Rise of the Planet of Serverless Computing: A Systematic Review

JINFENG WEN, Peking University
ZHENPENG CHEN, University College London
XIN JIN and XUANZHE LIU, Peking University

Serverless computing is an emerging cloud computing paradigm, being adopted to develop a wide range of software applications. It allows developers to focus on the application logic in the granularity of function, thereby freeing developers from tedious and error-prone infrastructure management. Meanwhile, its unique characteristic poses new challenges to the development and deployment of serverless-based applications. To tackle these challenges, enormous research efforts have been devoted. This article provides a comprehensive literature review to characterize the current research state of serverless computing. Specifically, this article covers 164 articles on 17 research directions of serverless computing, including performance optimization, programming framework, application migration, multi-cloud development, testing and debugging, and so on. It also derives research trends, focus, and commonly-used platforms for serverless computing, as well as promising research opportunities.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **General and reference → Surveys and overviews**;

Additional Key Words and Phrases: Serverless computing, literature view

1 INTRODUCTION

Serverless computing is an emerging cloud computing paradigm. It has been adopted to develop a wide range of software applications, including machine/deep learning [251, 267], numerical computing [226], video processing [74, 124], Internet of Things [143, 276], big data analytics [119,

ACM Transactions on Software Engineering and Methodology, Vol. 32, No. 5, Article 131. Pub. date: July 2023.

131

129], and so on. According to a recent report [1], the serverless market size will reach nearly $22 thousand million in 2025 from $ 3 thousand million in 2017. Moreover, it is predicted that serverless computing can be employed in 50% of global enterprises by 2025 [26].

The popularity of serverless computing can be attributed to its unique characteristics. Specifically, serverless computing allows software developers to focus on only the application logic in the granularity of function without having to manage complex and error-prone underlying tasks. The reduction in underlying cloud management is undoubtedly exciting news for software developers without a background in hardware infrastructure. Moreover, in serverless computing, software developers pay for only the resources actually consumed or allocated by their applications at a fine-grained pattern. This point is different from traditional cloud computing, where software developers always rent and retain resources regardless of whether the application is running. In addition, serverless computing also makes cloud providers manage resources in a unified manner, improving resource utilization and reducing resource waste. Based on these benign characteristics and its bright prospect, many major cloud providers have rolled out their serverless platforms, such as AWS Lambda [16], Microsoft Azure Functions [22], and Google Cloud Functions [35]. Moreover, there are also some available open-source serverless platforms, e.g., OpenWhisk [48] and OpenFaaS [45].

However, the unique characteristics of serverless computing also pose new challenges or issues to the development and deployment of serverless-based applications (i.e., *serverless applications*) [145, 253, 256, 257]. The **software engineering (SE)** research community has focused on a wide range of topics about serverless computing, including serverless evolution [242], characteristic analysis of serverless applications [114, 115], developers' challenges [256], application modeling [271], programming framework of specific applications [84, 276], multi-cloud development [218], stateful serverless applications [82], application migration [213], serverless economic [63], serverless dataset [120], technical debt conceptualization [162], testing and debugging [163], and so on. Moreover, other communities like the Systems research community, the Network research community, and the Services Computing research community have made significant efforts in resource management [136, 199, 274], cold start performance optimization [67, 125, 197], function communication [142, 229], general programming framework [123, 139], and so on. Addressing these challenges or issues can better facilitate software developers' application development practices on serverless platforms. However, to the best of our knowledge, there has not been a thorough analysis effort in these communities to investigate the current research state of serverless computing from the research scope and depth. A comprehensive literature review is a foundation for understanding an evolving research area. In the absence of such a literature review, it is challenging for researchers and practitioners to quickly grasp a global overview of research directions that have been studied and existing solutions in the serverless computing field. Moreover, it may prevent best software practices for serverless application engineering and the long-term evolution of the serverless computing ecosystem.

In this article, to fill this knowledge gap, we present a comprehensive literature review to explore the research scope and depth of the serverless computing literature. Our literature review is based on the collected 164 research articles to analyze and answer four key aspects, i.e., research directions, existing solutions, experimental setting and evaluation, and publication venues. Specifically, first, we aim at constructing a taxonomy for research directions of serverless computing to provide a global literature overview. Our taxonomy contains 17 research categories covering performance optimization, programming framework, application migration, cost, testing and debugging, and so on. Second, we aim at providing an in-depth analysis of existing solutions. We classify related studies of each research direction and elaborate on proposed solutions. Third, we aim at exploring how the existing solutions conduct the experimental setting and evaluation. We investigate the
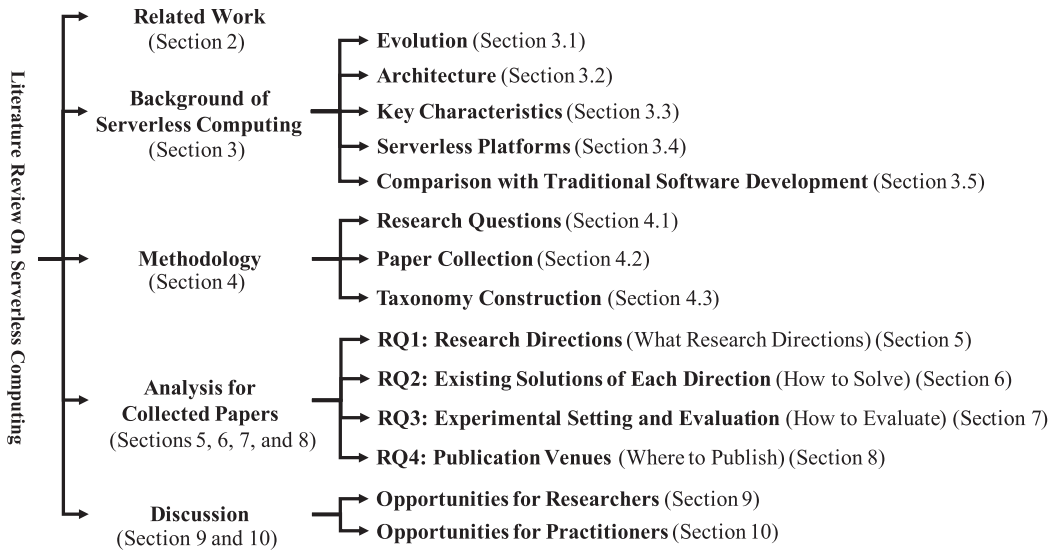
Fig. 1. Tree structure of the contents in this article.

distribution of experimental serverless platforms, the availability of experimental validation, and the availability of experimental datasets or codes. Fourth, we aim at showing the distribution of publication venues for selected research articles. Finally, we discuss open challenges and envision promising opportunities for researchers and practitioners related to serverless computing. In addition, we offer the data for research articles selected in this study[1] as an additional contribution to allow other researchers to replicate and update the results.

Figure 1 shows the content structure of this article. Section 2 introduces the related work. Section 3 summarizes the background of serverless computing, including evolution, architecture, characteristics, existing serverless platforms, and comparison with traditional software development. Section 4 presents our research methodology, including the research questions, the collection of articles, and the construction of taxonomy. Sections 5, 6, 7, and 8 answer our research questions based on the analysis of the collected articles. Specifically, Section 5 summarizes the research directions in the serverless computing literature; Section 6 classifies and elaborates on existing solutions for each research direction; Section 7 investigates the experimental setting and evaluation for solutions; Section 8 shows the distribution of publication venues. Based on the aforementioned analysis, we discuss opportunities for researchers and practitioners in Sections 9 and 10, respectively. Finally, Section 11 concludes this work.

2 RELATED WORK

There has been previous work that discussed different aspects of existing serverless platforms, including serverless architecture design [165, 247], development features and limitations [77, 130, 253], technology aspects [270], performance properties of serverless platforms [171, 257, 258], and so on. For instance, Li et al. [165] detailed the serverless architecture by introducing the related concepts, pros, and cons, and provided some architecture implications. Grogan et al. [130] compared the supported development options (e.g., language, concurrency, memory allocation) and constraints (e.g., deployment package size and payload size) of different serverless platforms. Yussupov et al. [270] presented a comprehensive technology review to analyze and compare the

---

[1]https://github.com/WenJinfeng/Serverless_Survey.

ten most prominent serverless platforms from development, event source, observability, access management, and so on.

In addition to the aforementioned studies, there have also been efforts to focus on surveying research work targeted at specific aspects, such as the cold start problem and scheduling policy [164] and resource management [183]. For example, Mampage et al. [183] presented a comprehensive review on the aspect of resource management. They analyzed and discussed the existing related work based on the proposed taxonomy. However, these studies cannot provide a global overview of the current state-of-the-art in research, making the authors may give some wrong or already existing discussions and prospects. Hassan et al. [134] presented a survey on research articles related to serverless computing. This survey mainly showed statistical information about research articles, such as the number of published articles per year, researcher distribution, and use cases, lacking the summary and classification of specific solutions. Overall, as far as we know, no previous work has provided a comprehensive literature review focused on the current research scope and in-depth analysis of specific solutions.

3 BACKGROUND

In this section, we introduce the background of serverless computing, including its evolution, architecture, key characteristics, and mainstream serverless platforms. Moreover, we briefly summarize the differences between serverless-based software development and traditional software development.

3.1 Evolution of Serverless Computing

Cloud computing provides the ability of computation services via the Internet. According to the NIST definition [187], traditional cloud computing has three service categories: "**Infrastructure as a Service**" (**IaaS**), "**Platform as a Service**" (**PaaS**), and "**Software as a Service**" (**SaaS**). Specifically, IaaS allows software developers to configure and use computation, storage, and network resources. For example, AWS provides a computation service like Elastic Compute Cloud (AWS EC2) [10] and a storage service like Simple Storage Service (AWS S3) [11]. However, IaaS does not hide the operation complexity of the application; thus, developers are still responsible for resource provisioning, runtime configuration, application code management, and so on. SaaS allows software developers to directly use the cloud provider's applications, such as Gmail [38] and Docs [37] provided by Google. SaaS completely hides the underlying operation complexity, but use cases are limited. Moreover, developers completely lose control of the application. PaaS allows software developers to develop, run, and manage applications using execution environments supported by cloud providers. For example, Google provides the App Engine [34], while Azure offers the App Service [20]. PaaS compromises the operation complexity between IaaS and SaaS, but software developers related to PaaS are still responsible for a part of the management and configuration tasks, which may increase the complexity of development and deployment [248].

To ease the cloud management burden on software developers, cloud providers present a new paradigm, i.e., serverless computing. Serverless computing is similar to PaaS [115, 242]; differently, it almost hides all complex management tasks about underlying servers for developers, i.e., "serverless", and it also allows developers to control their applications. Moreover, serverless computing can automatically scale depending on the demand, while PaaS does not [145, 248]. Unlike PaaS, serverless computing cannot support long-running processes and stateful applications [82]. Nowadays, there still is no clear-cut answer to the question of whether serverless can be considered the new-era PaaS.

Serverless computing-related applications (i.e., *serverless applications*) follow the microservice software style, which decomposes the application into a subset of independent tasks. In practice,
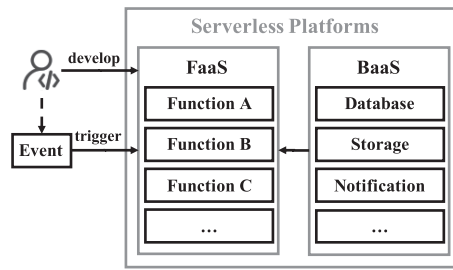
Fig. 2. The development diagram on the serverless platform.

serverless computing and microservices are closely related in certain aspects. For example, they are both dedicated to breaking a large monolith into small pieces that can be independently developed, deployed, and managed. Execution units of the serverless applications (i.e., *serverless functions*) can be viewed as one way to host microservices. In the aspect of monitoring and management, the more components contained in the application, the more moving pieces to keep track of, and the more robust the monitoring and log management tools need to be. However, serverless computing is also different from microservices as follows. First, the serverless function is a smaller granularity than the microservice, which may perform more than one function. Second, constructing microservice-based applications still needs additional efforts from developers for underlying tasks like scalability, fault tolerance, and load balancing, while the infrastructure provisioning of serverless applications is taken care of by serverless providers.

3.2 Architecture of Serverless Computing

Serverless computing is an emerging and potential cloud computing paradigm, and its significant advantage is to free software developers from the burden of complex and error-prone server management tasks. Serverless computing provides "**Backend as a Service**" (**BaaS**) and "**Function as a Service**" (**FaaS**) [145]. Specifically, BaaS represents tailor-made cloud services provided by cloud providers, e.g., cloud storage and notification services. These services can service FaaS optionally to simplify the backend functionality development for developers. FaaS represents that developers can write stateless, event-driven serverless functions, making them focus on the logic of serverless applications. Generally, FaaS is the core of serverless computing, allowing developers to develop and control their applications.

Software developers leverage the serverless platform provided by cloud providers to develop and execute their serverless applications composed of multiple serverless functions and cloud services. As shown in Figure 2, serverless functions will be triggered by pre-defined events, e.g., HTTP requests, data updates of the cloud storage, and the arrival of a notification. These events represent the developers' requirements, and developers can define some rules to bind their serverless functions with the corresponding events. When serverless functions are triggered, the serverless platform automatically prepares required runtime environments, e.g., containers or **virtual machines** (**VMs**), to serve them. These runtime environments are called function instances, and their preparation process generally contains instance initialization, application transmission, application code loading, and so on. After executions are complete, the serverless platform will automatically recycle and release these function instances and the corresponding resources.

3.3 Key Characteristics of Serverless Computing

To better understand serverless computing, we introduce its key characteristics as follows. (1) *Functionality and **no operations** (NoOps):* On serverless platforms, software developers can

select appropriate and familiar languages (e.g., Python, JavaScript, and Java) to write the function-level code snippet to create serverless applications [114, 115]. Moreover, serverless platforms provide user-friendly **integrated development environments (IDEs)**. For the deployment of serverless applications, software developers only need to upload their application code to the serverless platform without complex environment configurations. In addition, BaaS is the equivalent of off-the-shelf backend functionality. Its related services can be directly used in the application by developers to replace similar backend functionalities. Therefore, developers do not have to redevelop these functionalities and deal with server configurations [114]. (2) *Auto-scaling:* Serverless platforms can automatically scale function instances horizontally and vertically according to the application workload dynamics [134, 253]. Horizontal scaling is to launch (i.e., scale-in) new function instances or recycle (i.e., scale-out) running ones, while vertical scaling is to add (i.e., scale-up) or remove (i.e., scale-down) the amount of computation and other resources from running function instances. After completing requests, the corresponding function instances and allocated resources will retain in memory for a short time to prepare to be reused by subsequent requests of the same function. If there are no subsequent requests, these instances and resources will be automatically recycled by the serverless platform, i.e., scaling to zero. However, scaling to zero makes incoming new requests face the cold start problem, which takes a long time to prepare required runtime environments from scratch. (3) *Utilization-based billing:* In serverless computing, software developers charge for the actually allocated or consumed resources of the serverless application in the fine-granular execution unit [102, 167]. For example, AWS Lambda's pricing is related to the allocated memory, and Azure Functions considers the consumed memory. In addition, serverless functions are event-driven; thus, they will not run without being triggered, and developers do not pay any cost. This feature eliminates the concern of paying for idle resources. In summary, the billing pattern of serverless computing is relatively reasonable and inexpensive compared with traditional cloud computing, which requires always renting and paying for resources in memory on standby. (4) *Separation of computation and storage:* Serverless computing adopts the separation way of computation and storage [145], i.e., separately scaling and independently provisioning and pricing. Generally, computation refers to stateless serverless functions, while storage represents cloud storage services provided by cloud providers to store data from the serverless function. This separation way can ensure the auto-scaling ability of the serverless platform for bursty workloads. (5) *Additional limitations:* Cloud providers set some additional limitations for serverless functions to keep the vital auto-scaling feature of serverless platforms [77, 130]. Generally, these limitations contain function execution timeout, deployment package size, local disk size, memory allocation maximum, and so on. Moreover, different serverless platforms have different demands regarding these additional limitations. The following section will list specific demands for some serverless platforms.

3.4 Serverless Platforms

Major cloud providers have rolled out their commercial serverless platforms, such as AWS Lambda [16], Microsoft Azure Functions [22], and Google Cloud Functions [35]. However, these commercial serverless platforms hide the platform's underlying details and have a vendor lock-in problem for software developers. To address these restrictions, the serverless computing field has presented some open-source serverless platforms, such as OpenWhisk [48], OpenFaaS [45], and OpenLambda [47]. Their recognition and popularity are also due in part to the popularity of container orchestration like Kubernetes [42]. These open-source serverless platforms are being actively maintained.

Next, we will introduce some mainstream serverless platforms.

AWS Lambda: AWS Lambda is the most widely mentioned serverless platform. Since it was released in November 2014, serverless computing has started to gain increasing attention, and

Table 1. Feature Comparison of Serverless Platforms

Features	AWS Lambda	Microsoft Azure Functions	Google Cloud Functions	OpenWhisk	OpenFaaS	OpenLambda
Interface ways	CLI/API/GUI	CLI/API/GUI	CLI/API/GUI	CLI/API	CLI/API/GUI	CLI/API
Plugins for IDEs	VS/VS Code	VS/VS Code	Not available	VS Code/Xcode	Not available	Not available
Billing model	Execution time and allocated memory	Execution time and consumed memory	Execution time and allocated memory	Execution time and allocated memory	Unknown	Unknown
Timeout limitation	900 seconds	600 seconds	540 seconds	600 seconds	30 seconds	Unknown
Package size limitation	250 MB (uncompressed) and 50 MB (compressed)	No limit	500 MB (uncompressed) and 100 MB (compressed)	48 MB (uncompressed)	Unknown	Unknown
Memory allocation	From 128 MB to 10,240 MB	1,536 MB	128, 256, 512, 1,024, 2,048, and 4,096 MB	From 128 MB to 2,048 MB	Unknown	Unknown
Observability	AWS CloudWatch, AWS CloudTrail	Azure Application Insights	Google Cloud Operations	External tools	External tools	External tools
Serverless marketplace	AWS Serverless Application Repository	Azure Marketplace	Not available	Not available	OpenFaaS Function Store	Not available

other major cloud providers have followed this trend by releasing their serverless platforms. AWS Lambda offers different interaction ways for developers, including the **command-line interface (CLI)**, HTTP-based **application programming interface** (**API**), and **graphical user interface (GUI)**. Moreover, software developers can use the related IDE plugins, e.g., **Visual Studio (VS)** and **Visual Studio Code (VS Code)** [56], to access the platform through language-specific client libraries. The pricing of AWS Lambda is related to function execution time (in increments of 1 millisecond [17]), allocated memory size, and the number of invocations. In AWS Lambda, the function execution time, deployment package size, and memory allocation configuration are limited. At the time our article was written, its execution time limitation was 900 seconds, its deployment process supported up to 250 MB uncompressed size and 50 MB compressed size, and its memory allocation can be configured between 128 MB and 10,240 MB in the increment of 1 MB. For the observability of AWS Lambda, Amazon provides AWS CloudWatch [9] and AWS CloudTrail [13] to monitor and log serverless functions. In addition, AWS Lambda provides a serverless marketplace called **AWS Serverless Application Repository (AWS SAR)** [18] for application development purposes. This marketplace contains some serverless functions or applications contributed by third-party teams.

