning (ML) model is updated frequently. Embedding such a model into the function container image is not practical. Furthermore, ML models are large enough that cannot be fed as the function arguments, otherwise, the function startup overhead would become substantial.

In this scenario, the ML model is the function's state that is best to be stored in a synchronized storage service that can be accessed seamlessly upon the developers' demand. In this manner, the developers do not need to think about scattering, reusing and scalability of the state data on various storage or database services, instead, they can focus on the business logic of the application while the system automatically manages data reusing and caching as it sees fits. We believe that future serverless platforms will accommodate such dataflow semantics.

6.1.3 | Object-as-a-Service (OaaS): Going Beyond the Function Abstraction

Current serverless and FaaS solutions are not designed for (and cannot natively support) data-centric applications or *state data*. The developers have to intervene and undergo the burden of managing the application data using separate cloud services, *e.g.*, AWS RDS [163], to persist the state information. Apart from the data aspect, current FaaS systems do not offer any built-in semantics to limit the access to the internal, a.k.a. private, mechanics of the functions. However, providing unrestricted access to the developer team has known side effects, such as function invocation in an unintended context, and data corruption via direct data manipulation. To overcome such side effects, developers again need to intervene and undergo the burden of configuring external services, such as AWS IAM [164] and API gateway [165], to enable access control. This makes the development and maintenance of serverless applications difficult and cumbersome. To ease the management of data and accelerate the development of new services for them, a higher level of abstraction is desired that, in addition to hiding the resource allocation details, it can hide the details of access control and preserve its state from the users.

To natively support serverless data-centric application development, we envisage that in the future serverless platforms, the concept of *object* can be borrowed from the Object-Oriented Programming, as the first-class citizen to encapsulate both computing (functions) and state (data) within a single object entity, and offer the notion of *Object-as-a-Service* (OaaS) [10]. OaaS will be the next generation of the BaaS part of the serverless system that not only will handle the state management and persistence with minor user intervention, but also will offer a high level of abstraction to the user.

Not only do objects in OaaS offer encapsulation and abstraction benefits on top of the function abstraction, but they also unlock opportunities for built-in optimization features, such as data locality, data reliability, caching software reusability, and data access control. For instance, a serverless platform can offer the built-in encapsulation semantics for a cloud-based video streaming system [134]. In this system, video content is defined as an object that has the video file as its state and is bound to a set of functions that can be invoked by the viewer's application that can potentially change the object state. A few examples of such services are as follows: Generating multilingual subtitles for the safety-related videos; Removing harmful and illicit content from the child-safe video content.

As mentioned in Section 2.3, some stateful serverless solutions are implemented using the actor model. However, the major drawback of the actor model, considering no further optimization, is the limitation of reusing. OaaS can overcome this problem by leveraging the immutable data processing model. That is, upon invoking a function of an object, the OaaS platform outputs a new/updated object state, instead of updating the existing one. Implementing this semantic makes the function perform a stateless operation and enables the ability to apply approaches of reusing and approximation.

6.1.4 | Accelerators/ASICs Abstraction

GPU computing gained a massive uptake as Nvidia continuously enhances GPU program-ability by abstracting hardware coding into user-friendly functions in CUDA. The abstraction makes it feasible to port CPU computing only code into GPU accelerated code. Beyond GPU programming via GPU-specific code, is it possible to abstract serverless functions into hardware-agnostic code so that the framework can determine and utilize the appropriate hardware accelerator? For example, a large set of data can be processed one by one in the loop using mobile CPU, processed in batch using GPU, or approximately processed using approximator ASICs depending on what accelerator is available. Such capability can go a long way in popularizing heterogeneous computing in serverless clouds.

6.2 | Performance Aspects in Serverless

6.2.1 | Memory Contention

Unlike conventional cloud computing services (*e.g.*, IaaS services) where users are in charge of explicitly running and terminating the services, in serverless computing, the platform automatically allocates resources and runs the services upon request, and then de-allocates them when they are not needed anymore. This automated allocation and de-allocation of resources are realized via transient isolation platforms, *e.g.*, containers, that also enable charging users only for the times the services and functions are being used. Ideally, the containers should be loaded from cold storage just once and then maintained in memory to guarantee fast execution of the functions, which is critical for latency-sensitive applications [166]. However, in a large-scale serverless cloud, maintaining all the functions in memory is impractical, owing to both hardware and economic limitations. Hence, there is a memory contention between containers for memory access. Efficiently resolving this contention and determining the functions that should remain in memory to maximize function reusing is a challenging problem for the BaaS part of serverless systems.

6.2.2 | Function Workflow Optimization

As mentioned in Section 2.2, serverless functions can be in form of individual functions or as a workflow. Users can define a function workflow as multiple individual functions whose completion triggers the next function inline or define function workflow using a certain schema format. Currently, there are no platform-agnostic schema standards that we believe can be a useful development to maximize cross-platform compatibility, and streamlines cross-platform workflow migrations.

From the scheduling aspect, big data analytics and map-reduce workflows are observed to perform poorly on serverless systems [30]. The reason is that the serverless task schedulers are generally not tuned for workflow tasks. Functions with a large memory footprint are loaded to Perform a small task and then released, rather than being reused for other batches of data. Serverless tasks are typically expected to have a short start-up time, hence, their schedulers are designed to be simple and lightweight. However, the next-generation serverless platforms are expected to consider particular scheduling arrangements for the function workflows. One approach can be scheduling the first tasks of the workflow upon arrival with the minimum startup overhead; for the following tasks, the scheduler evaluates their data dependency with the other tasks and assigns them such that the data transfer overhead is minimized and function reusing is maximized.

6.2.3 | Data Locality Optimization

Data locality optimization can reduce the wasted bandwidth and computing resources by minimizing the inter-rack data transfer to the minimum within the serverless cloud datacenter. Such optimization is critical for effective data reusing. In a poorly optimized system, the data transfer latency of the reusable data may outweigh the time required to recompute such data without reusing. An example of such action is to place containers that frequently communicate within the same machine or rack, or schedule the tasks to be close to the data source.