Microsoft Azure Functions: Microsoft released its serverless platform Azure Functions in 2016. Azure Functions offers various interaction ways like CLI, API, and GUI and uses related plugins for VS and VS Code [56] to access the platform. The pricing model of Azure Functions is similar to AWS Lambda, but it relies on the consumed memory of serverless functions. Moreover, the minimum execution time of billing is in increments of 100 milliseconds [23]. Azure Functions

uses the function app as the execution and management unit, which is still composed of several functions. Azure Functions also has a function execution time limitation, e.g., 600 seconds at the time our article was written. Azure Functions has no deployment package limit and uses a flexible memory allocation, supporting 1,536 MB at most at the time our article was written. Microsoft uses Azure Application Insights [21] to provide the observability of Azure Functions. In addition, Azure Functions adopts three hosting plans: consumption, premium, and dedicated plans for serverless applications. For the marketplace, Microsoft provides a general-purpose Azure Marketplace [25] to include serverless applications.

Google Cloud Functions: In 2017, Google released its serverless platform, i.e., Google Cloud Functions. Like AWS Lambda and Microsoft Azure Functions, Google Cloud Functions supports various interaction ways like CLI, API, and GUI. However, Google Cloud Functions has no related plugins for IDEs to access the platform. The pricing model of Google Cloud Functions is related to provisioned memory and CPU. Similarly, Google Cloud Functions has limitations regarding the function execution time, deployment package size, and memory allocation size. For example, at the time our article was written, the function execution time limitation was 540 seconds. The deployment size had a 500 MB uncompressed size limit and a 100 MB compressed size limit. Developers can assign fixed memory values like 128 MB, 256 MB, 512 MB, 1,024 MB, 2,048 MB, and 4,096 MB. Google offers its Operation suite (i.e., Google Cloud Operations [36]) to achieve observability. In addition, Google does not provide a marketplace of serverless applications, but it has some code samples to guide the development process.

OpenWhisk: Apache OpenWhisk is an open-source serverless platform developed and maintained by IBM and Apache. It was released in 2016. Moreover, IBM Cloud Functions is based on OpenWhisk. In OpenWhisk, it combines several key technologies, e.g., Nginx [44], CouchDB [28], Kafka [41], and Docker [32]. Function invocations can be transformed as HTTP requests and imported into the Nginx server that supports Web protocol. The Nginx server pushes the request to the controller, which collaborates with the CouchDB that stores the data of the application. The controller and underlying worker nodes rely on Kafka, a publish-subscribe messaging system, to communicate. Kafka can receive messages from the controller to confirm the invoked worker node. For the programming model of OpenWhisk, actions, triggers, and rules are primary concepts. Specifically, actions represent functions to be executed, triggers are predefined events, and rules refer to the binding description between actions and triggers. In addition, OpenWhisk offers CLI and API interactions and supports local development and cloud environment development. Developers can leverage IDEs like VS Code [56] and Xcode [61] to connect OpenWhisk. The billing model of OpenWhisk relies on execution time and allocated memory that actions use. Each action has a default execution timeout limit of 600 seconds. The maximum code size for an action is 48 MB. The memory size can be set in the range of 128 MB to 2,048 MB. OpenWhisk relies on external tools for monitoring, e.g., Prometheus [50]. In OpenWhisk, there is no marketplace for serverless applications.

OpenFaaS: OpenFaaS is a project developed by Alex Ellis in 2016. The underlying system of OpenFaaS is based on Docker [32] and Kubernetes [42]. OpenFaaS can be deployed in public or private clouds, even in edge devices, due to its lightweight. In OpenFaaS, developers can use CLI, API, and GUI to implement interaction operations, but no related development IDEs. Generally, developers utilize CLI to communicate with the OpenFaaS gateway. This gateway connects an external function monitor tool called Prometheus [50], which records values of function-related metrics. In addition, OpenFaaS also supports workflow orchestration with synchronous and asynchronous function chains, parallelism, and branch. The default timeout of OpenFaaS is 30 seconds. OpenFaaS has an OpenFaaS Function Store [46] as its marketplace.

OpenLambda: OpenLambda is an Apache-licensed open-source serverless platform. This platform is based on Linux containers. In OpenLambda, developers need to upload their functions to the code store or function registry. When a serverless function is triggered, requests are sent to the load balancer component. This component can select appropriate workers leveraging the configured algorithm to serve requests. Function scheduling in OpenLambda is performed by the Nginx software load balancer. OpenLambda supports Docker containers and lightweight SOCK containers [197]. Moreover, OpenLambda offers CLI and API interactions for coordination with a variety of backends. Similarly, some external monitoring tools can also be utilized in OpenLambda for observability.

For these serverless platforms, CLI and API are common interface types to establish programmatic access. Except for Google Cloud Functions, other platforms support the development and deployment of the custom container image. This deployment way allows developers to use any programming language and heavy third-party library to avoid potential language and deployment package size limits. However, it also increases the additional burden of container management efforts, such as interface requirements and container interaction. For commercial serverless platforms, the corresponding cloud providers offer tailor-made monitoring and logging services to observe serverless functions. However, open-source serverless platforms generally integrate external tools to achieve the observability of serverless functions. In addition, commercial serverless platforms natively offer access management for authentication and resource, while open-source serverless platforms rely on only the hosting environment to implement the related access management. The characteristics of these serverless platforms are summarized in Table 1.

3.5 Comparison with Traditional Software Development

Serverless-based software development differs from traditional software development in many aspects. Traditional software development generally has two types: non-cloud-based software development and cloud-based software development. In this section, we compare their differences with serverless-based software development from multiple aspects, as shown in Table 2. Note that cloud-based software development mainly refers to software development based on the IaaS pattern in our work.

Server management: In non-cloud-based software development, developers need to coordinate and maintain various components and implement all server-side functionalities [115, 194]. Moreover, developers endure a high failure rate for physical servers [63]. In addition, server resources are not guaranteed to be optimally utilized. In cloud-based software development, developers can rent the required resources from the cloud without having to purchase physical servers [10]. This way reduces part of the server maintenance, but developers still need to coordinate components of the server side. In serverless-based software development, developers almost do not manage complex server tasks, focusing only on application logic. Therefore, the serverless application will require fewer engineers related to the operation, maintenance, and resource management.

Functionality implementation: In non-cloud-based software development, developers go through a complex process. First, developers select a technology stack and development framework. Then, they configure the local development environment and prepare the required resources. Moreover, developers need to implement complex backend functionalities themselves [52]. Therefore, this way is relatively inefficient. In cloud-based software development, developers have complete control and configuration over all aspects of the cloud infrastructure and applications, which obtains greater development efficiency. However, this kind of development still requires time and effort to determine how to set up and secure [60]. In serverless-based software development, the application can be designed as a set of event-triggered serverless functions and optional cloud services. Serverless functions are short-lived and stateless. Cloud services like cloud storage may be

Table 2. Comparison between Serverless-based Software Development and
Traditional Software Development

Features	Non-cloud-based software development	Cloud-based software development	Serverless-based software development
Server management	Full management tasks	Partial management tasks	No management tasks
Functionality implementation	Implement all functionalities from scratch Low efficiency	Implement partial functionalities from scratch Median efficiency	Write event-driven code Use BaaS to simplify High efficiency
Invocation pattern	Client-side calls	Client-side calls	Event triggers
Execution limitations	Uncertain limitations, depending on server capacity	Controllable limitations, depending on rented resources	Fixed inherent limitations
Execution place	Local	Cloud	Cloud
Performance	Always activated No cold starts No flexibility	Configurably activated No cold starts Flexibility	Activated only if triggered Cold starts Flexibility
Cost	Pay for everything Long development time Long market release time	Pay for rented resources Accelerable development time Accelerable market release time	Pay for actually allocated or assumed resources Short development time Short market release time
Tool maturity	High	Median	Low

required to save data of serverless functions. Moreover, services contained in BaaS help developers simplify the development of backend functionalities.

Invocation pattern: Invocations in traditional software development are dependent on client-side calls from the software developer. Moreover, this way involves a complex server process [63, 194]. In serverless-based software development, invocations of applications rely on the developer's predefined events, and the invocation process is automatic [145, 248].

Execution limitations: In non-cloud-based software development, the execution limitations have a high degree of uncertainty because they depend on the capacity of their servers [52]. In cloud-based software development, the execution time is within the lease time of the resources. Moreover, as long as the developer rents enough resources, there are generally no execution limitations [88]. In serverless-based software development, the application has inherent execution limitations, including function execution timeout, confined memory size, restricted local disk size of instances, and so on [248].

Execution place: In non-cloud-based software development, applications are executed in the limited local environment of developers [27]. In cloud-based and serverless-based software development, applications can be executed in the cloud environment with enough resource provision [194, 242].

Performance: For non-cloud-based software development, runtime environments are always in the active condition to respond to application requests immediately. However, this will waste too many resources when there are no requests. Moreover, the local environment may be hard to handle workloads with variable requirements, thereby lacking flexibility [27, 52]. In cloud-based software development, the rented resources can always keep activated to serve the application execution, which has no cold start problems for application executions. Moreover, developers can flexibly select resources' ability and configure resources' service time. In serverless-based software development, the serverless provider is responsible for runtime environment management. The advantage of this kind of unified resource management is to respond to any bursty workload. These runtime environments are activated when applications are triggered. When required environments

are not active, applications may face the cold start problem, which introduces a long preparation time. There have been a lot of efforts to alleviate this problem [67, 92, 107, 197, 252, 282].

Cost: In non-cloud-based software development, developers pay for everything, such as physical server purchase and installation, as well as the cost of maintenance-related engineers [52, 115]. Moreover, non-uniform architecture and low product maturity make application development time longer and market release time slower [27]. In addition, once the application is deployed successfully to the server, the server will be "always-on", and developers have to pay for it [63]. However, in cloud-based software development, developers pay for the service time of leased resources. Since this development can simplify the server preparation by renting resources, it also speeds up development efficiency for the developer and market release time of the application compared to non-cloud-based software development. In serverless computing, software developers pay for only actual resources allocated or consumed by applications [116]. Moreover, developers do not manage complex underlying tasks, saving a lot of application development time and market release time.

Tool maturity: For non-cloud-based software development, some functionalities like testing and debugging can be freely designed and evaluated based on the local environment. Furthermore, the relevant tools are already well-grounded in the software engineering research community [31]. Before the concept of serverless computing emerged, research related to traditional cloud computing had been conducted for more than a decade [86, 207]. Therefore, in cloud-based software development, there are already some tools to help automate its development process [54, 103]. However, serverless computing is an emerging paradigm of cloud computing. Existing serverless platforms lack rich support tools, such as testing and debugging [163]. The reason may be that the event-driven, distributed, and platform detail masking features make the application architecture more complex and the execution flow harder to reproduce.

4 METHODOLOGY

A systematic literature review is a means of evaluating and interpreting all available research relevant to a topic area [148]. Following Kitchenham's standard guidelines [148], we conduct the following review protocol in this section.

4.1 Research Questions

To better understand the research scope and depth of the current research state for the serverless computing field, we aim at focusing on the following four research questions.

RQ1 (Research directions): *What research directions have been investigated in the serverless computing literature?* This research question aims at investigating the research goal of existing studies and provide an overview of research directions about serverless computing.

RQ2 (Existing solutions): *How do existing studies tackle specific problems for each research direction?* This research question aims at understanding how existing work tackles research problems in each research direction. We classify the related studies of each research direction and dissect the solutions to problems associated with each research direction.

RQ3 (Experimental setting and evaluation): *How do the existing solutions perform experimental setting and evaluation?* This research question aims at exploring three sub-research questions related to experimental setting and evaluation as follows:

— **RQ3.1:** *Where are the existing solutions implemented/evaluated?* This sub-research question aims at investigating which serverless computing platforms existing techniques are implemented or evaluated on.

— **RQ3.2:** *Did the existing solutions have experimental validation?* This research question aims at investigating whether the presented solutions are validated by experimental evaluation.
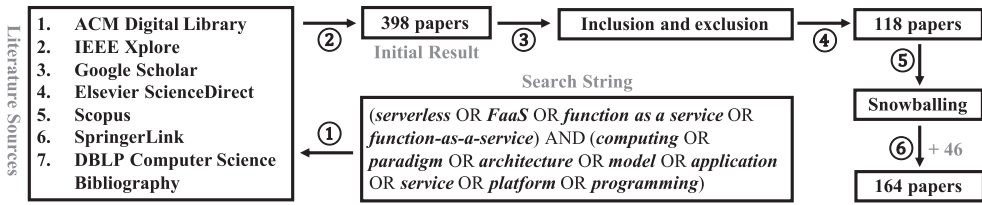
Fig. 3. The article collection process of our methodology.

— **RQ3.3:** *Did the existing studies provide the shared experimental dataset or code?* This research question aims at exploring the reproducibility of the existing solutions. We investigate whether the experimental datasets or code is shared and whether it is still accessible.

RQ4 (Publication venues): *Where are the research articles published?* This research question aims at investigating which venues existing serverless computing-related articles are published in.

4.2 Article Collection

To answer these research questions, we collect relevant published research articles about serverless computing. In this section, we describe our article collection criteria. We search the related research articles by defining keywords on the widely adopted engines and databases. Then, we determine the corresponding selection rules to collect the final research articles of the literature review for analysis. The specific article collection process of our methodology is shown in Figure 3.

4.2.1 Literature Sources. In our study, seven standard online engines or databases [134, 172, 231, 254, 270] are selected as literature sources. These sources contain (1) *ACM Digital Library*, (2) *IEEE Xplore*, (3) *Google Scholar*, (4) *Elsevier ScienceDirect*, (5) *Scopus*, (6) *SpringerLink*, and (7) *DBLP Computer Science Bibliography*.

4.2.2 Search String. To collect the research articles related to serverless computing, we follow the previous work [134, 172] to define the extensive search string and then apply it to literature sources. We define the keywords through a trial-and-error procedure performed by the first two authors and a discussion among all the authors. The final keywords used for searching contain (*serverless* OR *FaaS* OR *function as a service* OR *function-as-a-service*) AND (*computing* OR *paradigm* OR *architecture* OR *model* OR *application* OR *service* OR *platform* OR *programming*). Similarly, this search string is applied in our study to crawl research articles from the literature sources on January 1, 2022. In this situation, we collect research articles published between November 2014 (AWS release time) and January 1, 2022. In this process, we obtain 398 research articles in total.

4.2.3 Inclusion and Exclusion Criteria. We formulate the inclusion and exclusion criteria to effectively filter and select relevant research articles on serverless computing.

A article is retained when satisfying all inclusion criteria. The finalized inclusion criteria are as follows: (1) Publications that belong to serverless computing; (2) Publications that address and present the corresponding design solutions, algorithms, optimization approaches, or general ideas in terms of specific aspects of serverless computing; (3) Publications that are written in English.

A article is removed when satisfying one of the exclusion criteria. The finalised exclusion criteria are as follows: (1) Publications that are the benchmark suite; (2) Secondary or tertiary studies, e.g., empirical studies, literature reviews, and surveys; (3) Publications that are not available or full text because they cannot provide complete information; (4) Publications that are bachelor, master, or doctoral dissertations; (5) Pre-printed studies that submitted to the arXiv website. After the article screening, exclusion, and duplicate removal, we end up with 118 initial relevant research articles.

4.2.4 Snowballing. Based on the initial articles, we apply the commonly used snowballing process [134, 166, 254, 275] to increase the set of relevant research articles. The snowballing includes two steps: forward snowballing, which analyzes the references in each collected article, and backward snowballing, which searches for other relevant articles from those that cite the collected articles. We repeat the snowballing process until no new relevant articles are identified. In this phase, we add 46 research articles to our article list. As a result, we obtain 164 research articles in total for our literature review.

4.3 Taxonomy Construction

To provide an overview of research directions about serverless computing (i.e., RQ1), we aim at constructing the taxonomy of research directions. We follow the standard open coding procedure [223], which has been widely adopted in empirical software engineering research [98, 126, 138, 174, 256], to construct the taxonomy and ensure its reliability. Next, we illustrate the detailed process.

We randomly sample 70% of research articles to construct the initial taxonomy of research articles. We adopt an open coding procedure [223] to analyze selected research articles, in order to inductively create categories and subcategories of the taxonomy in a bottom-up way. The first two authors jointly participate in the taxonomy construction, and they read research articles over and over again to understand the research objectives. In this process, the *Abstract*, *Introduction*, *Related Work*, and *Conclusion* sections of all research articles are taken into account for careful inspection to determine their research goals.

The procedure of open coding is as follows. The authors give short phrases to represent research directions and then continue to group similar short phrases into categories and establish a hierarchical taxonomy of research directions. In the process of grouping categories, the authors repeatedly iterate between categories and research articles. If research articles are associated with more than one category, they are assigned to all relevant categories. If the authors have conflicts over labeling research directions, they introduce a third arbitrator, who has ten years of cloud computing experience, to discuss and resolve these conflicts. Through such a rigorous procedure, research directions of all research articles come to an agreement, and all the participants confirm the final label result, i.e., the initial taxonomy.

Next, we perform the extended construction of the taxonomy of research directions. The remaining 30% of research articles are independently labeled by the first two authors based on the initial taxonomy. Each research article is marked with the leaf categories of our taxonomy. For research articles that cannot be classified into the current taxonomy, they are placed in a new category named *Pending*. To calculate the inter-rater agreement during the independent labeling, we use Cohen's Kappa (κ) [101] as the evaluation. The value of the inter-rater agreement is 0.849, indicating an almost perfect agreement [158] and reliable labeling procedure. Then, the first two authors and the third arbitrator discuss and resolve existing conflicts. Moreover, the third arbitrator assists in identifying research articles that are temporarily in the *Pending* category. As a result, all research articles can be assigned to our taxonomy.

Taxonomy construction approach discussion. There are mainly two kinds of taxonomy construction approaches (i.e., automatic construction and manual construction) [170, 238]. Automatic construction approaches, including topic modeling, are generally applicable to datasets with large sample sizes [170, 225]. In our study, the number of selected research articles is only 164, which is not sufficient to produce reliable clustering and modeling results. In addition, the taxonomy generated by the automatic construction approach may not be orthogonal. In this article, we aim at providing orthogonal categorization of existing research directions. Therefore, we use manual
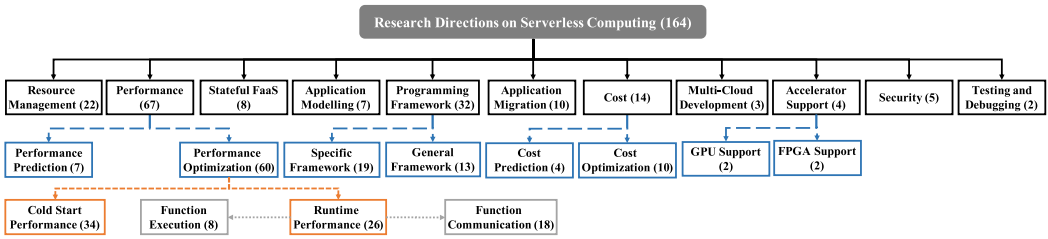
Fig. 4. A taxonomy of research directions in the serverless computing literature.

construction instead. These can explain why almost all the recent surveys [133, 166, 184, 189, 200] also use manual construction.

5 RQ1 (RESEARCH DIRECTIONS)

As shown in Figure 4, we construct a taxonomy linked to research directions of serverless computing for selected research articles. Note that a research article may belong to two or multiple research directions. For example, the work presented by Cordingly et al. [102] addressed the prediction problem of both application performance and cost because the cost is closely associated with the performance of the serverless application. Therefore, this work is assigned to the "Performance Prediction" research direction and "Cost Prediction" research direction.

Our taxonomy includes 11 root categories of research directions represented in the black box, such as "Resource Management (22)" and "Performance (67)". The number in parentheses indicates the number of articles that investigate the corresponding research direction. For example, 22 articles address the resource management problem. In addition, the boxes with colors (e.g., blue, orange, and grey) represent sub-research directions under a particular research direction scope. For instance, the research direction "Performance" contains two sub-research directions represented in the blue box: "Performance Prediction" and "Performance Optimization". The sub-research direction "Performance Optimization" includes two smaller research directions represented in the orange box: "Cold Start Performance" and "Runtime Performance". Furthermore, "Runtime Performance" is still divided into two smaller research directions represented as the grey box, i.e., "Function Execution" and "Function Communication". Non-divisible research directions are leaf categories. In total, there are 17 non-divisible research directions in this taxonomy.

Next, we explain our taxonomy in detail. First, studies related to "Performance" are the most, accounting for 40.85% (67/164) of all articles in this taxonomy. These studies contain 10.45% (7/67) "Prediction Prediction" and 89.55% (60/67) "Performance Optimization". Specifically, the performance of serverless platforms or applications affects resource utilization or developers' experience, respectively. Moreover, performance variance affects cost. Therefore, some studies have targeted the performance prediction of platforms or applications. However, most performance efforts are made on performance optimization, including 56.67% (34/60) "Cold Start Performance" optimization and 43.34% (26/60) "Runtime Performance" optimization. For the cold start performance, it refers to the overhead generated by the cold start process. When there are incoming requests, serverless platforms will automatically prepare the runtime environment from scratch to serve serverless functions, i.e., cold starts. This process needs to prepare instance initialization, application runtime, application code preparation, and so on, thus introduce undesired latency for requests and affect the user experience. For the runtime performance, it refers to the execution overhead of serverless functions (i.e., "Function Execution", 30.77% (8/26)) and communication overhead between serverless functions (i.e., "Function Communication", 69.23% (18/26)). The

function execution overhead represents the time of executing the actual functionality of serverless functions, not the end-to-end response time of a request. For the function communication overhead, it is due to the stateless feature of serverless functions. Current serverless platforms do not provide a mature point-to-point communication mechanism between serverless functions via the network. Generally, developers leverage cloud storage services like AWS S3 [11] to save data of serverless functions for intermediary state management. In this situation, serverless functions need to keep fetching the required data in external services to accomplish the collaboration of functions, thus producing a high-latency communication overhead.

Second, in our taxonomy, the second-largest research direction in proportion is "Programming Framework" (19.51% (32/164)). Serverless computing is an emerging cloud computing paradigm, and thus it may not fully support applications with unique type requirements or no type requirements. Therefore, related researchers have designed specific (i.e., "Specific Framework", 59.38% (19/32)) programming frameworks or generic programming frameworks (i.e., "General Framework", 40.63% (13/32)) to adapt to the corresponding requirement.

Third, our taxonomy shows that studies related to "Resource Management" account for the third-largest percentage, i.e., 13.41% (22/164). In serverless computing, resource management aims at managing the resource requirements of serverless applications and ensure the resource efficiency of serverless platforms. Better resource management will more easily make serverless platforms and applications achieve service-level agreements.

In addition, our taxonomy contains eight other research directions: "Stateful FaaS" (4.88% (8/164)), "Application Modeling" (4.27% (7/164)), "Application Migration" (6.10% (10/164)), "Cost" (8.54% (14/164)), "Multi-Cloud Development" (1.83% (3/164)), "Accelerator Support" (2.44% (4/164)), "Security" (3.05% (5/164)), and "Testing and Debugging" (1.22% (2/164)). The specific illustration is as follows. "Stateful FaaS": Serverless functions are developed in a stateless way. However, supporting the stateful feature in serverless applications can fulfill a broader range of workloads. Therefore, some efforts have aimed at designing stateful serverless computing. "Application Modeling": Serverless applications follow a new development paradigm. Therefore, it may be limited in the expression of the serverless application. Some studies have addressed the application modeling problem to clearly show the specific representation and dependency for serverless applications. "Application Migration": The prevalence and popularity of serverless computing make more and more legacy applications migrate to the serverless platform to enjoy its advantages like low cost and high scalability. This migration process corresponds to the application migration research direction of the serverless computing literature. "Cost": Major serverless platforms adopt a unique cost pattern, i.e., utilization-based billing. The cost is closely related to the execution time of the serverless function, allocated or consumed memory size, and the number of invocations. Some researchers have explored how to predict and optimize the cost of serverless functions or applications based on these factors and possible impact factors. "Multi-Cloud Development": Generally, software developers select a fixed serverless platform to develop their applications. However, different cloud providers offer various features for their serverless platforms. Using multiple clouds will allow application development and execution to enjoy more benefits. Therefore, some studies have made efforts toward the multi-cloud development of serverless computing. "Accelerator Support": Serverless platforms mainly provide the CPU resource-dominated runtime environment. Therefore, software developers' applications cannot use other accelerators like **Graphics Processing Unit (GPU)** and **Field Programmable Gate Arrays (FPGA)** hardware resources. However, more and more tasks like machine learning and deep learning require leveraging such hardware resources to accelerate their performance. In this situation, how to design accelerator-enabled serverless computing is essential to facilitate wide application availability. "Security": When using serverless computing, the serverless platform and application details are

agnostic to each other. This situation also leads software developers and cloud providers to distrust each other. Some security problems have been investigated in the serverless computing literature. "Testing and Debugging": Similar to traditional software applications, serverless-related developers are also required to test and debug their serverless applications. Testing and debugging play a vital role in the software quality assurance of serverless applications. However, serverless computing hides the underlying system implementation for software developers. Moreover, serverless functions contained in applications are independent and event-driven. These features make testing and debugging challenging so that developers cannot determine the application's correctness.

6 RQ2 (EXISTING SOLUTIONS)

To answer the research question of how existing studies tackle specific problems for each research direction, the first two authors read the entire content of the research article over and over again and then write the research summary together. The first two authors classify related studies of each research direction according to summaries. Next, we elaborate on the solutions to problems associated with each research direction.

6.1 Resource Management

In serverless computing, resource management is responsible for allocating the proper resource provision for serverless functions and scheduling serverless functions on appropriate function instances. This process guarantees software developers' and cloud providers' **quality of service (QoS)** requirements. The QoS goal of software developers is related to the serverless application's performance, cost, security, and so on. In contrast, the QoS goal of cloud providers is associated with the serverless platform's resource utilization, load balancing, throughput, and so on. Existing studies have tried to improve resource management from four kinds of solutions, including platform QoS requirement, dynamic resource adjustment, application characteristic, and architecture design. A summary of studies on resource management is shown in Table 3.

• **Platform QoS requirement:** From the cloud provider's perspective, achieving efficient resource utilization is critical in reducing resource waste. Therefore, some studies have considered managing the resource by measuring the QoS of the serverless platform. A QoS-aware resource manager can satisfy the QoS enforcement while maximizing the overall resource utilization of the serverless platform. HoseinyFarahabady et al. [136, 137] leveraged the model predictive controller to present the QoS-aware controller with feedback to guarantee the well-utilization of computation resources, the response time of processing events, and the QoS demand level for serverless functions. However, their studies have not considered the effect of the number saturation of the working thread. To address this problem, they presented a controller that can adjust the number of working threads for each QoS class [154]. This controller used a QoS violation index to determine the required resources, not the prediction module of the previous work [136, 137].

Tariq et al. [245] first conducted a measurement study to uncover existing serverless platforms' problems. They found inconsistent and incorrect concurrency limits, difficulty supporting bursty workloads, and inefficient resource allocation. To alleviate these problems, they introduced an effective QoS scheduler called Sequoia to realize a variety of flexible policies. Yuvaraj et al. [272] and Schuler et al. [222] found that inconsistent and runtime limitations (such as scalability and concurrency) to present a resource management framework based on reinforcement learning.

Solutions based on the platform QoS requirement consider the overall serviceability of platforms. In the long term, serverless platforms can benefit from this kind of solution. However, for user experience, this solution relies on the factor granularity that may be coarse and does not adjust for task types of developers.

Table 3. A Summary of Studies on Resource Management

Study	Solution	Used strategy[a]	Considered factor[b]
HoseinyFarahabady et al. [136, 137]	Platform QoS requirement	Prediction model	Resource utilization
Kim et al. [154]	Platform QoS requirement	QoS violation index	Number saturation of working threads
Tariq et al. [245]	Platform QoS requirement	Flexible policies	Concurrency limits
Yuvaraj et al. [272] and Schuler et al. [222]	Platform QoS requirement	Reinforcement learning	Scalability/concurrency
Kim et al. [153] and Suresh et al. [240, 241]	Dynamic resource adjustment	Runtime data analysis	Performance requirement; Resource consumption and lifetime
Mampage et al. [182]	Dynamic resource adjustment	Heuristic algorithm design	Application deadline
Enes et al. [119]	Dynamic resource adjustment	Resource monitoring and feedback	Consumed resources
Yu et al. [266]	Dynamic resource adjustment	Experience-driven algorithm design	Resource saturation point
Saha et al. [216] and Das et al. [105]	Application characteristic	Prediction model	Function execution history, network latency
Bhasi et al. [87]	Application characteristic	Estimation of number of containers	Workflow graph and invocation probability
Related studies [149, 181, 199, 201]	Application characteristic	Heuristic rules	Execution deadline, expected scheduling policy, and performance goals
Zhang et al. [274]	Application characteristic	Step dependency model	Task dependency
Kaffes et al. [146]	Architecture design	Centralized core-granular scheduler	Performance stability and maintainability
Zhang et al. [279]	Architecture design	Load balancer	Harvest VMs

[a]The approach adopted or used by the given solution.
[b]The factor considered in the approach used.

• **Dynamic resource adjustment:** Efficient and flexible resource management can handle various serverless workloads with different resources and latency requirements. Moreover, it can minimize cloud providers' costs. Some studies have considered how to dynamically adjust resources like CPU in the serverless platform. A potential way is to analyze history runtime data. For example, Kim et al. [153] presented a fine-grained CPU cap controller to dynamically adjust CPU usage limit according to the performance requirement similarity of serverless applications. This adjustment can minimize resource contention, improve the robustness of the controller, and thus reduce performance degradation. Similarly, function-level schedulers [240, 241] were presented to analyze the resource consumption and lifetime of serverless functions and then classify and schedule serverless functions. Meanwhile, these schedulers can dynamically adjust the CPU-share resources of containers for serverless functions. In addition, Mampage et al. [182] considered the application deadline to adjust and manage the CPU resource and minimize the cloud provider's cost.

However, some applications may be complex since their consumed resources could vary with workload demands. Stronger resource management is essential for applications like big data analytics. Leveraging real-time, precise resource monitoring and feedback may be a possible solution. Enes et al. [119] presented such a real-time approach. This approach relied on the operating-system-level virtualization to dynamically change provided resources via cgroups and kernel-backed accounting features. In addition, Yu et al. [266] presented a new serverless resource manager to make full use of idle resources. This manager used an experience-driven algorithm to

estimate the resource saturation point for the serverless function and then dynamically decided to harvest or offer resources for this serverless function.

Dynamic resource adjustment can change fine-grained resource allocation in a real-time way, showing high flexibility. However, the above studies have mainly focused on CPU resource adjustment, lacking the management and adjustment for other resources. In practice, a serverless application has different CPU, memory, and I/O usage requirements. Considering various resources in resource management can further improve overall performance and utilization.

• **Application characteristic:** Some studies have considered application characteristics to design the corresponding resource management strategies. Application characteristics help to get the corresponding prediction models. For example, some studies [105, 216] have analyzed the function execution history and network latency of serverless functions to predict the right memory consumption or function placement.

A serverless application can be viewed as a workflow. Existing serverless platforms may launch multiple containers to serve multiple serverless functions contained in the workflow, causing resource over-provisioning. In this situation, Bhasi et al. [87] presented a resource management framework called Kraken, which captured the workflow characteristic and invocation probability to estimate the number of containers provided and ensure the application performance. An application workflow may also have a workflow execution deadline. A scheduling algorithm should consider this factor and meet the constraint cost. Therefore, serverless deadline-budget workflow scheduling algorithms [149, 181, 201] were presented. These algorithms used a set of heuristic rules to process the workflow graph and assign the corresponding resources. Palma et al. [199] was a novel scheduling idea. It provided a kind of declarative language for developers to describe the application requirements, such as expected scheduling policy and performance goals. The underlying scheduler can understand these requirements to select the appropriate instances.

The above studies have mainly maximized resource utilization. However, considering the billing model of serverless functions, resource management should also evaluate the task execution cost related to aggregated runtimes. Inter-task scheduling requires minimizing both cost and task completion time. It is hard to tradeoff between cost and completion time since a short completion time means large allocated memory that causes a high pricing unit. A fine-grained task-level scheduler, Caerus [274], was presented to address this problem by on-demand invoking functions. Caerus used a step dependency model to model and schedule pipeline-able and non-pipeline-able dependencies across tasks.

These studies have considered different application characteristics, including memory consumption, function execution time, workflow graph, function invocation probability, budget, and so on. These characteristics are captured by analyzing the history information or describing specific requirements. A concern is that there is an information gap between developers, applications, and providers. Some serverless platforms hide the complexity of dynamically changing infrastructure for developers, and the underlying changes are uncertain, affecting application features.

• **Architecture design:** Existing scheduling policies have been basically coarse-grained and may not suitably satisfy the bursty, stateless, and short-lived applications. In this situation, Kaffes et al. [146] presented a new scheduling architecture with a centralized core-granular scheduler. Core granular can directly assign functions to individual cores, guaranteeing performance stability. Centralized design can maintain a global view of the cluster to manage cores and resources and eliminate the migration of heavy functions. However, this scheduling architecture design considered only the CPU resource allocation.

In serverless platforms, they adopt the strategy of over-provision resources for serverless applications to guarantee the application QoS. However, developers still pay for only resources that their applications actually consume. This kind of resource provision way is not friendly for cloud

Table 4. A Summary of Studies on Performance Prediction

Study	Solution	Considered factor	Target object[c]
Cordingly et al. [102]	Regression model prediction	CPU time (e.g., CPU user mode time and CPU kernel mode time)	Serverless function
Eismann et al. [111]	Regression model prediction	Resource consumption (e.g., heap used, user CPU time, system CPU time, voluntary context switches, bytes written to file system, and bytes received over network)	Serverless function
Mahmoudi et al. [179]	Statistical learning prediction	Cold start rate, the arrival rate of warm instances, and instance expiration rate	Serverless platform
Mahmoudi et al. [178]	Statistical learning prediction	Cold start rate, the arrival rate of each server, and server expiration rate	Serverless platform
Gias et al. [128]	Statistical learning prediction	Number of functions, function popularity, arrival rate, service rate, cold start rate, and idle lifetime rate	Serverless platform
Akhtar et al. [65]	Statistical learning prediction	Configuration (e.g., allocated memory and location) and execution time	Serverless function
Lin and Khazaei [167]	Statistical learning prediction	Allocated memory, response time, and average number of invocations	Serverless application

[c]The target object of the given solution.

providers of serverless computing. In this situation, a new architecture may be required to maximize the cloud provider's benefits. Zhang et al. [279] leveraged Harvest VMs to design a new architecture. Harvest VMs is a faster, cheaper alternative than VMs. The authors leveraged a series of measurement results to illustrate the suitability of Harvest VMs in serverless computing. Meanwhile, these results also guided the architecture design with Harvest VMs.

Designing new schedulers or resource environments can solve the problem fundamentally. However, this kind of solution inherently requires extensive engineering efforts to modify the underlying resource management strategies. Moreover, problems with security mechanisms are also addressed additionally.

6.2 Performance

6.2.1 Performance Prediction. For studies related to performance prediction, regression model-based prediction and statistical learning-based prediction are two common solutions. A summary of studies on performance prediction is shown in Table 4.

• **Regression model prediction:** Some studies have tried to train the regression model about performance by considering several affected factors. Cordingly et al. [102] thought about the performance prediction problem from the perspective of system resource usage. They generated the regression model for each serverless function to predict its runtime performance (e.g., CPU user mode time, CPU kernel mode time). This model considered the CPU heterogeneity of the serverless platform and memory settings. However, this approach trained the corresponding model for each serverless function; thus, this process may take a long time. Therefore, Eismann et al. [111] presented an approach called Sizeless that did not require dedicated performance learning. Sizeless designed an offline phase, which monitored some synthetic functions and collected the resource consumption data (e.g., system CPU time, bytes received) for all memory sizes. Based on the

collected data, Sizeless trained the multi-target regression model to predict the execution latency in all memory sizes for a real function.

However, these approaches above targeted only independent serverless functions, not serverless applications with multiple serverless functions and complex structures. Moreover, obtained performance models did not consider function features such as task type and input parameter size.

• **Statistical learning prediction:** Another kind of approach is to consider statistical learning to predict the performance of serverless platforms, functions, or applications. Some studies [128, 178, 179] have aimed at predicting the performance of serverless platforms. Prediction models were presented to create performance-driven and predictive serverless platforms. These models considered some key parameters, such as the cold start rate, cold start latency, arrival rate of warm instances, and instance expiration rate, to the original system with the Markovian arrival process. Obtained platforms can decrease resource waste and guarantee the QoS of serverless applications.

Other studies have aimed at predicting the response time of serverless functions or applications. Akhtar et al. [65] presented a framework called COSE to learn a performance model of the serverless function from execution logs. This model used the Bayesian Optimization strategy to learn the gap between configuration and runtime statistically, and it predicted the optimal configuration that provided satisfying user-specified performance. Lin and Khazaei [167] modeled the application performance into a probabilistic graph considering the structure transformation as well as the runtime response time of each serverless function.