While data locality is carried out by the user in the conventional cloud service models, it is the responsibility of the BaaS in the serverless paradigm. This is because the serverless aims at abstracting the user from the underlying resource management details and, as part of it, the user should not micromanage data locality. As the user grants full control of the process and data location to the cloud, the serverless platform has the luxury of optimally allocating tasks and data free from the user's constraints. For this reason, it is expected that the future serverless schedulers will leverage this flexibility to maximize data locality, thereby, reducing their cost and energy consumption.

6.3 | Efficiency via Domain-Specific Serverless Cloud Computing

Along with the rise of domain-specific computing and ASIC hardware, domain-specific programming languages are also emerging for popular applications, such as machine learning, cryptography, multimedia processing, and even fluid dynamics [167]. Within this trend, we envision that the next step will be the emergence of domain-specific serverless cloud platforms for popular applications. A domain-specific serverless system will be equipped with specialized hardware (ASICs), support of special-purpose programming languages, and built-in domain-specific functions in its repository. Such platforms will expedite the application development process and shorten the CI/CD cycles. They will help users become solution-oriented and focus on their specific business logic, rather than spending time on developing basic services. In addition, they can support more flexible and application-specific billing schemes (*e.g.*, per successful transaction completion) with even cheaper prices, due to using more efficient specialized hardware.

Importantly, a domain-specific serverless platform creates new scopes for serverless efficiency, via specialized function reusing and approximation techniques. For instance, a multimedia serverless cloud platform can harness its built-in knowledge of spatiotemporal multimedia streaming patterns to upraise the likelihood of function reusing.

6.3.1 | Utilizing Accelerator in Serverless Cloud

Serverless computing systems were originally pioneered to create an illusion that there is no server to manage. Fulfilling this aim is viable when the underlying computing resources are either homogeneous or of the same architecture, a.k.a. consistently heterogeneous [153]. As such, to date, commercial serverless computing providers support only various forms of CPU-based machines that are auto-provisioned by the BaaS subsystem to match the function desires [168]. However, to accommodate more flexibility and efficiency in FaaS and serverless computing, inconsistently heterogeneous systems with a combination of CPU, GPU, TPU, FPGA, and other forms of emerging ASICs [89, 168] (including hardware specifically designed for approximate computing) are desired. With the slow-down in general-purpose computing and the rise of specialized computing [169], we expect that application-specific hardware becomes prevalent in serverless systems in the near future [168].

Provided that the user functions can utilize heterogeneous resources through virtualization or heterogeneous-supported frameworks (*e.g.*, TensorFlow for machine learning tasks, FFmpeg [170, 171] for multimedia processing tasks), the challenge is how to schedule and provision such functions on the heterogeneous resources efficiently.

While various forms of heterogeneous-aware task scheduler already exist [95,142,153] in the HPC context, serverless tasks are often of finer granularity. Therefore, a lightweight and low-latency scheduling that is aware of the machine heterogeneity is desired.

To make informed scheduling decisions in a heterogeneous system, a function execution-time profiler is needed to provide an estimated execution time of a given function on the heterogeneous machine types [142]. Since each serverless function recurs multiple times, execution-time profiling can be carried out [23,172]. However, to reduce the uncertainty in the execution-time prediction, thereby, making more informed scheduling decisions, on different machine types, function profiling must be performed proactively and in an explorative way to examine the infrequently-used options and collect their execution time statistics. Importantly, scheduling new user-defined functions, for which there is no prior execution-time statistics, is more challenging. Methods based on Transfer Learning [157] (*i.e.*, reusing prior learning to speed up learning a new data set) should be developed to enable inferring the execution time of the new function based on other existing functions on different machine types. A similar challenge and solution can be posed upon the addition of new machine types to the serverless system.

6.4 | Performance Aspects in Serverless

6.4.1 | Accelerator Multi-tenancy in Serverless Systems

Accelerator hardware is usually designed to be utilized by one tenant at a time. Even GPU is not originally designed to support multi-tenancy, although frameworks like Volta-MPS have added the multi-tenancy support to the recent models of GPU. Due to a serverless payment scheme where users do not pay for under-utilized hardware, it is cost-prohibitive to allocate the entire accelerator hardware to a single function. Maximizing the accelerator utilization implies deploying multiple functions sharing the same hardware while allowing authorized data sharing but not leaking sensitive data. This is a logical development needed to enable economical accelerator deployment in serverless systems [173].

6.4.2 | Task-Centric Scheduling

In the current commercial serverless computing platforms, functions are triggered without a specified deadline or urgency levels, thus, the scheduler treats tasks with equal priorities. Meanwhile, if task urgency and their demands are provided to the serverless platform scheduler, the task with low urgency can be executed in batch in favor of more serverless efficiency (see container reusing in Section 4.3). Such urgency and demands for information are also helpful for the caching and other components to determine their operational priorities. Moreover, by detecting infeasible deadlines, the system can then promptly approximate the task, rather than missing its deadline because of exact computation. We envision a scheduler that looks into each task's QoS demand and then schedules them appropriately. However, requiring the user to profile their task's demand diminished the simplicity of the serverless platform. Another avenue of exploration is to identify the task urgency and their demands automatically at the platform level and transparently from the user's perspective.

6.5 | Edge-to-Cloud Serverless Platforms

6.5.1 | Seamless Function Call & Service Migration

The main benefits of serverless computing are the ease of use and abstracting of the users from the resource provisioning details. While the current serverless solutions are limited to cloud systems, it is desired to extend their benefits to the emerging edge-to-cloud systems and hide the complexity of dealing with multiple computing tiers (*i.e.*, device, edge, fog, and cloud tiers) from the user perspective. The platform can transparently determine the appropriate tier for the function execution and can seamlessly migrate the execution from one tier to another to overcome the inherent resource scalability problem of the edge systems [174]. These abilities can improve the system efficiency and unlock new use cases. For instance, consider the use case of a blind person who uses smart glasses and needs real-time processing of the observed objects to enter a hotel lobby where few people are playing low-latency online games using the available edge system. Upon the arrival of the blind person, to make room for the blind application, the game functions have to be live-migrated to the cloud, so that the gaming is not interrupted. The opposite can happen when the blind person leaves the place. All these take place while the users have an illusion of everything being executed on their own devices.