These above approaches leveraged the collected history runtime information to predict the function or application performance. In practice, it is impossible to know the historical performance situation for new serverless functions or applications.

6.2.2 Performance Optimization. Studies on performance optimization are related to optimizing the cold start performance and runtime performance.

6.2.2.1 Cold Start Performance. To address the cold start problem, serverless platforms like AWS Lambda adopt the fixed "keep-alive" policy to keep resources in memory for a few minutes, i.e., becoming warm instances, when serverless functions finish their executions. Subsequent requests can reuse warm instances with the required resources to reduce the number of cold starts. However, this policy cannot capture the actual invocation frequency of serverless functions, leading to resource waste in no requests. Therefore, many researchers have continued to tackle the cold start problem. The main solutions contain instance prewarm preparation, data cache-based optimization, function scheduling, snapshot-based optimization, and architecture design.

• **Instance prewarm preparation:** Instance prewarm preparation is to launch some required function instances in advance to serve the incoming requests. This kind of solution prevents the serverless application from going through the cold start process. Table 5 shows a summary of instance prewarm preparation studies on cold start performance. Specifically, some studies [107, 282] have focused on the cold start problem of serverless applications with the chain structure. The first serverless function contained in the chain-type serverless application is invoked, meaning that the following serverless functions will also be invoked in a cold start manner. Therefore, they designed the corresponding approaches to predict and prepare the runtime execution environment in advance for subsequent serverless functions contained in a chain-type serverless application.

In order to alleviate the cold start problem, some other studies [168, 239] have used a prewarm pool to process requests quickly. The prewarm pool can have containers with different resource configurations [168], or it can be adjusted according to the container runtime history over time [239]. However, these approaches may not quickly deal with burst requests and avoid resource waste. To further address these problems, Horovitz et al. [135] learned the seasonal invocation pattern of incoming functions through history invocation timestamps and then

Table 5. A Summary of Instance Prewarm Preparation Studies on Cold Start Performance

Study	Used strategy	Considered factor	Target object
Daw et al. [107] and Zuk et al. [282]	Predict the runtime environment	Application chain length	Serverless applications with the chain structure
Ling et al. [168]	Provide an oversubscribed prewarm container pool	Different resource configurations	Function requests
Suo et al. [239]	Maintain an adaptive live container pool	Container adjustment	Function requests
Horovitz et al. [135], Xu et al. [264], and Shahrad et al. [224]	Learn the seasonal invocation pattern or use time series prediction	History invocation timestamps or invocation frequency	Function requests

launched required instances for serverless functions in time. Similarly, Xu et al. [264] presented an adaptive strategy called AWU to predict when the serverless function will be invoked and then warm up the runtime execution environment in advance. AWU was based on time series prediction, which considered the invocation number of the serverless function over time and the moments between serverless functions being called.

In addition, Shahrad et al. [224] also presented a flexible and practical resource management policy. This policy can dynamically manage the prewarm time window size for each serverless function through the time series prediction. Unlike the time series prediction of the study [264], this policy [224] was based on each serverless function's invocation frequency and pattern to dynamically adjust the prewarm time and keep alive time.

Studies about instance prewarm preparation have learned specific characteristics, e.g., structure and invocation frequency, to design prewarm strategies. In these strategies, reducing resource waste becomes a key and important goal. Setting the window service size of the prewarm pool may be an effective method. However, researchers need to focus on specific functions that have a fixed execution frequency, guaranteeing the reduction of the number of cold starts. Therefore, it will be a challenge to set the service window size of the pool for other functions that are called irregularly.

● **Data cache-based optimization:** The traditional runtime environment of serverless computing is VMs, which have strong isolation and flexibility. However, VMs lack the advantage of low start latency for applications, and this latency is usually greater than 1,000 ms. In this situation, serverless computing uses another common runtime environment, containers, which can achieve low start latency (usually 50 ms–500 ms) than VMs. Moreover, containers typically include binaries and libraries. Pre-importing necessary or commonly used data on such a runtime environment can speed up the cold start process. Table 6 shows a summary of data cache-based optimization studies on cold start performance.

The representative studies based on data cache-based optimization are SOCK [197] and SAND [67]. Specifically, SOCK [197] and its previous implementation, Pipsqueak [196], cached interpreters and commonly used libraries in containers, and provided the lightweight isolation mechanism for serverless functions. Following this caching idea, Nuka [211] was a generic serverless engine where a local package caching design is presented to import required software packages. SAND [67] applied the application-level sandbox runtime sharing to reduce the number of containers and thus container preparation latency. SAND also provided the isolation mechanism for serverless functions to allocate or deallocate resources quickly. Similar to SAND, Dukic et al. [110] thought that current strict isolation is not necessary for safe concurrent requests of the same serverless function. Sharing the runtime may be a possible solution to alleviate the cold start problem for concurrent requests. Therefore, they presented an ultralightweight execution context named

Table 6. A Summary of Data Cache-based Optimization Studies on Cold Start Performance

Study	Used strategy	Considered factor	Target object
Oakes et al. [196, 197] and Qin et al. [211]	Add library-related data in runtime environments	Commonly used data	Interpreters and (software) libraries
Akkus et al. [67] and Dukic et al. [110]	Apply the sandbox runtime sharing	Isolation mechanism	Application or function runtime
Mohan et al. [188]	Add networking resources in pre-created containers	Creation and initialization of network namespaces	Networking resources
Yan et al. [265]	Add image-related data in runtime environments	Repeated fetching elimination	Memory and local disk (image data)
Solaiman et al. [232] and Fuerst et al. [125]	Design the container-based approach	Runtime information of requests	Containers

Photons, which can co-locate multiple function instances of the same serverless function in the same runtime via workload parallelism. However, the above studies on data cache-based optimization may be impractical since caching all required libraries or functions in memory will increase resource overhead, and cache policies are not easy to capture in the real workload.

Some related studies have designed other data cache approaches considering different objects, e.g., networking resources, image data, and containers. For example, Mohan et al. [188] found that the major overhead during container startup for concurrency invocations is the creation and initialization of network namespaces. Therefore, they cached networking resources in some pre-created containers to reduce the network overhead of container startup. Hermes [265] was a two-level caching mechanism including memory caching and local disk caching to support on-demand loading of image data and repeated fetching elimination. WLEC [232] was a container-based caching policy, which used cold, warm, and template queues to place required containers according to runtime information. These containers were managed and selected by a Container Management Service to respond to requests.

Except for considering different cache objects, Fuerst et al. [125] mapped the keep-alive of serverless computing to caching study, where keeping instances warm is viewed to cache an object, and the warm start is a cache hit. They designed FaasCache containing a set of caching-based keep-alive policies to reduce cold start overhead.

In most cases, using the data cache can effectively alleviate the cold start problem. However, this kind of approach will introduce high resource overhead and imbalanced resource consumption, leading to performance interference from the system environment. Moreover, cache policies may be hard to be determined in the real-world.

• **Function scheduling:** Generally, the function scheduling solution is to distribute incoming requests to warm instances to serve them. Warm instances have the prepared runtime execution resources, and thus they can serve requests faster than cold starts. Table 7 shows a summary of function scheduling studies on cold start performance. Related studies have aimed at improving the existing schedulers or compensate for the insufficiency of scheduling strategies in reusing warm containers. First, existing schedulers may be agnostic of container lifecycle, i.e., creation, use, pause, or eviction of containers, and thus they are ineffective in reducing cold starts. In this situation, Wu et al. [262] proposed a container lifecycle-aware scheduler called CAS to distribute requests. This scheduler leveraged an affinitive worker to maintain and manage the states of containers. Second, existing schedulers showed erratic performance behavior in application workloads with multi-tenant and high concurrency. Therefore, Kim et al. [150] designed a novel scheduling algorithm called FPCSch, allowing serverless platforms to schedule available containers into requests to obtain stable performance while reducing cold starts.

Table 7. A Summary of Function Scheduling Studies on Cold Start Performance

Study	Used strategy	Considered factor	Target object
Wu et al. [262]	Leverage an affinitive worker to maintain and manage the states of containers	Container lifecycle	Functions
Kim et al. [150]	Design a scheduling algorithm	Performance behavior	Functions
Related studies [85, 159, 227]	Present a choreography middleware, design a function fusion solution, or mine frequent dependencies	Application structure and dependency relationships	Functions
Related studies [62, 75, 160]	Distribute requests into instances with pre-loaded cache	Cache hit rate	Functions

Other researchers have considered the application structure [85, 159] or dependency relationships [227] among serverless functions to schedule serverless functions. If independent serverless functions can be fused or composed into a serverless function, the scheduler assigns these independent serverless functions to be executed on the same function instance. In this situation, the number of cold starts will directly decrease. Bermbach et al. [85] and Lee et al. [159] leveraged this knowledge to reduce the number of cold starts. In addition, Shen et al. [227] used a frequent pattern mining approach and invocation history to find dependencies between serverless functions. These dependencies can guide the scheduler to schedule the connected functions on the same instance, thus diminishing the occurrences of cold starts.

Considering that some studies like SOCK [197] and SAND [67] used data cache techniques to alleviate the cold start problem, the scheduler or load balancing may significantly affect the cache-hit ratio, thus influencing the performance of cold start and overall application. Based on it, some scheduling algorithms like PASch [75], GRAF [160], and others [62] have been presented to distribute incoming requests into container instances with pre-loaded cache to improve the cache hit rate and thus the performance of serverless applications.

Using function scheduling approaches can make requests execute function instances in the warm mode, which reduces the number of cold starts. Moreover, this kind of approach makes full use of idle resources. However, it is not clear whether the presented approaches can handle dynamic workloads or other domains.

● **Snapshot-based optimization:** The industry and academia have used the snapshot way, a promising solution, to alleviate the high latency of cold starts. Specifically, this way captures the complete state (called snapshot) of the current function execution and stores the state in local storage (e.g., SSD) or in a disaggregated storage service. When the same function is invoked again, the serverless platform can quickly initialize the function instance according to the corresponding snapshot to immediately process this request. Snapshot-based optimization becomes attractive since it does not require main memory during functional inactivity and can reduce the high latency of cold starts. Table 8 shows a summary of snapshot-based optimization studies on cold start performance.

Cadden et al. [92] presented SEUSS to capture the function state (containing function logic, language interpreter, and libraries) at an arbitrary point during executions. Moreover, SEUSS saved the state as an in-memory snapshot. New invocations of the same function can be rapidly started from its snapshot. A similar idea to SEUSS, Replayable Execution [252] was to use process-level checkpointing to save and restore the state. Moreover, the state was allowed to share between different containers. Du et al. [109] found that sandboxes have a high application initialization latency, which dominated the overhead of cold start latency. To reduce this part of overhead, they

Table 8. A Summary of Snapshot-based Optimization Studies on Cold Start Performance

Study	Used strategy	Considered factor	Target object
Cadden et al. [92]	Capture function logic, language interpreter, and libraries	Function execution state saving	Same functions
Wang et al. [252]	Use process-level checkpointing	Function execution state sharing	Same functions
Du et al. [109] and Ustiugov et al. [246]	Capture sandbox runtime to minimize the critical path processing time	Application initialization latency reduction	Applications or functions
Agache et al. [64]	Capture the state of the virtual machine monitor and the emulated devices	Fast runtime environment	VMs

designed Catalyzer based on Google's gVisor [39] to set checkpoints and restore application and sandbox runtime. Catalyzer minimized the critical path processing time of VM loading through the snapshot way. Following the same design principles as Catalyzer, a runtime environment called Firecracker [64] was presented by AWS. Its Firecracker VM was loaded from a snapshot, which contained the state of the virtual machine monitor and the emulated devices. Moreover, Firecracker customized the virtual machine manager (e.g., hypervisor) to create microVMs to isolate multiple tenants with affordable overhead. However, Ustiugov et al. [246] characterized the snapshot-based serverless infrastructure Catalyzer [109] and found that a function executed from the snapshot took 95% longer to execute than if the same function was executed from a resident in memory, on average. Due to the state constantly being written to the page, the page frequently generates faults, thus causing high latency. Fortunately, they found the same stable working set of pages among different invocations of the same function. Therefore, they leveraged this insight to present REAP. REAP can obtain the function's stable working set of guest memory pages and pre-load them into memory to speed up the performance.

Using the snapshot-based optimization approach speeds up the function start time, avoiding creating new runtime environments from scratch. However, they also pose shortcomings. For example, generating a stable working set (for the work [246]) is not enough when the input data payload is significantly different among invocations. In this situation, the runtime environment waited until all data was loaded to start, leading to a slower execution latency.

• **Architecture design:** Some studies have used novel design principles or underlying environments to present the new serverless platform, which will fundamentally alleviate the cold start problem. We briefly summarize the related studies as shown in Table 9. Specifically, Boucher et al. [89] presented a novel serverless platform based on language-based isolation. Using language-based isolation is faster than using process-level isolation for microsecond-scale serverless functions. In language-based isolation, a single-threaded worker process is hosted in each worker core, and it can directly execute serverless functions, one at a time. Moreover, the presented serverless platform used task preemption, supported by commodity CPUs at a microsecond scale. Other studies have applied lightweight runtime. Hall and Ramachandran [132] and Long et al. [173] used the lightweight and fast WebAssembly runtime [58] to replace the container to remedy the performance overhead of cold starts. In contrast to VMs and containers, WebAssembly is a high-level language VM runtime and shows a binary format with inherent memory and execution safety guarantees.

Unikernel [55] is an emerging fine-grained, lightweight sandbox that uses libarayOS with essential dependency libraries. Security of the Unikernel is higher than containers, the image size is also small, and notably, the start latency is within 10 ms. USETL [121] was a Unikernel-based

Table 9. A Summary of Architecture Design Studies on Cold Start Performance Optimization

Study	Used strategy	Considered factor	Target object
Boucher et al. [89]	Use the faster than traditional process-level isolation	Language-based isolation usage	Architecture
Hall and Ramachandran [132] and Long et al. [173]	Use the lightweight high-level language VM runtime	WebAssembly usage	Architecture
Fingler et al. [121] and Tan et al. [243]	Use lightweight sandbox with libraryOS and has strong flexibility	Unikernel usage	Architecture

design to be specific to the cold start performance of serverless **extract, transform, and load (ETL)** workloads. USETL leveraged strong language preference and maintained a pool of Unikernels with initialized runtime. Following the Unikernel idea, Tian et al. [243] also used the Unikernel design to make serverless functions run in the Unikernel, offering extremely low cold start latency to execute functions. Moreover, this design leveraged a hardware technique named VMFUNC [57] to achieve communication among serverless functions in different Unikernels.

The above architecture designs and containers have different characteristics, in terms of security, performance, and flexibility. Using a lightweight design with good security, performance, and flexibility becomes an inevitable trend in future serverless computing. Probably, the next effort will focus on how to generalize such a design to commodity and open-source serverless platforms.

6.2.2.2 Runtime Performance. Studies on runtime performance optimization are about performance optimizations of function execution and function communication.

1. Function Execution

In the related studies on performance optimization of function execution, memory configuration, function scheduling, and architecture design are common solutions. Table 10 shows a summary of these studies.

● **Memory configuration:** Generally, the response time of the serverless function is affected by the allocated memory [258, 268]. However, when the allocated memory size is large enough, the response time of the serverless function becomes insensitive to the memory [167]. Therefore, changing the size of the allocated memory may be an opportunity for performance optimization. Lin and Khazaei [167] used their performance model with a heuristic algorithm to find a memory configuration. This configuration can achieve the minimum average response time of the serverless application under budget constraints.

● **Function scheduling:** Designing an appropriate scheduling strategy can reduce resource competition for CPU and network among function instances, thus improving the execution performance of serverless applications. Current serverless platforms are agnostic to the application type or invocation frequency during the request processing. It may make certain serverless functions be located on the same VM node, influencing their execution performance. Mahmoudi et al. [180] and Przybylski et al. [206] analyzed the collected information or the application's profile to the corresponding function scheduling. When there are bursty or real-time workloads, a real-time serverless prototype is required to execute them at a guaranteed invocation rate and minimal performance effect. Such a prototype can be achieved through predictive container management and admission control [195]. However, the Alibaba Cloud Function Compute team found that the deployment way to use custom container images needs to pull large container images (larger than 1.3 GB) from the backend store. When bursty workloads are to pull the same large container image, it will cause a severe performance problem of network bandwidth. Therefore, they presented a rapid

Table 10. A Summary of Studies on Function Execution

Study	Solution	Used strategy	Considered factor
Lin and Khazaei [167]	Memory configuration	Use a performance model with a heuristic algorithm	Allocated memory size
Mahmoudi et al. [180]	Function scheduling	Use statistical machine learning to analyze resource utilization and application's profile	Application type
Przybylski et al. [206]	Function scheduling	Analyze the collected information	Invocation frequency and function duration time
Nguyen et al. [195] and Wang et al. [250]	Function scheduling	Use predictive container management or design a rapid container provisioning	Real-time workloads, container image size
Singhvi et al. [230]	Architecture design	Redesign the control and data planes of the architecture	The problem between sandbox and scheduling
Chadha et al. [96]	Architecture design	Use a Just-in-Time compiler based on LLVM	Execution cost of compute-intensive functions
Carreira et al. [93]	Architecture design	Share code optimization information across runtimes	Runtime knowledge (hot starts)

container provisioning FaaSNet [250] to accelerate container provisioning to serve bursty requests. FaaSNet organized VMs as function-based tree structures and used a tree balancing algorithm to dynamically adapt the tree topology to adjust VM joining and leaving.

• **Architecture design:** Some new architectures have been designed to optimize the function execution performance. Atoll [230] was a scalable, low-latency serverless platform where the control and data planes of its architecture were redesigned via decoupling sandbox allocation from scheduling, introducing deadline-aware scheduling, and co-designing the load balancing and scheduling layers. Atoll can handle short-lived serverless functions with unpredictable arrival patterns and maximize the request processing with the user-specified deadline. In addition, instead of using traditional pre-compiled libraries, Chadha et al. [96] used a Just-in-Time compiler, which is based on LLVM [43] for Python, to optimize the performance of compute-intensive serverless functions. In addition, Carreira et al. [93] tried to utilize runtime knowledge to optimize the executed function code. First, they demonstrated the significant impact of runtime optimizations. Moreover, the code to execute multiple times (i.e., in hot starts) was optimized code, which is executed at the maximum performance. Then, they presented a holistic system named Ignite to share code optimization information across runtimes for the same function to reduce profiling and compilation overheads.

For the performance optimization of function runtime, three kinds of solutions have been presented, i.e., memory configuration, function scheduling, and architecture design. The implementation of memory configuration is relatively easy because researchers can directly try to adjust the allocated size. Designing new architectures require huge efforts or changes for researchers. However, these designs may significantly improve the performance of function runtime.