A serverless platform for the edge-to-cloud continuum extends the scope for computational reusability and approximation to circumstances where a task result that is already cached in another tier can be fetched from there, instead of executing it locally [175]. Another interesting reusing potential that can be unleashed is in a scenario where edge devices forward common (reusable) tasks to a central cloud, so that other edge tiers can reuse them by fetching them from the cloud [176].

6.5.2 | Unified Management

From the service providers' perspective, it is a daunting task to manage the resources across cloud, fog, edge, and the device. Multi-tier applications are usually orchestrated to perform different duties on different tasks. For example, user devices take care of the UI/IO tasks, edge devices aggregate local data and then cloud machines do the heavy computing part. Such different tasks generally are written specifically to the hardware-specific tier.

What if all the applications—from a mobile device, edge to cloud devices—become serverless functions that run on the same standardized serverless framework that connects to the unified mean pane? Then, the function development, deployment, and migration can be done seamlessly within a single point of control. In addition, rather than each multi-tier system deploying its own edge devices separately, multiple multi-tier systems can also share the edge to cloud resources and data, since they all run on the same standardized framework. Such an idea can also be viewed as a CDN that not only provides static content at the edge but also allow serverless function executions on any local edge and fog server.

6.6 | Serverless Security and Trustworthiness

6.6.1 | Trustworthiness

Security is one of the criteria for developers in selecting a serverless platform [177]. With numerous components and the nature of function triggering, serverless computing exposes a large attack surface compared to its predecessors [178]. Poorly-designed functions meant for internal purposes often lack authentication and can be attacked via a direct triggering or an injection attack [179]. Meanwhile, the fact that serverless functions are stateless and short-lived limits the time available to attackers and the impact of successful attacks. This shifts the attack strategies to become shorter and more indirect.

Cloud providers offer the shared responsibility model to the customers that make the cloud providers responsible for the infrastructure security and continuously applying security updates to the low-level software, hence, relieving some security burdens from the developers. However, this leads to one common problem in cloud computing security: the trustworthiness of cloud providers [180]. In the serverless paradigm, the trustworthiness issue is extended to cover several aspects: From the performance aspect, the developer cannot assure if the serverless platform achieves the required QoS, such as the number of available containers in the shared pool that are ready to serve the function's triggers; From the isolation aspect, the developers cannot understand how the underlying isolation frameworks (VM/container) are configured for the functions. In particular, in serverless systems, the VMs can be potentially shared with other cloud users, due to the small function sizes. This increases the risk of being attacked. Some serverless platforms [181] offer built-in CI/CD pipelines to let developers conveniently build their source code to the function. This feature exposes other attack vectors, because the building steps may contain vulnerabilities (*e.g.*, the vulnerability of the third-party library) pulled by the platform to build the function image. The topic of platform trustworthiness is highly critical to the adoption rate of data reusing techniques across users where the risk of data leak or compromise is a concern.

6.6.2 | Secure Container

As mentioned earlier, one of the security concerns in the serverless paradigm is the lack of isolation. Containers expose vulnerabilities, such as privilege escalation from sharing the host kernel. To address the problem, the developers have choices to use rootless containers, or isolated containers [182] (*e.g.*, gVisor [183] or Kata Containers [184]). We note that, in practice, there should still be a trade-off between security and efficiency. The secure container offers high security but relatively low startup latency. On the other hand, relaxing isolation opens up more optimization opportunities [185]. Section 5.6.2 discusses container instance reusing, which is an example of such relaxation.

6.6.3 | Denial of Wallet

In the area of performance, serverless computing tolerates denial-of-service attacks more than its predecessors, because of its inherent ability to scale. However, the ability to scale brings a new possibility for attackers to perform Denial-of-Wallet attacks [186], an attack that forces the financial exhaustion of the application's owner instead of disabling the service availability. When a service function is being targetted via a Denial-of-Wallet attack, the serverless platform steps in and triage the running cost down via enacting dropping and deferring low-importance tasks (see Section 5.6) to preserve the user's funding and to keep the more important tasks running.

7 | SUMMARY

It is envisioned that the future generation of highly scalable applications will predominantly rely on the serverless computing paradigm, hence, comprehensively studying the anatomy of this paradigm and identifying the scopes for efficiency can bring about major benefits to the users, developers, and providers. Accordingly, in this paper, we explored the ways to make the serverless computing paradigm efficient via investigating two main thrusts, namely computational reuse, and approximate computing. We started by characterizing the internal mechanics and studying different dimensions of serverless systems. Then, we surveyed the current state of the serverless and FaaS solutions (summarized in Table 3). Next, we categorized various approaches of reusing and approximation, respectively. An overview of these approaches is shown in Figure 7. In sum, we state that the characteristics of the serverless paradigm, where functions are compact, single-purpose, and portable, create a unique scope for efficiency, mostly via computational reuse, and then via approximate computing.

In this paper, we also outlined several potential directions (summarized in Figure 8) that can push the envelope of the serverless paradigm toward the next generation of cloud computing systems. In summary, four prominent directions that we discussed as the future of the serverless paradigm are as follows: (A) Enabling higher-level abstractions, such as Object-as-a-Service and cloud-native programming languages, within the serverless paradigm; (B) Improving the performance of serverless clouds via exploiting the extensive function reusing and approximation opportunities exist in these systems; (C) Extending the serverless platforms beyond cloud infrastructure to cover multi-tier edge-to-cloud continuum, and (D) Extending the serverless abilities towards cloud-native serverless security.

Acknowledgments

We would like to thank the anonymous reviewers of the paper, and members of the HPCC Lab, particularly Davood G. Samani and Pawissanutt Lertpongrujikorn, who brainstormed with us on this paper. This research is supported by the National Science Foundation (NSF) under awards# CNS-2047144 and CNS-2007209.

References

[1] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[2] P. G. Lopez, A. Slominski, M. Behrendt, and B. Metzler, "Serverless predictions: 2021-2030," *arXiv preprint arXiv:2104.03075*, 2021.

[3] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016.

[4] S. Arachchi and I. Perera, "Continuous integration and continuous delivery pipeline automation for agile software project management," in *Proceedings of the Moratuwa Engineering Research Conference*, ser. MERCon '18. IEEE, May 2018, pp. 156–161.

[5] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings of the IEEE International Conference on Cloud Engineering*, ser. IC2E '18. IEEE, Apr. 2018, pp. 159–169.

[6] Amazon, "Amazon Athena - Serverless Interactive Query Service - Amazon Web Services," online; Accessed on 4 May 2022. [Online]. Available: https://aws.amazon.com/athena/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc

[7] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proceedings of the 17th usenix symposium on networked systems design and implementation*, ser. nsdi '20, Aug. 2020, pp. 419–434.

[8] X. Li, M. A. Salehi, M. Bayoumi, N.-F. Tzeng, and R. Buyya, "Cost-efficient and robust on-demand video stream transcoding using heterogeneous cloud services," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 3, pp. 556–571, Mar. 2018.

[9] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[10] P. Lertpongrujikorn and M. Amini Salehi, "Object as a service (oaas): Enabling object abstraction in serverless clouds," in *Proceedings of the 16th IEEE Cloud Conference*, Jul. 2023.

[11] M. Nazari, S. Goodarzy, E. Keller, E. Rozner, and S. Mishra, "Optimizing and extending serverless platforms: A survey," in *Proceedings of the Eighth International Conference on Software Defined Systems (SDS)*. IEEE, 2021, pp. 1–8.

[12] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, 2021.

[13] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, 2021.

[14] A. Raza, I. Matta, N. Akhtar, V. Kalavari, and V. Isahagian, "Function-as-a-service: From an application developer's perspective," *JSys*, vol. 1, no. 1, pp. 1–20, 2021.

[15] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[16] Y. Li, Y. Lin, Y. Wang, K. Ye, and C.-Z. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, 2022.

[17] A. Chakraborty, M. Kumar, N. Chaurasia, and S. S. Gill, "Journey from cloud of things to fog of things: Survey, new trends, and research directions," *Software: Practice and Experience*, vol. 53, no. 2, pp. 496–551, 2023.

[18] C. N. C. Foundation, "CNCF Annual Survey 20212," online; Accessed on 14 Mar. 2023. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/

[19] Datadog, "The State of Serverless," online; Accessed on 14 Mar. 2023. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[20] IBM, "Serverless in the Enterprise, 2021," online; Accessed on 6 Apr. 2022. [Online]. Available: https://www.ibm.com/downloads/cas/ZJLWQOAQ

[21] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, Dec. 2020.

[22] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '18, Jul. 2018, pp. 133–146.

[23] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '20. USENIX Association, Jul. 2020, pp. 205–218.

[24] serverlessworkflow.io, "serverlessworkflow/specification," online; Accessed on 21 Mar. 2023. [Online]. Available: https://github.com/serverlessworkflow/specification/blob/main/specification.md#Workflow-Definition-Structure

[25] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1591–1604, 2022.

[26] R. S. Bird and P. L. Wadler, *Functional Programming*. Prentice Hall, 1988.

[27] M. B. Josephs, "Functional programming with side-effects," *Science of computer programming*, vol. 7, pp. 279–296, 1986.

[28] S. Zobaed, M. E. Haque, M. F. Rabby, and M. A. Salehi, "Senspick: Sense picking for word sense disambiguation," in *Proceedings of the 15th IEEE International Conference on Semantic Computing (ICSC)*. IEEE, Jan. 2021, pp. 318–324.

[29] A. Jain, A. F. Baarzi, N. Alfares, G. Kesidis, B. Urgaonkar, and M. Kandemir, "Spiitserve: Efficiently splitting complex workloads across faas and iaas," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, Nov. 2019, p. 487.

[30] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '19, Feb. 2019, pp. 193–206.

[31] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1591–1604, 2022.

[32] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2438—-2452, Jul. 2020.

[33] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *The VLDB Journal*, pp. 1–19, Sep. 2020.

[34] J. Schleier-Smith, L. Holz, N. Pemberton, and J. M. Hellerstein, "A faas file system for serverless computing," *arXiv preprint arXiv:2009.09845*, 2020.

[35] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas, "Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1289–1302, Sep. 2018.

[36] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proceedings of the 15th European Conference on Computer Systems*, Apr. 2020, pp. 1–15.

[37] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '20. USENIX Association, Jul. 2020, pp. 419–433.

[38] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia *et al.*, "Apiary: A dbms-backed transactional function-as-a-service framework," *arXiv preprint arXiv:2208.13068*, 2022.

[39] C. Hewitt, "Actor model of computation: Scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010.

[40] Microsoft, "Durable entities - Azure Functions | Microsoft Docs," online; Accessed on 4 May 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp

[41] L. Inc., "High Performance Microservices and APIs | Kalix.io," online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.kalix.io

[42] Apache, "Apache Flink: Stateful Computations over Data Streams," online; Accessed on 4 May 2022. [Online]. Available: https://flink.apache.org/

[43] G. McGrath and P. R. Brenner, “Serverless computing: Design, implementation, and performance,” in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops*, ser. ICDCSW ’17. IEEE, Jun. 2017, pp. 405–410.

[44] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, “Serverless computing for container-based architectures,” *Future Generation Computer Systems*, vol. 83, pp. 50–59, Jun. 2018.

[45] V. Kjorveziroski, S. Filiposka, and A. Mishev, “Evaluating webassembly for orchestrated deployment of serverless functions,” in *Proceedings of the 30th Telecommunications Forum (TELFOR).* IEEE, 2022, pp. 1–4.

[46] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, “From warm to hot starts: Leveraging runtimes for the serverless era,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 58–64.

[47] D. G. Samani and M. Amini Salehi, “Exploring the impact of virtualization on the usability of the deep learning applications,” in *Proceedings of the 22th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, ser. CCGrid ’22, May 2022.

[48] krustlet.dev, “Kubernetes Rust Kubelet,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://github.com/krustlet/krustlet

[49] G. Sayfan, *Mastering Kubernetes: Level up Your Container Orchestration Skills with Kubernetes to Build, Run, Secure, and Observe Large-Scale Distributed Apps*. Packt Publishing Ltd, 2020.

[50] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[51] R. Bruno, S. Ivanenko, S. Wang, J. Stevanovic, and V. Jovanovic, “Graalvisor: Virtualized polyglot runtime for serverless applications,” *arXiv preprint arXiv:2212.10131*, 2022.

[52] C. Wimmer and T. Würthinger, “Truffle: A self-optimizing runtime system,” in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 13–14.