2. Function Communication

Since serverless functions are stateless and are executed in different function instances, they generally rely on external storage to communicate with each other to accomplish complex tasks. However, such communication collaboration introduces additional overhead for application execution. Some studies have aimed at alleviating this overhead through different strategies: memory sharing, cache-based design, storage optimization, and network optimization. Table 11 shows a summary of studies on function communication.

• **Memory sharing:** When serverless functions are co-located on the same machine, using memory sharing can avoid the overhead of data movement. Generally, memory sharing-based

Table 11. A Summary of Studies on Function Communication

Study	Solution	Used strategy	Considered factor
Related studies [104, 142, 215]	Memory sharing	Co-locate multiple requests on the same container	Share the same memory in a container
Shillaker and Pietzuch [229]	Memory sharing	Leverage WebAssembly-based isolation to execute functions within the same process	Specific assumptions
Kotni et al. [156]	Memory sharing	Leverage simple load/store instructions to data sharing	Execute serverless functions as threads
Related studies [236, 244, 261]	Cache-based design	Leverage the additional object caching layer of the computation node to save the same data	Link the caching layer to the computation node
Lykhenko et al. [175]	Cache-based design	Combine a multi-site cache with the storage layer	Get a rapid data consistency
Mvondo et al. [191]	Cache-based design	Design an opportunistic RAM-based caching system to predict caching efficiency and memory usage	Guarantee strong consistency and persistence
Romero et al. [214]	Cache-based design	Provide an in-memory caching layer in applications	Guarantee scaling
Nadgowda et al. [192]	Storage optimization	Present a new storage system with the data de-duplication operation	Reduce the function redundancy activation near the event sources
Related studies [155, 177, 208]	Storage optimization	Implement the multi-tier storage solution	Consider multi-tier storage, multi-tier cost, or multi-tier data passing methods
Sampe et al. [219]	Storage optimization	Design a data-driven middleware to improve data locality	Incorporate computation into the data pipeline
Zhang et al. [277]	Storage optimization	Present low-latency cloud storage to directly interact with the required data	Embed small storage functions into the storage
Wawrzoniak et al. [255]	Network optimization	Use the TCP hole-punching technique of P2P	Bypass the network constraints in conventional TCP/IP network stack

approaches [104, 142, 156, 215, 229] refer to designing and managing shared memory to achieve low-latency communication between serverless functions. Specifically, Shillaker and Pietzuch [229] presented a runtime called Faasm, which contained a WebAssembly-based isolation abstraction to isolate the memory of running functions and allow memory to be shared among functions in the same address space. However, Faasm was based on specific assumptions about the language-based isolation and application programming interface. Jia et al. [142] presented a runtime named Nightcore to address the overhead imposed by interactive serverless functions. Nightcore supported arbitrary invocation patterns to execute multiple invocation requests of the same function in the same container. Moreover, it optimized I/O between requests via efficient threading. Overall, the internal function call of Faasm [229] and Nightcore [142] had the same functionality. Differently, Faasm made serverless functions execute within the same process leveraging WebAssembly-based isolation. Nightcore directly co-located multiple requests on the same container and used shared memory to achieve efficient function communication. However, Faasm and Nightcore were both to modify the serverless application. In this situation, Faastlane [156] was presented to minimize the latency between function interactions, supporting unmodified

applications. Its data sharing leveraged simple load/store instructions, and Faastlane made serverless functions contained in a workflow execute as threads within a shared virtual address space.

• **Cache-based design:** Some approaches have used caches to improve state consistency guarantees and runtime performance of serverless applications. Lambdata [244] was a novel serverless system that allowed developers to declare the data read and write intents of their serverless functions. To speed up the communication performance, Lambdata leveraged the additional object caching layer of the computation node to save the same data for multiple function invocations and scheduled the related functions to the same computation node to reuse data. Linking the caching layer to the computation node was also adopted by Wu et al. [261]. They presented HydroCache to optimize network traffics. HydroCache chose an autoscaling key-value storage engine, Anna [12], to provide low-latency data access and transactional causal consistency. However, the worker of HydroCache was established multiple times with the storage to obtain the latest version, leading to the tail latency of function execution. To address this problem, Lykhenko et al. [175] proposed a solution called FaaSTCC, combining a multi-site cache with the storage layer. The key idea was to leverage a small amount of metadata that is passed from function to function to capture the difference between versions.

Similar to HydroCache, Sreekanti et al. [236] introduced a stateful serverless platform named Cloudburst, which also used Anna and imported a local storage cache to execute multiple serverless functions. However, HydroCache and Cloudburst introduced specific assumptions, such as consistency semantics and protocols, since they relied on Anna. Moreover, the work of Cloudburst did not discuss the specific caching configuration. A work designed at the same time as Cloudburst was OFC [191]. OFC designed an opportunistic RAM-based caching system to dynamically predict caching efficiency and memory usage through machine learning. OFC had stronger consistency and persistence guarantees than Cloudburst, and it supported more widely workload types. However, these approaches have not considered another critical factor, i.e., scaling, which may mitigate the data access impact. Therefore, Romero et al. [214] presented FaaT, which was serverless and bundled in the serverless application as an in-memory caching layer. In FaaT, different cache replacement and persistence policies were designed to achieve auto-scaling and transparency.

• **Storage optimization:** Since the data of serverless functions are generally stored in external storage, characteristics (e.g., data correctness, scalability, and efficiency) of the storage are essential to cooperate serverless functions. Some studies have aimed at storaging optimization to present new storage systems. Sanity [192] was a storage system to tackle the limitation of duplicated data. Data de-duplication operation was performed close to the event sources to reduce the function redundancy activation. To use an elastic and distributed data storage, Klimovic et al. [155] presented a novel storage system named Pocket to access data and automatically scale with the serverless application under desired performance and cost. Pocket can dynamically adjust resources and provide low latency, high throughput, scalable resources, and smart data placement across multi-tier storage, such as DRAM, Flash, and disk. Locus [208] used a mixture of fast but expensive storage and slow but cheap storage to balance the communication performance and cost. Overall, Pocket and Locus were both to implement the multi-tier storage solution to improve the performance and cost-efficiency of serverless workloads. A data-driven middleware Zion [219] for object storage was presented to improve data locality and reduce communication latency. Zion incorporated computation into the data pipeline and executed it in a scalable way to address the storage's scalability and resource contention problems. Besides incorporating computation into the data pipeline, embedding small storage functions into the storage was also noticed by Zhang et al. [277]. They presented low-latency cloud storage, Shredder, for serverless function chains. Developers can embed some small storage functions in the storage to directly interact with the required data. These storage functions can mitigate the network overhead between the serverless application and data.

Mahgoub et al. [177] compared different data passing methods, including VM-Storage, Direct-Passing, and state-of-practice Remote-Storage. The results showed that these methods performed poorly in all serverless scenarios. Therefore, they presented a management layer called SONIC to deploy a hybrid approach about three data passing methods to improve the application performance. A unified API was exposed to developers, allowing them to select the optimal data-passing method according to application-specific factors, such as input payloads. SONIC also can leverage optimized storage like Pocket [155] and Locus [277] in its selection methods.

• **Network optimization:** In practice, current serverless functions cannot directly communicate with each other via the network. To address this problem, Wawrzoniak et al. [255] presented a system called Boxer to support direct function-to-function communication in the existing serverless platform. Boxer used the TCP hole-punching technique of P2P to bypass the network constraints in conventional TCP/IP network stack. Such a system can directly achieve direct data exchange to benefit serverless computing.

There are various approaches to address the performance optimization of function communication. The evaluation results of these approaches show the effectiveness of performance enhancement. However, these approaches are based on external storage to archive the function interaction. This way inevitably produces additional data transmission overhead. Implementing direct communication is a challenging task, since the existing work [255] did not yet support executing large-scale communication-intensive applications and providing high-throughput and low-latency networks. Therefore, more efforts may be needed in the future to enable direct communication and maintain high scalability.

6.3 Stateful FaaS

In the research direction of stateful FaaS, some studies have tried to address it by considering application model design, log-based mechanism, and architecture design. Table 12 shows a summary of studies on stateful FaaS.

• **Application model design:** Mainstream cloud providers have rolled out their serverless orchestration services. In fact, these services play a kind of glue to compose serverless functions together. For example, AWS Step Functions [19] uses the state machine to combine multiple serverless functions in different patterns (e.g., sequential or parallel executions) and provides the state store in this serverless orchestration service. Microsoft Azure used durable functions [59] as the extension of serverless functions to allow developers to add the state and establish communication. In academia, Akhter et al. [66] presented a high-level application model to help developers develop and deploy their stateful applications. This model can transfer serverless functions and the required data as a stateful dataflow graph, and the data can be shared automatically to achieve scalability and low latency. Another high-level application model with an extension stateful function language and a compiler [91] was developed to help software developers generate cloud infrastructure supporting the state access of the serverless application.

• **Log-based mechanism:** To build serverless stateful functions on existing serverless platforms, Zhang et al. [273] were inspired by the log-based fault tolerance protocol to propose Beldi. Beldi introduced new refinements to the existing log-based fault tolerance approach in terms of data structure and specific algorithms. These refinements can record the application state and logs. Moreover, Beldi periodically re-executed functions that have not yet finished executing, guaranteeing the at-least-once execution semantics to prevent duplicated operation execution. In addition, Beldi's design motivated Boki's shared log approach [141] for stateful serverless computing. Boki exported the shared log API to serverless functions to store the state. Moreover, read and write paths were separated and individually optimized. However, developers need to adapt the use way

Table 12. A Summary of Studies on Stateful FaaS

Study	Solution	Used strategy	Considered factor
Akhter et al. [66]	Application model design	Transfer functions and data as a stateful dataflow graph	Share data in graphs
Brand et al. [91]	Application model design	Use an extension stateful function language and a compiler	Generate cloud infrastructure supporting the state access
Zhang et al. [273]	Log-based mechanism	Refine the log-based fault tolerance approach	Record the application state and logs and guarantee the at-least-once execution semantics
Jia et al. [141]	Log-based mechanism	Use the shared log to store the state	Use specific APIs to develop applications
Sreekanti et al. [236]	Architecture design	Leverage the additional object caching layer of the computation node	Link the caching layer to the computation node
Barcelona et al. [81, 82]	Architecture design	Map invocations as threads and built the shared object layer	Guarantee strong state consistency and simplify the global state semantics across threads
Sreekanti et al. [235]	Architecture design	Introduce a fault-tolerant shim layer for stateful serverless functions	Guarantee strong fault tolerance

of Boki with shared logs to write their applications since Boki-based development is different from the original development of serverless applications.

• **Architecture design:** Some studies have designed new architectures for serverless computing to support stateful applications. Cloudburst [236] was a specified architecture that used Anna storage with flexible auto-scaling. Crucial [81, 82] was also a new design for providing fine-grained state management. The invocation of a serverless function was mapped to a thread, and a distributed shared object layer was built on the in-memory data store. This way can guarantee strong state consistency and simplify the global state semantics across threads. Since the Beldi solution [273] offered the at-least-once execution semantics for failed function attempts, it may expose fractional writes. Therefore, Sreekanti et al. presented AFT [235] with stronger fault tolerance. AFT demanded servers to interpose and coordinate all database accesses and introduced a fault-tolerant shim layer for stateful serverless functions. Moreover, it used built-in atomicity and idempotence guarantees to provide exact-once semantics with safety and liveness in failures. However, AFT modified the server running so that it was limited to other platforms.

The key to implementing stateful FaaS is how to manage the state and keep the state consistent. Based on it, developers write stateful serverless applications without having to worry about concurrency control and fault tolerance. In this situation, most related studies have been based on strongly consistent databases. However, there is concern about whether the state of different applications is stored in a centralized database or different databases. If using a centralized database, researchers need to ensure that a malicious request from one application cannot observe the state of another. If using different databases, state isolation and consistency issues need to be addressed.

6.4 Application Modeling

This research direction refers to modeling the serverless application to deepen the understanding of software application design and reflect the changes in high-level abstraction or low-level implementation. There are two kinds of solutions: specification-based analysis and configuration-based analysis. Table 13 shows a summary of studies on application modeling.

• **Specification-based analysis:** It is helpful for application modeling to leverage certain specifications [157, 209, 228, 263]. For example, F(X)-MAN [209] was a model considering services

Table 13. A Summary of Studies on Application Modeling

Study	Solution	Used strategy	Considered factor
Chen et al. [209]	Specification-based analysis	Use a model considering services and connectors as entities	Compose atomic or composite services
Wurster et al. [263] and Yussupov et al. [271]	Specification-based analysis	Use the standard topology, or add business process model and notation	Capture the lifecycle of application provision and management, or control flow
Kritikos et al. [157]	Specification-based analysis	Propose the extension of the cloud modeling language	Cover all application lifecycle aspects
Obetz et al. [198]	Configuration-based analysis	Analyze events associated with serverless functions	Construct a new type of call graph for the serverless application
Samea et al. [217]	Configuration-based analysis	Analyze the configuration and introduce a model-driven approach	Transform the source model of applications into the low-level implementation

and connectors as entities. Different connectors (e.g., composition connectors, adaptation connectors, and parallel connectors) are composed of atomic or composite services. Considering the event-driven feature of serverless computing, Wurster et al. [263] used the standard **topology and orchestration specification for cloud applications** (**TOSCA**) to design an event-driven deployment modeling approach. However, TOSCA was incomplete since it did not contain the platform's requirement representation and could not support all application lifecycle aspects, such as deployment, requirement, and metric ones. Overall, there is no specification language to model the serverless application in a platform/cloud-independent manner. To address this problem, Kritikos et al. [157] proposed the extension of the cloud modeling language named CAMEL to specify serverless functions in an independent manner. Furthermore, Yussupov et al. [271] relied on TOSCA and **business process model and notation** (**BPMN**) to present a vendor-and technology-agnostic modeling method. BPMN captured the control flow composed of multiple activities, and this flow can be represented as a semantically-transparent graphical notation.

• **Configuration-based analysis:** In serverless computing, the configuration operation is essential for serverless applications since it represents some developers' requirements. However, the event-driven feature complicates serverless application modeling compared to traditional applications. Considering that some configuration information may be written in a configuration file like ".yml" [53], Obetz et al. [198] analyzed events associated with serverless functions in this configuration file to construct a new type of call graph for the serverless application. The call graph modeled the relationships between serverless functions, events, and used services. On the other hand, configuration changes may reflect the low-level implementation of the serverless application. To simplify the low-level implementation, Samea et al. [217] introduced a model-driven approach to automatically transform the source model of applications into the low-level implementation by analyzing the configuration.

In the related studies of application modeling, specification-based analysis and configuration-based analysis are two common approaches. These approaches can utilize the specific specification model or configuration files to model serverless applications. However, serverless applications have different characteristics from traditional software applications or cloud applications. For example, serverless functions are event-driven, and some cloud services are optionally used for communication between serverless functions. A standard programming model for serverless applications is lacking and needs to be proposed to represent this kind of emerging paradigm.
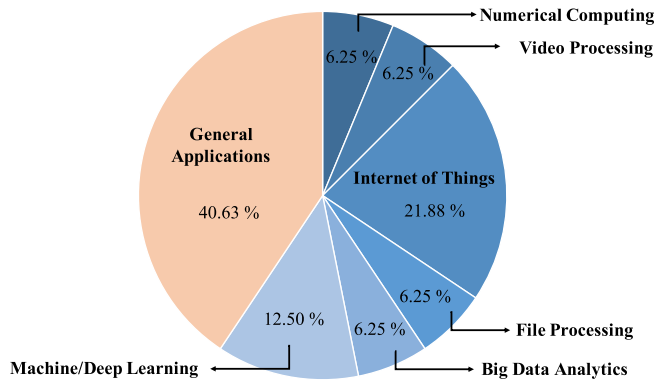
Fig. 5. The type distribution of programming frameworks presented by existing studies.

6.5 Programming Framework

Some limitations of serverless computing urge researchers to present new programming frameworks to serve specific applications and general applications. Figure 5 shows the type distribution of programming frameworks presented by existing studies. We find that designing frameworks for general applications is the most common, accounting for 40.63% of all frameworks. In frameworks of specific applications, the **Internet of Things (IoTs)** scenario is most widely investigated, accounting for 21.88% of all frameworks. Next, we introduce specific frameworks and general frameworks.

6.5.1 Specific Framework. Serverless computing has been applied to various specific scenarios, including numerical computing, video processing, the Internet of Things, file processing, big data analytics, and machine/deep learning. However, different scenarios may pose new challenges that need to be solved in the serverless computing paradigm or have some problems that leverage serverless computing to overcome further. A summary is shown in Table 14.

• **Numerical computing:** Traditional numerical computing tasks need scientists to manage infrastructure and maintain scalability in order to handle the varied resource parallelism. Serverless computing can free scientists from these burdens. Moreover, serverless computing shows a disaggregated data center pattern. In fact, this disaggregation makes linear algebra workloads with large dynamic requirements of memory and computation obtain benefits. However, the computation time of such workloads is the dominant overhead. In this situation, a serverless linear algebra framework named NumPyWren [226] was presented to address this problem. NumPyWren analyzed the data dependencies in the serverless application to extract the task graph that had potential parallel executions. Parallel functions can use an intermediate state in a distributed object store to speed up the processing latency. In practice, NumPyWren was implemented based on Py-Wren [144], which was a queue-based master-worker approach and processed serverless functions in parallel when possible.

• **Video processing:** Video processing workloads would invoke thousands of threads of execution in a few seconds. However, fine-grained parallelism did not support video encoders. In this demand, ExCamera [124] and Sprocket [74] were developed for serverless video processing. ExCamera achieved a high-parallel video encoder to handle chunks of video. Furthermore, ExCamera used a state machine for tasks to support fine-grained control and overcome the communication challenge. Sprocket was a scalable video processing framework to enable much more sophisticated video processing applications. Sproket transformed a single video input according to

Table 14. A Summary of Specific Framework Studies on Programming Framework

Study	Used startegy	Target object
Shankar et al. [226] and Jonas et al. [144]	Speed up the computation time of workloads with large dynamic requirements	Numerical computing
Fouladi et al. [124] and Ao et al. [74]	Support the high or dynamic parallelism	Video processing
Cheng et al. [99]	Provide the supportability of data moving for data-intensive IoT applications	Internet of Things
Bermbach et al. [84, 205] and Zhang et al. [276]	Address resource constraint and function placement on edge nodes	Internet of Things
Fortier et al. [122]	Implement function composition and communication	Internet of Things
Wolski et al. [260]	Simplify deployment operations	Internet of Things
Jindal et al. [143]	Implement data access over heterogeneous clusters	Internet of Things
Pérez et al. [203, 204]	Support all programming languages	File processing
Related studies [119, 129]	Support MapReduce tasks	Big data analytics
Wang et al. [251]	Reduce the training cost for ML training epoch	Machine/deep learning
Related studies [140, 267]	Address the deployment size limit for the large model	Machine/deep learning
Ali et al. [70]	Support batching and its associated setting parameters	Machine/deep learning

a user-specified program. Moreover, it supported dynamic task creation during processing through dynamic levels of parallelism.