[53] S. Wu, Z. Tao, H. Fan, Z. Huang, X. Zhang, H. Jin, C. Yu, and C. Cao, “Container lifecycle-aware scheduling for serverless computing,” *Software: Practice and Experience*, vol. 52, no. 2, pp. 337–352, 2022.

[54] O. C. Initiative, “About the Open Container Initiative,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://opencontainers.org/about/overview/

[55] M. Amini Salehi, A. N. Toosi, and R. Buyya, “Contention management in federated virtualized distributed systems: implementation and evaluation,” *Software: Practice and Experience*, vol. 44, no. 3, pp. 353–368, 2014.

[56] D. Ghatrehsamani, C. Denninnart, J. Bacik, and M. Amini Salehi, “The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms,” in *Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP ’20, Aug. 2020, pp. 1–11.

[57] D. Kumar and A. F. F. Magloire, “Hypervisor based performance characterization: Xen/kvm,” April 2018.

[58] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46, 2005, pp. 10–5555.

[59] M. Zhang, C. Krintz, and R. Wolski, “Edge-adaptable serverless acceleration for machine learning internet of things applications,” *Software: Practice and Experience*, vol. 51, no. 9, pp. 1852–1867, 2021.

[60] G. Gain, C. Soldani, F. Huici, and L. Mathy, “Want more unikernels? inflate them!” in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 510–525.

[61] fission.io, “fission-workflow,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://github.com/fission/fission-workflows

[62] Bitnami, “Kubeless,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://github.com/vmware-archive/kubeless

[63] Iron.io, “Iron.io - DevOps Sp;itopm from Startups to Enterprise,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.iron.io/

[64] D. Jackson and G. Clynch, “An investigation of the impact of language runtime on the performance and cost of serverless functions,” in *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion).* IEEE, 2018, pp. 154–160.

[65] D. Bortolini and R. R. Obelheiro, “Investigating performance and cost in function-as-a-service platforms,” in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 14th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2019).* Springer, 2019, pp. 174–185.

[66] Amazon, “AWS Lambda - Serverless Compute - Amazon Web Services,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://aws.amazon.com/lambda/

[67] ——, “AWS Step Functions| Serverless Microservice Orchestration | Amazon Web Services,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://aws.amazon.com/step-functions

[68] ——, “Serverless Compute Engine - AWS Fargate- Amazon Web Services,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://aws.amazon.com/fargate/

[69] Microsoft, “Azure Functions Serverless Compute | Microsoft Azure,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[70] G. Cloud, “Cloud Functions | Google Cloud,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://cloud.google.com/functions/

[71] IBM, “IBM Cloud Functions,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.ibm.com/cl oud/functions

[72] Microsoft, “Durable Functions Overview - Azure | Microsoft Docs,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp

[73] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, “Analyzing open-source serverless platforms: Characteristics and performance,” *arXiv preprint arXiv:2106.03601*, 2021.

[74] A. Ellis, “OpenFaaS – Serverless Functions Made Simple,” Online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.openfaas.com/

[75] A. S. Foundation, “Apache OpenWhisk is a Serverless, Open Source Cloud Platform,” Online; Accessed on 21 Mar. 2023. [Online]. Available: https://openwhisk.apache.org/

[76] D. Inc., “Docker Registry | Docker Documentation,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://docs.docker.com/registry/

[77] Apache, “Apache Foundation,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.apache.org/

[78] ——, “Apache Kafka,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://kafka.apache.org/

[79] ——, “Apache CouchDB,” http://couchdb.apache.org/, online; Accessed on 29 Aug. 2021.

[80] I. F5, “NGINX | High Performance Load Balancerm Web Serverm & Reverse Proxy,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.nginx.com/

[81] Apache, “Apache Zookeeper,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://zookeeper.apac he.org/

[82] fission.io, "Fission: Serverless Functions for Kubernetes," online; Accessed on 29 Aug. 2021. [Online]. Available: https://github.com/fission/fission

[83] Oracle, "Fn Project - The Container Native Serverless Framework," online; Accessed on 21 Mar. 2023. [Online]. Available: https://fnproject.io/

[84] ——, "Fn Flow," online; Accessed on 21 Mar. 2023. [Online]. Available: https://github.com/fnproject/flow

[85] C. N. Foundation, "Knative," online; Accessed on 08 Oct. 2021. [Online]. Available: knative.dev

[86] C. N. C. Foundation, "The Cloud Native Computing Foundation (CNCF) Survey," online; Accessed on 21 Mar. 2023. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

[87] knix.io, "KNIX MicroFunctions," online; Accessed on 21 Mar. 2023. [Online]. Available: https://github.com/knix-microfunctions/knix

[88] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: Towards {High-Performance} serverless computing," in *Proceedings of the Usenix Annual Technical Conference*, ser. USENIX ATC '18, 2018, pp. 923–935.

[89] nuclio.io, "Nuclio: Serverless Platform for Automated Data Science," online; Accessed on 21 Mar. 2023. [Online]. Available: https://nuclio.io/

[90] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo, "Dfaas: Decentralized function-as-a-service for federated edge computing," in *Proceedings of the 10th IEEE International Conference on Cloud Networking (CloudNet)*. IEEE, 2021, pp. 1–4.

[91] S. Ghaemi, H. Khazaei, and P. Musilek, "Chainfaas: An open blockchain-based serverless platform," *IEEE Access*, vol. 8, pp. 131 760–131 778, 2020.

[92] Z. Li, R. Chard, Y. Babuji, B. Galewsky, T. J. Skluzacek, K. Nagaitsev, A. Woodard, B. Blaiszik, J. Bryan, D. S. Katz, I. Foster, and K. Chard, "funcx: Federated function as a service for science," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4948–4963, 2022.

[93] I. Casas, J. Taheri, R. Ranjan, L. Wang, and A. Y. Zomaya, "A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems," *Future Generation Computer Systems*, vol. 74, pp. 168–178, Sep. 2017.

[94] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2018, pp. 271–284.

[95] C. Denninnart, J. Gentry, and M. Amini Salehi, "Improving robustness of heterogeneous serverless computing systems via probabilistic task pruning," in *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium Workshops*, ser. IPDPSW '19. IEEE, Jun. 2019, pp. 6–15.

[96] C. Denninnart and M. A. Salehi, "Harnessing the potential of function-reuse in multimedia cloud systems," *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 31, no. 3, pp. 617—-629, 2021.

[97] C. Denninnart, M. Amini Salehi, A. N. Toosi, and X. Li, "Leveraging computational reuse for cost- and qos-efficient task scheduling in clouds," in *Proceedings of the 16th International Conference on Service-Oriented Computing*, ser. ICSOC '18, Nov. 2018, pp. 828–836.

[98] H. Sim, S. Kenzhegulov, and J. Lee, "Dps: Dynamic precision scaling for stochastic computing-based deep neural networks," in *Proceedings of the 55th Annual Design Automation Conference*, Jun. 2018, pp. 1–6.

[99] A. Rahimi, A. Ghofrani, K.-T. Cheng, L. Benini, and R. K. Gupta, “Approximate associative memristive memory for energy-efficient gpus,” in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, ser. Data ’15. IEEE, Mar. 2015, pp. 1497–1502.

[100] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt, “Duphunter: Flexible high-performance deduplication for docker registries,” in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC ’20, Jul. 2020, pp. 769–783.

[101] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, “Cimplifier: Automatically debloating containers,” in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, Aug. 2017, pp. 476–486.

[102] V. Vassiliadis, M. A. Johnston, and J. L. McDonagh, “Fast, transparent, and high-fidelity memoization cache-keys for computational workflows,” in *Proceedings of the IEEE International Conference on Services Computing (SCC)*. IEEE, 2022, pp. 174–184.

[103] Y. Tang and J. Yang, “Secure Deduplication of General Computations,” in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIC ATC’ 15, Jul. 2015, pp. 319–331.

[104] C. Denninnart, *Cost- and QoS-Efficient Serverless Cloud Computing*. University of Louisiana at Lafayette, 2020.

[105] J. Zhang, A. Wang, X. Ma, B. Carver, N. J. Newman, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan *et al.*, “Infinistore: Elastic serverless cloud storage,” *arXiv preprint arXiv:2209.01496*, 2022.

[106] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, “Incapprox: A data analytics system for incremental approximate computing,” in *Proceedings of the 25th International Conference on World Wide Web*, Apr. 2016, pp. 1133–1144.

[107] A. Mokhtari, P. Jamshidi, and M. Amini Salehi, “Felare: Fair scheduling of machine learning applications on heterogeneous edge systems,” in *Proceedings of the 15th IEEE Cloud Conference*, Jul. 2022.

[108] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards wearable cognitive assistance,” in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’14, Jun. 2014, pp. 68–81.

[109] P. Guo, B. Hu, R. Li, and W. Hu, “FoggyCache: Cross-Device Approximate Computation Reuse,” in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom ’18. Association for Computing Machinery, Oct. 2018, pp. 19–34.

[110] T. Jiang, X. Chen, Q. Wu, J. Ma, W. Susilo, and W. Lou, “Secure and efficient cloud data deduplication with randomized tag,” *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 3, pp. 532–543, Mar. 2017.

[111] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, “Cloudcache: On-demand flash cache management for cloud computing,” in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, ser. FAST ’16, Mar. 2016, pp. 355–369.

[112] Y. Mao, E. Kohler, and R. T. Morris, “Cache craftiness for fast multicore key-value storage,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, Apr. 2012, pp. 183–196.

[113] C. Wallenta, J. Kim, P. J. Bentley, and S. Hailes, “Detecting interest cache poisoning in sensor networks using an artificial immune algorithm,” *Applied Intelligence*, vol. 32, no. 1, pp. 1–26, 2010.

[114] X. Wang, G. Li, X. Dong, J. Li, L. Liu, and X. Feng, “Accelerating deep learning inference with cross-layer data reuse on gpus,” in *Proceedings of the European Conference on Parallel Processing*, ser. Euro-Par ’20. Springer, Aug. 2020, pp. 219–233.

[115] L. Popa, M. Budiu, Y. Yu, and M. Isard, “DryadInc: Reusing Work in Large-Scale Computations,” in *Proceedings of 1st USENIX workshop on Hot Topics in Cloud Computing*, ser. HotCloud ’09, Jun. 2009.

[116] D. Tiwari and Y. Solihin, “MapReuse : Reusing Computation in an In-Memory MapReduce System,” in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS ’14, May 2014, pp. 61–71.

[117] M. Hersche, G. Karunaratne, G. Cherubini, L. Benini, A. Sebastian, and A. Rahimi, “Constrained few-shot class-incremental learning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 9057–9067.

[118] H. Zhang, M. Shen, Y. Huang, Y. Wen, Y. Luo, G. Gao, and K. Guan, “A serverless cloud-fog platform for dnn-based video analytics with incremental learning,” *arXiv preprint arXiv:2102.03012*, 2021.

[119] Y. Wu, Y. Chen, L. Wang, Y. Ye, Z. Liu, Y. Guo, and Y. Fu, “Large scale incremental learning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 374–382.

[120] Z. Huang, S. Wu, S. Jiang, and H. Jin, “Fastbuild: Accelerating docker image building for efficient development and deployment of container,” in *Proceedings of the 35th Symposium on Mass Storage Systems and Technologies*, ser. MSST ’19. IEEE, May 2019, pp. 28–37.

[121] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX symposium on operating systems design and implementation*, ser. (OSDI ’16), Nov. 2016, pp. 265–283.

[122] N. Saurabh, J. Remmers, D. Kimovski, R. Prodan, and J. G. Barbosa, “Semantics-aware virtual machine image management in iaas clouds,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS ’19. IEEE, May 2019, pp. 418–427.

[123] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv preprint arXiv:1804.02767*, 2018.

[124] Y. Qu, H. Dai, Y. Zhuang, J. Chen, C. Dong, F. Wu, and S. Guo, “Decentralized federated learning for uav networks: Architecture, challenges, and opportunities,” *IEEE Network*, vol. 35, no. 6, pp. 156–162, 2021.

[125] S. Seneviratne and D. C. Levy, “Task profiling model for load profile prediction,” *Future Generation Computer Systems*, vol. 27, no. 3, pp. 245–255, 2011.