• **Internet of Things:** The development paradigm of serverless computing could be beneficial for IoT applications. In the industry, AWS presented AWS IoT Greengrass service [15] to process data on edge. Azure designed Azure IoT edge service [24] to connect edge devices and serverless functions. In academia, some researchers also have focused on IoT frameworks. Specifically, the event-driven programming pattern and the separation of computation and storage make IoT applications inefficient. To address the supportability for data-intensive or dataflow IoT applications, Cheng et al. [99] proposed a functional programming model to allow the movement between code and data. However, considering that resources are constrained on edge nodes, Pfandzelter and Bermbach [205] designed a lightweight serverless framework, tinyFaaS, to adapt the edge feature. Specifically, tinyFaaS provided a specified endpoint with alternative messaging protocols for the communication of low-power devices. Moreover, their team also presented another approach called AuctionWhisk [84]. AuctionWhisk used the auction-inspired mechanism to control and arrange function locations across geo-distributed sites. A serverless teleoperable hybrid cloud system called STOIC [276] was presented to cooperate with edge clouds and public clouds. Dyninka [122] was designed to define and compose multiple serverless functions leveraging the multi-tier programming paradigm and compiler.

To simplify the deployment operation of IoT applications, a new IoT programming model, CSPOT [260], was designed to host services at device scale, edge scale, and cloud scale. CSPOT integrated various features, including multi-tier processing, robustness, compatibility, security guarantee, and record-and-replay debugging. To consider the platform heterogeneity, Jindal et al. [143] introduced an extension framework of FaaS to address the data access behavior over heterogeneous clusters via **Function Delivery Network (FDN)**. FDN can deliver the function to the appropriate cluster according to the required computation and data.

• **File processing:** Serverless computing allows developers to upload their files to the serverless platform. Triggered serverless functions handle them and return the processed output. However, early serverless platforms were limited to supporting all programming languages. Therefore, Pérez et al. [203, 204] introduced a highly-parallel event-driven programming model, which combined a middleware that can simplify the development and deployment process. Moreover, it used

customized runtime environments, i.e., container images, to bypass the language limitation. However, at the time our article was written, serverless platforms had provided the deployment way of the container image to support serverless applications written in any language. Compared with simply providing the source code to the serverless platform, using a custom container image has to follow specific interface requirements, introducing additional management efforts for software developers.

• **Big data analytics:** MapReduce is one of the most widely used programming models for big data applications. Giménez–Alventosa et al. [129] investigated the suitability that serverless computing was applied in MapReduce tasks. Moreover, they presented a new framework with high performance for MapReduce tasks. This framework automated the partitioning of input data according to the allocated memory. Similarly, for MapReduce jobs, Enes et al. [119] relied on operating-system-level virtualization technology to design a novel scalable framework to provide resources dynamically.

• **Machine/deep learning:** Serverless computing can provide flexible resource provisioning and easy-to-use deployment opportunity for **machine learning** (**ML**) applications. Moreover, the billing advantage of serverless computing is attractive to developers of machine learning. SIREN [251], a serverless programming framework for distributed machine learning, was presented to process data via serverless functions. SIREN leveraged deep reinforcement learning to design a novel serverless scheduler. This scheduler can dynamically adjust the assignment of function instances and memory sizes in each ML training epoch, reducing the training cost.

ML tasks generally generate and save the trained models in order to be used for the subsequent inference. However, some models may be larger than the deployment size limit of serverless computing, showing the infeasibility issue. To fill this gap, researchers presented Gillis [267] and AMPS-Inf [140] frameworks. They used the automatic model partitioning idea for the large model in the serverless computing environment.

Serverless computing is suitable for ML inference tasks due to the short-lived execution feature. Batching is a crucial factor in improving the execution performance in ML inference tasks. However, serverless computing is stateless, which may not support batching and its associated setting parameters (batch size and timeout). Ali et al. [70] demonstrated that ML inference tasks could not benefit from serverless computing without batching. Therefore, they presented the BATCH framework to support batching for ML tasks in serverless computing using a dispatching buffer. Moreover, BATCH can provide automated adaptive batching and setting parameters to guarantee performance and cost.

In the serverless computing literature, there are six types of specific scenarios that have been applied to serverless computing. These related studies have presented the corresponding frameworks to address specific limitations, improving feasibility and execution efficiency. IoT-based serverless frameworks are most widely investigated, implying the potential future trend of edge serverless computing.

6.5.2 General Framework. Many general programming frameworks have been designed to make serverless computing popular and less restrictive and facilitate the practice of broader applications. These frameworks have been based on various improvement points, including orchestration model design, function scheduling, runtime mechanism, abstract-based design, and architecture design. Table 15 shows a summary of general framework studies on programming framework.

• **Orchestration model design:** Major cloud providers of serverless computing have presented the corresponding orchestration services of serverless functions, such as AWS Step Functions [19] and Azure Durable Functions [59]. These services allow software developers to construct complex applications with more serverless functions and various structures via a central orchestrator

Table 15. A Summary of General Framework Studies on Programming Framework

Study	Solution	Used strategy	Considered factor
Baldini et al. [79]	Orchestration model design	Express and build the orchestration of existing software function blocks	Robustness of the programming model
Gerasimov et al. [127]	Orchestration model design	Design an orchestration-specific domain language	Data compatibility
Zheng et al. [280]	Orchestration model design	Adopt the copy-based strategy and connector-based strategy	Supportability of cross-regional scenarios
Maslov and Petrashenko [185]	Orchestration model design	Use a control-flow graph and the finite automaton	Developers' demand
Carver et al. [94, 95]	Function scheduling	Design a decentralized scheduler that incorporated static scheduling and dynamic scheduling	Task graph coordination
Zhang et al. [278]	Runtime mechanism	Use checkpoints in the application code	Function timeout issue
Al-Ali et al. [68]	Runtime mechanism	Design the new application primitive	Function alternative
Fouladi et al. [123]	Runtime mechanism	Design a unique intermediation representation layer	Diversity of application types
Jangda et al. [139]	Abstract-based design	Design operational semantics	Low-level behaviors
Meissner et al. [186]	Abstract-based design	Extend the event sources to obtain the application persistence	Retroactive programming capability
Mujezinović and Ljubović [190]	Architecture design	Combine AWS Lambda with AWS Fargate technology	Function execution time and storage space issues
Sánchez-Artigas et al. [220]	Architecture design	Employ object storage for serverless functions	Shuffle operation problem

or workflow engine. This way is similar to the service composition approach [169], integrating multiple services with different types to obtain new benefits.

Except for orchestration services from the industry, some studies in academia have also proposed new orchestration models. Baldini et al. [79] presented a robust programming model to express and build the orchestration of existing software function blocks. Considering that the type of input arguments and intermediate results should be checked on their compatibility during the function orchestration, Gerasimov [127] designed an orchestration-specific domain language called Anzer to check the compatibility between serverless functions. GlobalFlow [280] was a new orchestration model to address the cross-region scenario problem of serverless functions. There were two strategies, including a copy-based strategy and a connector-based strategy. In the copy-based strategy, all serverless functions were copied to execute in one region. This strategy is suitable for tasks without any data usage or data communication. In the connector-based strategy, serverless functions were grouped according to regions, and functions with the same region were integrated into a sub-workflow. Then, a lightweight connector was used to establish the communication of sub-workflows. This strategy can achieve data locality and reduce the data communication overhead. In addition, Maslov and Petrashenko [185] used a control-flow graph and the finite automaton to arrange the function orchestration according to developers' demands.

● **Function scheduling:** Researchers have considered designing efficient schedulers in new programming frameworks. WUKONG [94, 95] was presented as a serverless-oriented and decentralized framework. It adopted a decentralized scheduler that incorporated static scheduling and dynamic scheduling. Specifically, first, it partitioned the global task graph into multiple local

subgraphs before and during executions. Then, each WUKONG's executor scheduled and executed the functions contained in the subgraph, improving the data locality. Moreover, the data reading and writing problem was addressed by task clustering and delayed I/O. WUKONG can greatly improve application performance and resource utilization. However, WUKONG may introduce additional security risks due to its weak isolation.

• **Runtime mechanism:** For some presented general frameworks, researchers have modified their runtime mechanisms to obtain a stronger processing ability. Specifically, to allow software developers to develop general-purpose parallel applications on serverless platforms, functionality partition challenges need to be solved to satisfy the execution constraint of serverless functions. A programming framework called Kappa [278] was designed to address these challenges. Kappa ran the serverless application code on the original serverless platforms like AWS Lambda. Then, it used checkpoints to check the timeout issue of serverless functions and provided concurrency mechanisms for general-purpose parallel applications. Instead of checking the application code, Al-Ali et al. [68] designed the new application primitive that provided the process, not the function, to execute a broad class of applications in the serverless architecture. In addition, gg [123] was presented. In gg, a unique intermediate representation layer was designed to manage various applications by abstracting the computation and storage. Additionally, gg leveraged common services like dependency management, straggler mitigation, and scheduling to support widely used applications, e.g., software compilation, unit tests, and video encoding.

• **Abstract-based design:** Some studies have considered different abstract representations to design general programming frameworks. λ [139] was an operational semantics to keep low-level behaviors of serverless platforms, including concurrency, function retry, failure, and instance usage. In addition, retro-λ [186] was a concept specified for serverless applications. It extended the event sources to obtain the application persistence, allowing the application logic decomposition to support the retroactive programming capability of serverless platforms.

• **Architecture design:** Serverless platforms restrict the function execution time and storage space of function instances, making some applications unable to run on the platform. Therefore, Mujezinović and Ljubović [190] proposed a novel architecture, which combined AWS Lambda with AWS Fargate technology [14] to bypass these limitations. Moreover, the architecture used the famous producer-consumer architectural pattern to handle applications with high-frequency data. In addition, to address the shuffle operation problem that needs to exchange data across all instances, Sánchez-Artigas et al. [220] presented the design of Primula, which had the shuffle operator ability by employing object storage for serverless functions. Moreover, Primula also provided automatic parallelism and data detection to guarantee executions.

The above general serverless programming frameworks are based on different solutions, including orchestration-based model design, function scheduling, runtime mechanism, abstract-based design, and architecture design. These frameworks help software developers to develop serverless applications with fewer usage restrictions. Moreover, they support a wider range of application types. However, presented programming frameworks require application execution to be deterministic. For dynamic applications or workloads, their execution paths may be mutable, causing general frameworks not to work.

6.6 Application Migration

In studies related to application migration, manual and automated conversion approaches have been presented. A summary of studies on application migration is shown in Table 16.

• **Manual conversion:** For manual conversion approaches, researchers of related studies have illustrated step-by-step how to transform applications in a manual way. However, major studies have been specific to a certain class of applications. For example, a partial migration method [78]

Table 16. A Summary of Studies on Application Migration

Study	Solution	Used strategy	Target object
Bajaj et al. [78]	Manual conversion	Implement components with short-lived functionality as serverless functions	Web applications
Christidis et al. [100] and Chahal et al. [97]	Manual conversion	Address some restricted factors	AI-related applications
Elordi et al. [118]	Manual conversion	Provide a specific decomposition methodology	Deep neural network inference applications
Stafford et al. [237]	Manual conversion	Design multiple refactoring iterations	General applications
Yussupov et al. [269]	Automated conversion	Use a canonical serverless application model and assessment metrics	Specific applications
Perera et al. [202]	Automated conversion	Generate high-level serverless architecture design diagrams	Specific applications
Kaplunovich et al. [147]	Automated conversion	Parse and refactor the original code	Java applications
Related studies [108, 213]	Automated conversion	Parse and refactor the original code	Node.js applications

was proposed to handle web applications. In this method, some scalable and resource-intensive components were implemented as microservices, while components with short-lived functionality were transformed into serverless functions. Christidis et al. [100] and Chahal et al. [97] processed AI-related applications by considering some restricted factors, such as the code size, model storage, training and inference difference, and performance tuning. Elordi et al. [118] aimed at deep neural network inference tasks. They provided a specific decomposition methodology, where original applications can be transformed into an application pattern that can execute on AWS Lambda.

Except for specific applications, Stafford et al. [237] designed multiple refactoring iterations for general applications to analyze memory usage and execution time before and after the refactoring. Then, they determined the best practices, such as function type, dependency relationship, and request type.

● **Automated conversion:** For automated conversion approaches, the application conversion process is done automatically. However, in this process, applications need to be assessed for their portability on a specific serverless platform before migration. SEAPORT [269] was an automated assessment method that used a canonical serverless application model and assessment metrics (e.g., service and component similarity). TheArchitect [202] was a rule-based system to automatically generate high-level serverless architecture design diagrams, simplifying the conversion process. However, how to automate conversions is still a critical question. ToLambda [147] targeted Java applications to serverless applications. Node2FAAS [108] and DAF [213] targeted Node.js monolithic applications into serverless applications. These approaches automatically parsed the original code and transferred it as event-driven functions.

Although some studies have presented the corresponding application migration approaches, there are still two problems to be addressed. First, how to automate conversions is a critical question, since a mature approach is lacking. Second, providing a generic application migration approach is challenging for different application types. Researchers may consider a series of factors, such as the vendor lock-in problem, function granularity, and state management.

6.7 Cost

Regarding the cost aspect of serverless computing, related researchers have addressed problems of cost prediction and cost optimization of serverless applications.

Table 17. A Summary of Studies on Cost Prediction

Study	Solution	Used strategy
Cordingly et al. [102]	Regression model prediction	Consider CPUs and memory settings to train the model
Eismann and Grohmann [113]	Regression model prediction	Consider function's input parameters and monitoring data to mixture density networks
Akhtar et al. [65]	Statistical learning prediction	Consider cost and unseen configurations to learn their relationship using Bayesian Optimization
Lin and Khazaei [167]	Statistical learning prediction	Consider the consumed memory and execution time to generate a probability-based cost graph

6.7.1 Cost Prediction. For cost prediction studies, there are mainly two common solutions, including regression model prediction and statistical learning prediction. Table 17 shows a summary of cost prediction studies.

• **Regression model prediction:** Some cloud providers of serverless computing have provided pricing calculators on their websites, but these calculators do not generate the corresponding cost for average runtime and memory size. Therefore, Cordingly et al. [102] referenced the platform's pricing policy to estimate the cost for the predicted runtime performance. The runtime performance was obtained from the regression model that considered different CPUs and memory settings. Differently, Eismann and Grohmann [113] found that the execution latency of the function was related to the function's input parameters. Based on this insight, they presented a prediction approach, which applied the monitoring data from this function to mixture density networks to predict distributions of execution latency and output parameters of the serverless function. Then, runtime performance information was combined into a workflow model to estimate the cost via the Monte Carlo simulation.

• **Statistical learning prediction:** Another kind of approach is to consider statistical learning. Considering that the cost is affected by the allocated memory, Akhtar et al. [65] proposed a framework called COSE. COSE used Bayesian Optimization to learn the relationship between cost and unseen configurations of a serverless function from trace logs. COSE can predict the cost under different configurations. Lin and Khazaei [167] treated the serverless application as the directed acyclic graph. Then, they introduced a probability-based cost graph that can get the average cost of the serverless application. This graph considered the consumed memory and execution time of each serverless function, as well as the transition probability between serverless functions.

6.7.2 Cost Optimization. In studies related to cost optimization, there are three solutions, including function scheduling, underlying combination, and resource adjustment. Table 18 shows a summary of cost optimization studies.

• **Function scheduling:** Generally, the serverless application is composed of one or more serverless functions. The transition from one function to another will increase the number of invocations. However, the price is related to the number of invocations. In this situation, a possible solution is to fuse multiple functions as a function and then schedule this function to an appropriate instance for executions. Based on it, Elgamal [117] discussed the problems of function fusion and function placement and then presented the corresponding cost graph. The cost optimization was concluded as the constrained shortest path problem about this cost graph to find the best structure. In addition, developers may use a hybrid public-private cloud to develop and execute their applications. In such an infrastructure environment, the scheduling strategy of functions is critical in minimizing the cost of public cloud usage while meeting the specified performance deadline. A hybrid cloud scheduling strategy called Skedulix [105] was presented to solve this problem. Moreover, Skedulix used a greedy algorithm to determine the function placement dynamically.

Table 18. A Summary of Studies on Cost Optimization

Study	Solution	Used strategy
Elgamal et al. [117]	Function scheduling	Analyze function fusion and function placement to present the cost graph
Das et al. [105]	Function scheduling	Consider the performance deadline to minimize the cost of public cloud usage
Related studies [131, 135, 176]	Underlying combination	Combine serverless computing and VM rentals to design scalable and hybrid approaches
Related studies [65, 90, 167, 234, 281]	Resource adjustment	Present optimization algorithms to provide an optimal memory configuration

• **Underlying combination:** Specific applications executed on serverless platforms may be costly since serverless computing is currently good at short-lived and stateless tasks. The combination of serverless computing and VM rentals may make applications cost more cost-effective. Some studies [131, 135, 176] have presented scalable and hybrid approaches to analyze and choose the optimal cost in the environment of serverless computing and VM rentals, while ensuring the performance constraint.

• **Resource adjustment:** The cost of the serverless function is affected by the pre-allocated memory size. Therefore, some studies [65, 90, 167, 281] have presented optimization algorithms to provide an optimal configuration setting (e.g., memory) that can achieve the minimum cost under performance constraints. In addition to finding the optimal memory configuration, Spillner [234] measured the memory consumption situation of the serverless function within a particular time and then created trace profiles in advance to automatically update memory.

Cost is closely related to performance. Therefore, the related studies have first leveraged the performance information to further predict and optimize the cost. Moreover, these studies have considered memory allocation size and function duration time according to the billing pattern of serverless platforms. However, this billing pattern is effective for CPU-bound computation workloads. Other workloads (such as I/O-bound workloads) need to pay for extra computation resources that they are underutilizing. Therefore, a new optimization insight may be useful for cost optimization. Moreover, serverless providers may consider a broader range of factors (e.g., memory, networking, and storage) in the billing pattern, in order to provide a fair cost prediction.

6.8 Multi-cloud Development

To support the multi-cloud development of serverless computing, researchers have mainly designed new serverless computing architectures or frameworks. Table 19 shows a summary of studies on multi-cloud development. Soltani [233] was a Peer to Peer architecture that used container cluster manager technology to allow developers to enjoy the respective strengths of multiple clouds. The multi-cloud development architecture presented by Vasconcelos [249] contained three key components, i.e., FaaS Cluster, FaaS Proxy, and Multi-Cloud Resource Allocator. Specifically, FaaS Cluster represented a cluster that contained multiple instances from geographically different cloud infrastructures, while FaaS Proxy provided the request interface like a gateway. Multi-Cloud Resource Allocator can control resources in any of the clouds. In addition, Sampé et al. [218] presented an extensible multi-cloud framework to execute regular Python code on any serverless platform transparently. The architecture design of this framework was motivated by Python's multiprocessing and dynamism features.

The existing studies have designed new architectures to support multi-cloud development. Through these architectures, developers may enjoy benefits from different clouds. For example, AWS is arguably the most innovative cloud, while Google Cloud Platform offers superior services.

Table 19. A Summary of Studies on Multi-cloud Development

Study	Used strategy
Soltani et al. [233]	Use container cluster manager technology
Vasconcelos et al. [249]	Design the multi-cloud development architecture with customize three components
Sampé et al. [218]	Leverage Python's multiprocessing and dynamism features to design the framework

Table 20. A Summary of Studies on Accelerator Support

Study	Solution	Used strategy
Kim et al. [151]	GPU support	Integrate NVIDIA-Docker and support GPU-based containers
Naranjo et al. [193]	GPU support	Use GPU virtualization
Ringlein et al. [212]	FPGA support	Design a platform architecture with disaggregated FPGAs
Bacis et al. [76]	FPGA support	Monitor and time-share FPGAs

However, cloud services from different clouds are still unable to communicate and interact, which may lead to inefficiencies in the collaboration of multiple clouds. Therefore, bridging the interaction between different cloud services may be an important topic in the research of multi-cloud development.

6.9 Accelerator Support

Generally, the runtime environment of serverless platforms is mainly based on CPU resources. However, more and more tasks like video processing and deep learning require leveraging other hardware resources to accelerate. In the related studies, researchers have tried to support GPU and FPGA accelerators in serverless platforms. A summary is shown in Table 20.

6.9.1 GPU Support. To address the GPU supportability, Kim et al. [151] presented a new serverless framework, which integrated NVIDIA-Docker into the open-source serverless framework. Moreover, this new framework can support the deployment of GPU-supported containers. However, the biggest disadvantage of this approach is that each GPU cannot serve multiple function invocations simultaneously.

Different from using GPU-support containers, Naranjo et al. [193] used GPU virtualization. They tried several virtualized access methods to GPUs, including remote access to GPU devices via the rCUDA framework [51], as well as direct access to GPU devices via PCI passthrough [49]. The results showed that using GPU virtualization is an efficient and accelerated way of serverless computing, achieving the sharing of GPUs among functions.

6.9.2 FPGA Support. To make the serverless platform support FPGAs, Ringlein et al. [212] designed a platform architecture with disaggregated FPGAs for serverless computing. In addition, BlastFunction [76] was a distributed FPGA sharing platform, which contained multiple device managers to monitor and time-share FPGAs. Moreover, BlastFunction can achieve multi-tenancy for FPGAs.

In serverless computing, the most widely used function instances are based on the CPU. In this research direction, the existing studies have aimed at GPU support and FPGA support. Although more accelerators supported in serverless platforms may not be economically viable for most application types, the mix of multiple heterogeneous accelerators may create a new research dimension.

6.10 Security

In the security aspect of serverless computing, there are mainly two kinds of solutions, including SGX-based design and information flow tracking. Table 21 shows a summary.

Table 21. A Summary of Studies on Security

Study	Solution	Used strategy
Qiang et al. [210]	SGX-based design	Leverage Intel Software Guard Extensions and WebAssembly sandboxed environment to protect the API gateway
Alder et al. [69]	SGX-based design	Use SGX with the trustworthy resource measurement mechanism
Alpernas et al. [71]	Information flow tracking	Achieve the security guarantee through static program labeling with dynamic data labeling
Datta et al. [106]	Information flow tracking	Design transparent function-level information flow model
Sankaran et al. [221]	Information flow tracking	Check access control policies and make a decision for requests in advance

● **SGX-based design:** Serverless applications follow the event-driven paradigm, and they generally are triggered by HTTP requests; thus, the API gateway is a crucial component. However, this component often suffers from malicious attacks. To alleviate this situation, Qiang et al. [210] leveraged Intel **Software Guard Extensions** (**SGX**) and WebAssembly sandboxed environment to design Se-Lambda to protect the API gateway. Particularly, SGX can create a trusted execution environment where sensitive data can be stored and used normally, preventing malicious attackers from stealing sensitive data from the serverless application. Moreover, Alder et al. [69] used SGX with the trustworthy resource measurement mechanism in the serverless platform to guarantee security and accountability.

● **Information flow tracking:** Since serverless applications are composed of multiple serverless functions, using **information flow control** (**IFC**) methods may be suitable and beneficial for information security in serverless computing. Trapeze [71] was implemented as a dynamic IFC model to track and monitor the global information flow. Trapeze achieved the security guarantee through static program labeling with dynamic data labeling. However, on one hand, Trapeze left the burden for developers on the definition of information flow policies and the implementation of declassification functions. In contrast, Valve [106] did not require declassification. Valve was a transparent function-level information flow model, allowing developers to use fine-grained controls in their serverless applications and write proper policy configurations. On the other hand, Trapeze's implementation also relied on the programming language of serverless functions and predefined key-value store functions. Based on these limitations of Trapeze, a transparent approach named will.iam [221] was presented, which was agnostic to serverless functions and underlying platforms. will.iam can automatically check access control policies and make a decision for requests in advance. Valve and will.iam were complementary, where Valve was to understand the data flow information of their serverless applications and write reasonable policies for will.iam.

The programming paradigm of serverless computing can reduce the attack surface, since maintaining and patching up the security loopholes is assigned to the cloud providers. The existing studies on security have mainly presented solutions from the serverless platform. According to the development pattern of serverless applications, the serverless platform offers multiple settings and features and features. Developers may use incorrect settings or configurations, resulting in a security thread. Moreover, these settings or configurations can act as an entry point for attacks against serverless architectures. On the other hand, the permissions or rights to the functions should be properly configured. However, in practice, it can create a situation where serverless functions become overprivileged, thus causing a potential security threat. Therefore, studies related to security issues can also be added to the configuration analysis.

Table 22. A Summary of Studies on Testing and Debugging

Study	Used strategy
Winzinger and Wirtz [259]	Use additional instrumentation to obtain the function coverage
Alpernas et al. [72]	Design a record-and-replay debugger to recreate and re-execute the application

6.11 Testing and Debugging

Table 22 shows a summary of studies on testing and debugging. In the distribution environment of serverless computing, serverless functions interact with other functions or cloud services. It makes serverless application testing challenging [163]. Winzinger and Wirtz [259] implemented a data flow testing framework to obtain the function coverage. This framework used additional instrumentations in the source code to detect data flows between functions and services, between return values, and between functions and functions.

Debugging the serverless application is an essential step in facilitating serverless computing [161, 242, 256]. The current solution is to use the record and replay approach. Watchtower [72] was presented to observe the application changes through instrumenting libraries. Watchtower can be structured as a serverless application to scale at the same rate. Then, developers can analyze output logs to detect the violations affecting the correctness of serverless applications. If a violation was found, a record-and-replay debugger contained in Watchtower could be used to recreate and re-execute the application from a specific state on the developer's local machine.

Software testing and debugging are critical to assess whether a software application successfully meets requirements and specifications. That is also applied to serverless applications. However, the related studies are relatively few. In serverless applications, several serverless functions or external services are integrated. Therefore, a sufficient testing tool should focus more on the interactions between serverless functions and external services. In addition, serverless functions are event-driven. The actual root cause of the error may be challenging to locate the specific execution flows. Therefore, the distributed tracing approach with serverless features may be useful.

7 RQ3 (EXPERIMENTAL SETTING AND EVALUATION)

To answer the research question of how the existing solutions perform experimental setting and evaluation, we aim at exploring three sub-research questions, i.e., RQ3.1, RQ3.2, and RQ3.3. For RQ3.1 and RQ3.2, the first two authors independently view the *Implementation* and *Evaluation* parts in each research article, in order to determine the experimental serverless platforms and the availability of experimental validation for the presented solution. For RQ3.3, the first two authors independently view the full text in each research article to check whether the experimental datasets or code is shared and whether it is still accessible via website search. These answers about RQ3.1, RQ3.2, and RQ3.3 are deterministic. The results given by the first authors are aggregated together to find the conflicts. These conflicts are resolved by the third arbitrator, and the final answer agrees with the first two authors. Figures 6, 7, and 8 show the final results.

We use Figure 6 to answer RQ3.1. Figure 6 shows the distribution of experimental or evaluated serverless platforms for existing solutions. The distribution shows that a wide variety of serverless platforms are available for implementing or evaluating proposed solutions. These serverless platforms include custom systems related to serverless computing, AWS Lambda, OpenWhisk, OpenFaaS, OpenLambda, Google Cloud Functions, IBM Cloud Functions, Azure Functions, and other serverless platforms (such as Alibaba Cloud Function Compute and Knative). From the percentage of serverless platforms in Figure 6, we can summarize the following four points. First, a custom system related to serverless computing is the widest implementation way, accounting for 31.49% of all serverless platforms. To address problems of a certain serverless aspect, some researchers [67, 92, 109, 146, 153, 245] have designed new serverless platforms to implement or
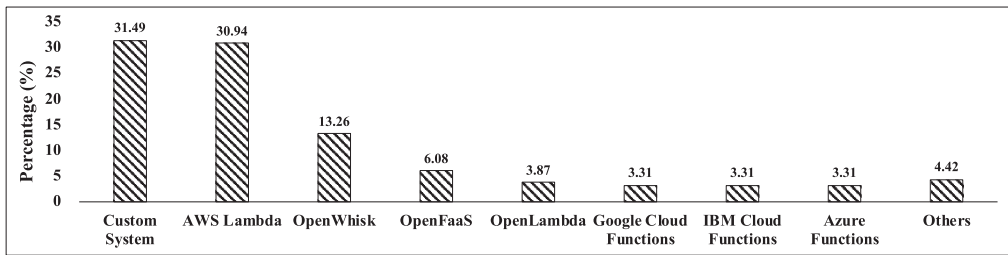
Fig. 6. The distribution of experimental or evaluated serverless platforms for existing solutions.

host their solutions. For example, SAND [67] was a novel, high-performance serverless platform, which designed some custom components to improve the cold start performance. Kaffes et al. [146] designed a new serverless system to execute highly bursty, stateless, and short-lived applications. This platform highlighted a centralized core-granular scheduler, which is more fine-grained than traditional serverless schedulers. The new requirement of the scheduler forced researchers to present the new underlying design of serverless platforms; thus, their solution was implemented in a custom system. Second, AWS Lambda is the second most widely used serverless platform in the serverless computing literature, accounting for 30.94% of all serverless platforms. Moreover, AWS Lambda is used far more than other commercial serverless platforms like Google Cloud Functions and Azure Functions. It illustrates that AWS Lambda is more mature regarding serverless features and infrastructure to help researchers design or present their solutions. For instance, Pocket [155] was a novel storage system for optimizing the communication problem between serverless functions. Pocket placed the original storage and collaborated with AWS Lambda to scale with the serverless functions automatically. Third, the most commonly used open-source serverless platform is OpenWhisk, which accounts for 13.26% of all serverless platforms. The specific detail of OpenWhisk is explained in Section 3. Researchers can modify the underlying architecture of OpenWhisk to add or redesign components to achieve their goals. For instance, Zhang et al. [279] presented a new insight into using Harvest VMs, whose resource management differed from traditional underlying resource management for VMs and containers. Considering the characteristics of Harvest VMs, they redesigned a load balancer in OpenWhisk. Finally, the "Others" category in Figure 6 refers to serverless platforms with low usage, containing Alibaba Cloud Function Compute, Knative, Fission, Kubeless, and Huawei's FaaS Framework. These serverless platforms account for only 4.42% of the total platforms, illustrating that these platforms have not been widely used in the serverless computing literature. For example, the Alibaba Cloud Function Compute team [8] presented a rapid container provisioning approach, FaaSNet [250], to improve its own platform's container provisioning speed to serve requests.

We use Figure 7 to answer RQ3.2. Figure 7 represents the distribution of the availability of experimental validation of the existing solutions. The result shows that 93.90% of the existing solutions are evaluated by using experimental validation. For instance, Barcelona-Pons et al. [82] leveraged different application types and presented four evaluation questions to validate the benefits of their solution. The distribution result is related to our inclusion criteria for selecting research articles. In our article, we select research articles addressing specific problems related to serverless computing and presenting the corresponding solutions. Generally, presented solutions need to be evaluated to demonstrate their benefits or efficiency. Therefore, a majority of existing solutions have experimental validation, while only 6.10% of all solutions are not validated experimentally. For example, Boucher et al. [89] presented a novel proposal to describe a FaaS design that attempts to be truly micro, but this design lacked the related experimental evaluation.
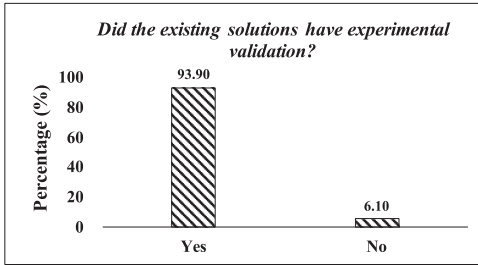
Fig. 7. The distribution of the availability of experimental validation of the existing solutions.
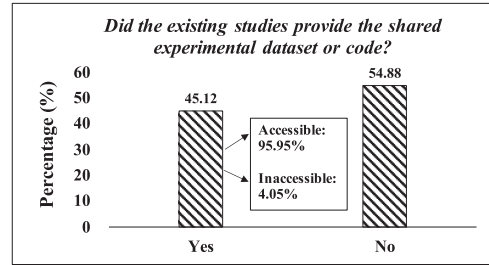

Fig. 8. The distribution of the availability of experimental datatsets or code of the existing studies.

We use Figure 8 to answer RQ3.3. Figure 8 represents the distribution of the availability of experimental datasets or codes of the existing studies. The results show the following two points. First, 45.12% of the existing studies provide the shared datasets or code, while 54.88% of studies do not release the used datasets or code. It shows that more studies have focused on describing the design and implementation of their solutions rather than further providing reproducibility possibilities. Second, for the shared datasets or code, 95.95% is accessible, indicating high reproducibility. Only 4.05% of the shared links are inaccessible. For example, the data links provided by Chadha et al. [96] and Jindal et al. [143] cannot point to valid pages on the Internet. It suggests the researchers pay attention to the correctness of publicly available data links.

8   RQ4 (PUBLICATION VENUES)

To answer the research question of where research articles are published, the first two authors independently search Google Scholar to determine each research article's publication venue, e.g., a specific publication conference or journal. The results given by the first two authors are compared to find the conflicts, and these conflicts are resolved by the third arbitrator. Moreover, all authors agree on the final results.

We analyze publication venues for research articles according to the well-adopted metrics of Computer Science Rankings (CSRankings[2]). Figure 9 shows the main distribution of publication venues for research articles addressing specific problems. Meanwhile, we also investigate other important publication venues related to software engineering, cloud computing, and so on, in Figure 9. Table 23 shows a summary of main publication venues and their covered research directions. Due to the long name of the publication venue, we use abbreviations to represent them, and the corresponding full names are shown in Figure 9. For example, "OSDI" represents *USENIX Symposium on Operating Systems Design and Implementations*, while "TOSEM" represents *ACM Transactions on Software Engineering and Methodology*.

From Figure 9 and Table 23, we can summarize the following points. First, the top publication venues are "SoCC", "TPDS", and "CLOUD", accounting for 28 of the 164 articles, respectively. Serverless computing is the next generation of a promising cloud computing paradigm. Therefore, it is reasonable that more research articles are published in cloud computing-related conferences like "SoCC" and "CLOUD". Second, the publication venue covers multiple research directions related to serverless computing. For example, Table 23 shows that conferences for "OSDI", "SOSP", "EuroSys", and "USENIX ATC" have six research directions of serverless computing, including resource management, general framework, cold start performance, function communication, function execution, stateful FaaS. In addition, publication venues about "SoCC", "TPDS", and
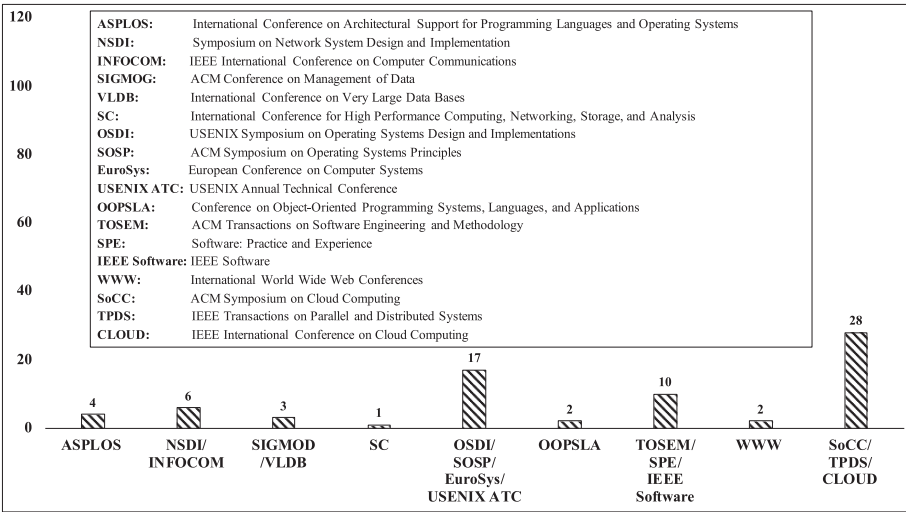
---

[2]https://csrankings.org.

Fig. 9. The main distribution of publication venues for research articles addressing specific problems.

Table 23. A Summary of Main Publication Venues and their Covered Research Directions

Publication venues	Covered research directions (leaf node representation)
ASPLOS	Cold Start Performance; Function Communication
NSDI/INFOCOM	Specific Framework; Cost Prediction; Cost Optimization; Performance Prediction; Resource Management; Cold Start Performance; Function Communication; Stateful FaaS
SIGMOD/VLDB	Function Communication; Stateful FaaS
SC	Specific Framework
OSDI/SOSP/EuroSys/USENIX ATC	Resource Management; General Framework; Cold Start Performance; Function Communication; Function Execution; Stateful FaaS
OOPSLA	Security; General Framework
TOSEM/SPE/IEEE Software	Application Modeling; Specific Framework; Cold Start Performance; Multi-Cloud Development; Application Migration; Stateful FaaS
WWW	Security; Resource Management
SoCC/TPDS/CLOUD	Testing and Debugging; Cost Prediction; Cost Optimization; Performance Prediction; Application Modeling; General Framework; Specific Framework; Resource Management; Cold Start Performance; Function Communication; Function Execution; FPGA Support

"CLOUD" include twelve research directions, ranging from testing and debugging [72] to resource management [87]. Third, current publication venues mainly contain two kinds of communities, i.e., the Software Engineering community and the Systems community. Specifically, some studies have been published in software engineering-related journals, such as "SPE", "IEEE Software" and "TOSEM". These studies have addressed problems with the application modeling [271], programming framework of specific framework [84, 276], multi-cloud development [218], application migration [213], and so on. Other researchers have presented their approaches in system-related venues, such as "OSDI", "SOSP", "EuroSys", and "USENIX ATC". These approaches mainly resolved problems with resource management [274, 279], cold start issues [109, 125, 246], function communication [155, 208], and so on. Overall, this key point indicates that current serverless computing has difficult unresolved issues on the software application side and serverless platform side. In practice, the biggest beneficiaries of serverless computing are software developers. Existing difficult issues will prevent software developers from enjoying the advantages of easy development and fast application execution. Therefore, addressing issues on the software application side and serverless platform side is to better facilitate software developers' application development practices on serverless platforms.