[126] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, “Auctionwhisk: Using an auction-inspired approach for function placement in serverless fog platforms,” *Software: Practice and Experience*, vol. 52, no. 5, pp. 1143–1169, 2022.

[127] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, “A survey on deep transfer learning,” in *Proceedings of the International conference on artificial neural networks*, Oct. 2018, pp. 270–279.

[128] W. Li, R. Huang, J. Li, Y. Liao, Z. Chen, G. He, R. Yan, and K. Gryllias, “A perspective survey on deep transfer learning for fault diagnosis in industrial scenarios: Theories, applications and challenges,” *Mechanical Systems and Signal Processing*, vol. 167, p. 108487, 2022.

[129] H. Jayakumar, A. Raha, Y. Kim, S. Sutar, W. S. Lee, and V. Raghunathan, “Energy-efficient system design for iot devices,” in *Proceedings of the 21th Asia and South Pacific Design Automation Conference*, ser. ASP-DAC ’16. IEEE, Jan. 2016, pp. 298–301.

[130] M. Satyanarayanan and D. Narayanan, “Multi-fidelity algorithms for interactive mobile applications,” *Wireless Networks*, vol. 7, no. 6, pp. 601–607, 2001.

[131] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, “Analysis and characterization of inherent application resilience for approximate computing,” in *Proceedings of the 50th Annual Design Automation Conference*, May 2013, pp. 1–9.

[132] A. Hepworth, K. Tew, M. Trent, D. Ricks, C. G. Jensen, and W. E. Red, “Model consistency and conflict resolution with data preservation in multi-user computer aided design,” *Journal of Computing and Information Science in Engineering*, vol. 14, no. 2, 2014.

[133] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, “Paraprox: Pattern-based approximation for data parallel applications,” *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 35–50, Apr. 2014.

[134] C. Denninnart and M. A. Salehi, “SMSE: A Serverless Platform for Multimedia Cloud Systems,” *arXiv preprint arXiv:2201.01940*, 2022.

[135] Z. Wen, P. Bhatotia, R. Chen, M. Lee *et al.*, “Approxiot: Approximate analytics for edge computing,” in *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS ’18. IEEE, Jul. 2018, pp. 411–421.

[136] A. Gupta, A. Bansal, and V. Khanduja, “Modern lossless compression techniques: Review, comparison and analysis,” in *Proceedings of the 2nd International Conference on Electrical, Computer and Communication Technologies*, ser. ICECCT ’17. IEEE, Feb. 2017, pp. 1–8.

[137] X. Zheng, R. Zarcone, D. Paiton, J. Sohn, W. Wan, B. Olshausen, and H.-S. P. Wong, “Error-resilient analog image storage and compression with analog-valued rram arrays: An adaptive joint source-channel coding approach,” in *Proceedings of the IEEE International Electron Devices Meeting*, ser. IEDM ’18. IEEE, Dec. 2018, pp. 3–5.

[138] J. Paulo and J. Pereira, “Distributed Exact Deduplication for Primary Storage Infrastructures,” in *Proceedings of the 14th IFIP International Conference on Distributed Applications and Interoperable Systems*, Jun. 2014, pp. 52–66.

[139] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *Proceedings of the 10th European Conference on Computer Systems*, Apr. 2015, pp. 1–16.

[140] P. Pokhilko, E. Epifanovsky, and A. I. Krylov, “Double precision is not needed for many-body calculations: Emergent conventional wisdom,” *Journal of chemical theory and computation*, vol. 14, no. 8, pp. 4088–4096, 2018.

[141] A. R. Zamani, I. Petri, J. Diaz-Montes, O. Rana, and M. Parashar, “Edge-supported approximate analysis for long running computations,” in *Proceedings of the 5th IEEE International Conference on Future Internet of Things and Cloud*, ser. Ficloud ’17. IEEE, Aug. 2017, pp. 321–328.

[142] M. A. Salehi, J. Smith, A. A. Maciejewski, H. J. Siegel, E. K. Chong, J. Apodaca, L. D. Briceno, T. Renner, V. Shestak, J. Ladd *et al.*, “Stochastic-based robust dynamic resource allocation for independent tasks in a heterogeneous computing system,” *Journal of Parallel and Distributed Computing*, vol. 97, pp. 96–111, 2016.

[143] S. Vogel, A. Guntoro, and G. Ascheid, “Efficient hardware acceleration for approximate inference of bitwise deep neural networks,” in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, ser. DASIP ’17. IEEE, Sep. 2017, pp. 1–6.

[144] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, “Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency,” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’16. IEEE, Oct. 2016, pp. 1–14.

[145] Z. Du, K. Palem, A. Lingamneni, O. Temam, Y. Chen, and C. Wu, “Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators,” in *Proceedings of the 19th Asia and South Pacific Design Automation Conference*, ser. ASP-DAC ’14. IEEE, Jan. 2014, pp. 201–206.

[146] A. Bookstein, V. A. Kulyukin, and T. Raita, “Generalized hamming distance,” *Information Retrieval*, vol. 5, no. 4, pp. 353–375, 2002.

[147] F. S. Snigdha, D. Sengupta, J. Hu, and S. S. Sapatnekar, “Optimal design of jpeg hardware under the approximate computing paradigm,” in *Proceedings of the 53nd ACM/EDAC/IEEE Design Automation Conference*, ser. DAC ’16. IEEE, Jun. 2016, pp. 1–6.

[148] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi *et al.*, “Axilog: Language support for approximate hardware design,” in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE ’15. IEEE, Mar. 2015, pp. 812–817.

[149] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, “Serverless applications: Why, when, and how?” *IEEE Software*, 2020.

[150] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, “Grandslam: Guaranteeing slas for jobs in microservices execution frameworks,” in *Proceedings of the 14th EuroSys Conference*, Mar. 2019, pp. 1–16.

[151] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, “Fifer: Tackling resource underutilization in the serverless era,” in *Proceedings of the 21st International Middleware Conference*, Dec. 2020, pp. 280–295.

[152] D. Yin and T. Kosar, “Data-aware approximate workflow scheduling,” *arXiv preprint arXiv:1805.10499*, 2018.

[153] J. Gentry, C. Denninnart, and M. A. Salehi, “Robust dynamic resource allocation via probabilistic task pruning in heterogeneous computing systems,” in *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS ’19. IEEE, Jun. 2019, pp. 375–384.

[154] A. Mokhtari, C. Denninnart, and M. A. Salehi, “Autonomous task dropping mechanism to achieve robustness in heterogeneous computing systems,” in *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium Workshops*, ser. IPDPSW ’20. IEEE, May 2020, pp. 17–26.

[155] R. B. Roy, T. Patel, and D. Tiwari, “Daydream: Executing dynamic scientific workflows on serverless platforms with hot starts,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2022.

[156] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, “A review of serverless use cases and their characteristics,” *arXiv preprint arXiv:2008.11110*, 2020.

[157] L. Torrey and J. Shavlik, “Transfer learning,” in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI global, 2010, pp. 242–264.

[158] N. Daw, U. Bellur, and P. Kulkarni, “Xanadu: Mitigating cascading cold starts in serverless function chain deployments,” in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 356–370.

[159] V. Tankov, D. Valchuk, Y. Golubev, and T. Bryksin, “Infrastructure in code: Towards developer-friendly cloud applications,” in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1166–1170.

[160] Amazon, “Fast NoSQL Key-Value Database - Amazon DynamoDB | Amazon Web Services,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://aws.amazon.com/dynamodb

[161] ——, “AWS Serverless Application Model | Amazon Web Services,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://aws.amazon.com/serverless/sam/

[162] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI ’18, Oct. 2018, pp. 427–444.

[163] Amazon, “Amazon RDS | Cloud Relational Database | Amazon Web Services,” online; Accessed on 21 Mar. 2023. [Online]. Available: https://aws.amazon.com/rds

[164] ——, “AWS Identity and Access Management (IAM),” online; Accessed on 4 May 2022. [Online]. Available: https://aws.amazon.com/iam/

[165] ——, “Amazon API Gateway — Amazon Web Services,” online; Accessed on 4 May 2022. [Online]. Available: https://aws.amazon.com/api-gateway/

[166] L. Baresi, D. F. Mendonça, and M. Garriga, “Empowering low-latency applications through a serverless edge computing architecture,” in *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer, Sep. 2017, pp. 196–210.

[167] K. F. A. Friebel, S. Soldavini, G. Hempel, C. Pilato, and J. Castrillon, “From domain-specific languages to memory-optimized accelerators for fluid dynamics,” in *Proceedings of the FPGA for HPC Workshop, held in conjunction with IEEE Cluster 2021*, Sep. 2021.

[168] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, “Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 1–17.

[169] N. C. Thompson and S. Spanuth, “The decline of computers as a general purpose technology,” *Communications of the ACM*, vol. 64, no. 3, pp. 64–72, 2021.

[170] FFmpeg, “FFmpeg,” online; Accessed on 2 Apr. 2023. [Online]. Available: https://ffmpeg.org

[171] H. Zeng, Z. Zhang, and L. Shi, “Research and implementation of video codec based on ffmpeg,” in *Proceedings of 2nd International Conference on Network and Information Systems for Computers (ICNISC)*, Apr. 2016, pp. 184–188.

[172] S. Wu, C. Denninnart, X. Li, Y. Wang, and M. A. Salehi, “Descriptive and predictive analysis of aggregating functions in serverless clouds: the case of video streaming,” in *Proceedings of the 22nd IEEE International Conference on High Performance Computing and Communications*. IEEE, Dec. 2020, pp. 19–26.

[173] S. Risco and G. Moltó, “Gpu-enabled serverless workflows for efficient multimedia processing,” *Applied Sciences*, vol. 11, no. 4, p. 1438, Feb. 2021.

[174] C. Li, J. Bai, Y. Ge, and L. Youlong, “Heterogeneity-aware elastic provisioning in cloud-assisted edge computing systems,” *Future Generation Computer Systems (FGCS)*, vol. 112, pp. 1106 – 1121, 2020.

[175] V. Veillon, C. Denninnart, and M. A. Salehi, “F-fdn: Federation of fog computing systems for low latency video streaming,” in *Proceedings of the 3rd IEEE International Conference on Fog and Edge Computing*, ser. ICFEC ’19, May 2019, pp. 1–9.

[176] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, “The seminal role of edge-native applications,” in *Proceedings of the IEEE International Conference on Edge Computing*, ser. EDGE ’19. IEEE, Jul. 2019, pp. 33–40.

[177] K. Kritikos and P. Skrzypek, “A review of serverless frameworks,” in *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 161–168.

[178] E. Marin, D. Perino, and R. Di Pietro, “Serverless computing: A security perspective,” *Journal of Cloud Computing*, vol. 11, no. 1, pp. 1–12, 2022.

[179] O. W. A. S. Project, “OWASP Top 10 (2017) Interpretation for Serverless,” online; Accessed on 30 Mar. 2022. [Online]. Available: https://owasp.org/www-pdf-archive/OWASP-Top-10-Serverless-Interpretation-en.pdf

[180] M. Amini Salehi, T. Caldwell, A. Fernandez, E. Mickiewicz, E. W. D. Rozier, S. Zonouz, and D. Redberg, “Reseed: A secure regular-expression search tool for storage clouds,” *Software: Practice and Experience*, vol. 47, no. 9, pp. 1221–1241, 2017.

[181] Amazon, "Serverless CI/CD for the Enterprise on AWS," online; Accessed on 1 Jun 2022. [Online]. Available: https://aws.amazon.com/quickstart/architecture/serverless-cicd-for-enterprise

[182] R. Containers, "Rootless Containers | Rootless Containers," online; Accessed on 30 Mar. 2022. [Online]. Available: https://rootlesscontaine.rs/

[183] Google, "gVisor," online; Accessed on 30 Mar. 2022. [Online]. Available: https://github.com/google/gvisor

[184] K. Containers, "Kata Containers - Open Source Container Runtime Software | Kata Containers," online; Accessed on 30 Mar. 2022. [Online]. Available: https://katacontainers.io/

[185] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with Serverless-Optimized containers," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '18. Boston, MA: USENIX Association, Jul. 2018, pp. 57–70.

[186] D. Kelly, F. G. Glavin, and E. Barrett, "Denial of wallet—defining a looming threat to serverless computing," *Journal of Information Security and Applications*, vol. 60, p. 102843, 2021.