Note that we do not use any analysis framework. In our work, we use an Excel spreadsheet to provide all of the extracted articles and the analyses, following a similar strategy from other studies [73, 80, 98, 256].

9 OPPORTUNITIES FOR RESEARCHERS

In this section, we aim at discussing open challenges compared to the efforts already made and envision promising research opportunities for future research on serverless computing.

Generalizability of application conversion approaches. Based on the benign characteristics of serverless computing, various applications are migrated to the serverless platform for executions. As reported in the characterization study about serverless applications [114], applications are diverse and not limited to any specific types. However, existing application conversion approaches have targeted only a few specific applications, such as AI applications [97, 100, 118], Web applications [78], and Java applications [108, 147]. There are not yet generic conversion tools for any application. Though Stafford et al. [237] presented a multiple-refactoring iteration approach, this approach just changed the runtime results of different states. Then, it determined which change was beneficial to the application. Moreover, this approach did not provide specific conversion steps. Providing a generic application conversion approach is challenging, which requires addressing a series of problems, e.g., application type, vendor lock-in, function granularity identification, event-driven code transformation, cloud service selection, and state communication.

Cold start performance problem of serverless applications. Although the state-of-the-art solutions for cold start performance optimization (e.g., data cache-based optimization [67, 197] and snapshot-based optimization [92, 252]) can effectively improve the cold start performance, they mainly target the acceleration of container creation or runtime initiation. However, containers generally lack isolation or flexibility due to their inherent nature. Therefore, providing isolation or flexibility for current common sandboxes like containers is a tricky problem. On the other hand, even if there is a short time for container acceleration, cold start performance still faces another overhead, i.e., application initialization overhead, which may take up a large portion of the cold start time in the future. In this situation, it also reveals a new opportunity, i.e., how to optimize the overhead of application initialization in the case of faster runtime initialization already available.

Performance variability of serverless functions. In related studies of performance prediction for serverless functions, most solutions have been based on the collected runtime information of a serverless function in a period to predict the performance value of a constant [65, 102, 111]. However, the auto-scaling feature of serverless computing makes function performance variable and resource allocation dynamic. It is not enough to rely on factors considered constant value to predict dynamic performance. Eismann et al. [112] discussed the performance variability of serverless computing and determined serverless-specific changes, such as uncertainty in cold and warm starts, load intensity, and short-term and long-term performance fluctuations. Therefore, it is essential that the performance problem of serverless computing take into account the dynamic feature or distributions. Unfortunately, most serverless platforms are untouchable to software developers and expose little to no information about the underlying environment. Therefore, considering performance variability in serverless performance is difficult, but it also hints at a future research opportunity.

Data privacy of IoT applications. In the related studies of the programming framework, IoT-specified serverless frameworks are the most widely studied in specific application scenarios, accounting for 36.84% (7/19) of all specific frameworks. Presented frameworks have addressed the movement problem of data and code [99], resource constraint problem of edge devices [205, 276], deployment operation problem [260], heterogeneous cluster access problem [143], and so on. However, these studies have not focused on the data privacy challenge for IoT applications. Edge devices

are placed in different geographical locations and may collect data not wanted to be made public. Moreover, they communicate with each other. In this situation, edge devices have no unified cloud management like serverless functions; thus, they are vulnerable to attacks. Data privacy preservation is critical for IoT applications. With respect to data confidentiality, developers can encrypt the data before storing it in external cloud storage when writing IoT functions. However, when another IoT function queries the required data from the storage, query operation may encounter obstacles in the absence of decryption. In future research, a possible solution is to design a novel "index", which uses encryption techniques that can support the execution of operations or query evaluation.

Testing tools. AWS Lambda can use GUI to invoke serverless functions, indirectly implementing the intent of the function testing. In practice, there is still a lack of mature testing tools for serverless applications. Although Winzinger and Wirtz [259] implemented a data flow testing framework, tested applications still need to be deployed to the serverless platform after modifying the corresponding source code. The most challenging part of implementing testing tools may be to mock the real serverless environment in the local environment [163]. This is unlike monolithic applications and microservice applications, where the environment can be tested directly locally. In serverless computing, software developers do not know which containers will be used in underlying platforms during deployment; thus, it shows an open challenge for serverless application testing. Moreover, the serverless application is composed of multiple dependent serverless functions. Integration testing may be necessary to ensure the correctness of the serverless application. Therefore, providing a serverless-specific testing tool will be a promising opportunity for serverless applications.

Security and efficient direct communication between serverless functions. To alleviate the function communication problem, researchers have adopted various optimization solutions, such as memory sharing [142, 156], cache-based design [244, 261], and storage optimization [155, 208]. However, these strategies have still been built on the idea of temporary data stored in a specific place like memory and external storage for function communication. On the one hand, this idea may expose security issues, especially for sensitive data. Even if data can be encrypted during function communication, data shared in memory or cached in containers may still be maliciously attacked or stolen by other tenants. Therefore, providing a secure serverless platform remains challenging but also a research opportunity. On the other hand, this idea of temporary data stored in a specific place cannot avoid the additional overhead of saving and fetching or managing data. Boxer [255] supported the direct communication between serverless functions. It leveraged a modified network, which used TCP hole-punching techniques of P2P to solve the limitation of the conventional network. However, for large-scale communication-intensive applications, Boxer may not provide high-throughput and low-latency networks for functions. Therefore, it will elicit a promising research opportunity, i.e., how to design efficient direct network communication for serverless functions.

Effectiveness of the pricing model. Existing studies about cost prediction and optimization [117, 176, 234, 281] have been based on the billing model that pays for actually consumed computation resources. Such a billing model is effective and economical for computation-intensive workloads. However, when developers execute I/O-intensive or disk-intensive workloads, the current billing model may be expensive, and allocated computation resources are also not efficiently utilized. In this situation, the challenges that cloud providers of serverless computing face are to further consider a more suitable billing pattern for various resource types, e.g., CPU, memory, networking, and storage. Moreover, existing solutions of cost prediction and optimization studies relied solely on the consumption of computation resources. A dynamic pricing prediction and optimization scheme may be a research opportunity in the future.

Fine-grained resource configuration on application development. Generally, software developers configure a small set of parameters (e.g., function timeout, memory size, and cloud providers) for their applications, and these parameters are simple. This development way frees developers from underlying resource management. However, some experienced developers may be willing to configure the fine-grained resource policies to effectively improve the overall quality of service of applications [65]. Therefore, this situation also motivates a research opportunity for performance improvement. The serverless platform can optionally support some fine-grained configuration options about the underlying runtime information. For example, developers can configure network conditions, function placement, I/O bandwidth, and so on.

Heterogeneous accelerator support. Besides existing studies about GPU support [151, 193] and FPGA support [76, 212], other accelerators like **Tensor Processing Unit (TPU)** also should be noticed for cloud providers of serverless computing. However, supporting new accelerators may be challenging in serverless platforms because it may require designing a new scheduler, resource allocation pattern, or billing model. Moreover, accelerators generally have some internal restrictions, e.g., I/O bandwidth and energy efficiency, and thus they are difficult to implement in the serverless platform. On the other hand, supporting more accelerators may not be economically viable for certain scenarios. However, providing heterogeneous accelerator support creates a new research dimension of significant importance for scenarios that use function instances with a mix of CPU, GPU, FPGA, and TPU.

Monitoring tools. The monitoring capability may not be enough for current serverless platforms. Some third-party monitoring tools, such as Epsagon [33] and Datadog [29], may also be applied to trace the serverless application. However, they still do not contain resource consumption and infrastructure-related metrics, e.g., CPU utilization, network condition, and infrastructure performance, to further understand the serverless application and platform. Therefore, in the serverless platform, providing comprehensive observability of both serverless applications and platforms is a complex undertaking, but it is also an opportunity for future research on serverless computing.

Representativeness and completeness of benchmark dataset. Studies related to serverless computing have used some benchmark datasets to verify the efficiency of their solutions [67, 113, 141, 279] or obtain characterization results [114, 171, 257, 258]. However, we find that these studies have not used a standard benchmark dataset. For example, SAND [67] used image processing applications, while Boki used the real applications from DeathStarBench microservices [30]. Moreover, the measurement work [279] used multiple Python serverless functions from Function-Bench [152]. To characterize serverless applications, the authors [114] collected different applications from open-source projects, academic literature, industrial literature, and domain-specific feedback. This situation of the used benchmark dataset indicates that the serverless computing field has not a uniform and representative dataset. On the other hand, the completeness of the benchmark dataset (i.e., containing diverse application types) is also critical for evaluating generic techniques or frameworks. A complete dataset can validate key insights and find potential weaknesses. Therefore, constructing a representative and complete benchmark dataset will be a promising opportunity for future serverless research.

Computing continuum. One potential research direction is the computing continuum. Serverless computing hides the differences away and allows developers to deploy and execute stateless and lightweight functions. This paradigm can collaborate with technologies related to the mobile, edge, and cloud computing to form the computing continuum [83]. This continuum enables the convergence of heterogeneous infrastructures and creates disruptive applications in the future. For example, Baresi et al. [83] provided an implementation possibility in the computing continuum scenario, which combined serverless computing with technologies related to mobile and edge computing.

Configuration management. In serverless computing, the concept of infrastructure as code is used to simplify the custom usage of resources for developers [40]. Infrastructure as code means that developers configure the required resources through the specific resource configuration format without hand-coded programming. However, some configuration-based questions [2–7] are frequently asked by developers in Stack Overflow. These questions account for the second-largest percentage of the total challenges that developers develop their serverless applications [256]. Thus, fixing configuration-related errors can significantly reduce the deployment time of serverless applications. An automated detection may be useful to reduce the risk of misconfiguration and ease the burden of configuration management on serverless-based developers. Perhaps, researchers will design a new resource configuration pattern or configuration management mechanism specific to serverless computing.

10 OPPORTUNITIES FOR PRACTITIONERS

Serverless computing is an emerging concept, and its related techniques will continue to be updated and adapted in the future. In our study, we aim at providing a snapshot of the current research state of the art of serverless computing at the time of writing. Meanwhile, we also aim at providing some software practices of serverless application engineering for future practitioners.

We present and answer RQ3 about experimental setting and evaluation. This part shows the distribution of experimental serverless platforms and the availability of experimental datasets/code. The results in Figure 6 hint at the popularity of serverless platforms. Therefore, when developing serverless applications, practitioners can implement the serverless-related custom system themselves, or they can use AWS Lambda (a mainstream commercial serverless platform) or OpenWhisk (an open-source serverless platform). Moreover, Figure 8 shows that nearly half of the studies have open-source datasets or codes. Practitioners can directly access such information to facilitate their development process of serverless applications.

From Figure 5, we can observe the type distribution of specific frameworks, where IoT applications are widely studied, accounting for 36.84% (7/19) of all specific frameworks. In practice, practitioners also focus on and develop a large number of IoT applications, because these applications are closely related to daily life. For example, IoT applications can be useful in smart homes. A home is where people always need safety and security, and locks are the foundation of home security. Traditional locks have keys, but keys get lost easily. In this situation, a digital smart door lock system can be developed by practitioners to make homes more secure. Although practitioners may encounter some problems (e.g., data access, data moving, and resource constraint of edge devices) in designing such a system, they can all benefit from the existing research efforts [99, 143, 205, 276].

Furthermore, we observe that the performance problem is a main focus in the serverless computing literature. In our study, we summarize solutions of performance optimization, containing cold start performance, function execution performance, and function communication performance. In designing latency-sensitive applications, practitioners can check which part of the application is the performance bottleneck and then refer to the corresponding solution. For example, for the cold start performance of applications, practitioners can try to reduce the number of cold starts by fusing multiple serverless functions into a serverless function [85, 159, 227]. For the function execution performance, practitioners can consider the effect of memory allocation size. Based on the historical execution information, the memory size that makes the performance optimal can be inferred [167].

In the future, some potential trends in serverless applications will appear. For example, (1) containers are considered more coarse-grained in comparison to serverless functions, and they are treated as an alternative choice. A potential trend is that containers and serverless become an infrastructure foundation for application platforms. Such a way can give full play to respective

strengths, since serverless platforms have started supporting containers to package and deploy the application code. (2) For serverless computing, there is a lack of standardization and interoperability between serverless providers. Some open-source serverless platforms like Knative may speed up the standardization process by leveraging Kubernetes techniques. Moreover, through standards such as cloud events, developers do not have to feel locked into a single cloud service provider for services anymore. However, there is a question to think about: in practical use, should developers or users expect that serverless application code becomes easily portable across various serverless platforms? Perhaps, an efficient serverless code needs to depend on cloud-specific services like databases. (3) The low-code development way can reduce the burden of development projects and minimize the need for specialized technical skills. Moreover, it accelerates the application transformation. Overall, this way has allowed solution-oriented developers without having any engineering background to create valuable applications. Thus, in serverless computing, combining low-code development can further improve productivity and solve business problems quickly.

11 CONCLUSION

In this article, we presented a comprehensive literature review to summarize the current research state of the arts of serverless computing. Specifically, first, we collected and analyzed 164 research articles to construct a taxonomy containing 17 research directions of the serverless computing literature. Second, we classified the related studies of each research direction and elaborated on existing solutions. Third, we explored the experimental setting and evaluation of solutions. Fourth, we showed the distribution of publication venues for selected research articles. Finally, we discussed open challenges and envisioned promising research opportunities for future research on serverless computing. Our analysis of available research work on serverless computing can significantly decrease ambiguity and the entry barrier for novice researchers and practitioners. Moreover, summarized taxonomy, solutions, distributions, and analyses will be of great value for (1) future researchers to pursue promising research topics and insightful ideas for solutions and (2) future practitioners to conduct best software practices of serverless application engineering.

REFERENCES

[1] 2019. A Research and Markets Report. Retrieved May 01, 2022 from https://www.researchandmarkets.com/reports/4828585/serverless-architecture-market-by-deployment.

[2] 2022. Retrieved May 01, 2022 from https://stackoverflow.com/questions/47327765/creating-two-dynamodb-tables-in-serverless-yml.

[3] 2022. Retrieved May 01, 2022 from https://stackoverflow.com/questions/55123962/cannot-create-sqs-subscription-to-an-sns-topic-through-cloudformation-in-localst.

[4] 2022. Retrieved May 01, 2022 from https://stackoverflow.com/questions/44032664/reference-function-from-within-serverless-yml.

[5] 2022. Retrieved May 01, 2022 from https://stackoverflow.com/questions/39793242/serverless-response-template-with-status-code.

[6] 2022. Retrieved May 01, 2022 from https://stackoverflow.com/questions/58224566/serverless-framework-ignoring-authorizer-block-in-lambda-proxy-setup.

[7] 2022. Retrieved May 01, 2022 from https://stackoverflow.com/questions/54581575/conditional-resource-in-serverless.

[8] 2022. Alibaba Cloud Function Compute. Retrieved May 01, 2022 from https://www.alibabacloud.com/products/function-compute.

[9] 2022. Amazon CloudWatch. Retrieved May 01, 2022 from https://aws.amazon.com/cloudwatch/.

[10] 2022. Amazon Elastic Compute Cloud. Retrieved May 01, 2022 from https://aws.amazon.com/ec2/?nc1=h_ls.

[11] 2022. Amazon S3. Retrieved May 01, 2022 from https://aws.amazon.com/s3/.

[12] 2022. Anna. Retrieved May 01, 2022 from http://highscalability.com/blog/2018/9/8/the-anna-key-value-store-now-has-355x-the-performance-of-dyn.html.

[13] 2022. AWS CloudTrail. Retrieved May 01, 2022 from https://aws.amazon.com/cloudtrail/.
[14] 2022. AWS Fargate. Retrieved May 01, 2022 from https://aws.amazon.com/fargate/.
[15] 2022. AWS IoT Greengrass. Retrieved May 01, 2022 from https://aws.amazon.com/greengrass/.
[16] 2022. AWS Lambda. Retrieved May 01, 2022 from https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.
[17] 2022. AWS Lambda Pricing. Retrieved May 01, 2022 from https://aws.amazon.com/lambda/pricing/.
[18] 2022. AWS Serverless Application Repository. Retrieved May 01, 2022 from https://aws.amazon.com/serverless/serverlessrepo/.
[19] 2022. AWS Step Functions Documentation. Retrieved May 10, 2022 from https://docs.aws.amazon.com/step-functions/index.html.
[20] 2022. Azure App Service. Retrieved May 01, 2022 from https://azure.microsoft.com/en-us/services/app-service/.
[21] 2022. Azure Application Insights. Retrieved May 01, 2022 from https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview.
[22] 2022. Azure Functions. Retrieved May 01, 2022 from https://docs.microsoft.com/en-us/azure/azure-functions/.
[23] 2022. Azure Functions Pricing. Retrieved May 01, 2022 from https://azure.microsoft.com/en-us/pricing/details/functions/.
[24] 2022. Azure IoT Edge Documentation. Retrieved May 01, 2022 from https://docs.microsoft.com/en-us/azure/iot-edge/?view=iotedge-2018-06.
[25] 2022. Azure Marketplace. Retrieved May 01, 2022 from https://azuremarketplace.microsoft.com/en-us/.
[26] 2022. The CIO's Guide to Serverless Computing. Retrieved May 01, 2022 from https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing.
[27] 2022. Cloud Development vs Traditional Software Development. Retrieved May 01, 2022 from https://intersog.com/blog/cloud-development-vs-traditional-development/.
[28] 2022. Retrieved May 01, 2022 from CouchDB. https://couchdb.apache.org/.
[29] 2022. DataDog: Modern Monitoring and Security. Retrieved May 10, 2022 from https://www.datadoghq.com/.
[30] 2022. DeathStarBench: Open-source Benchmark Suite for Cloud Microservices. Retrieved May 01, 2022 from https://github.com/delimitrou/DeathStarBench.
[31] 2022. Retrieved May 01, 2022 from Debugging and Testing. https://www.d.umn.edu/~gshute/softeng/testing.html.
[32] 2022. Docker. Retrieved May 01, 2022 from https://www.docker.com/.
[33] 2022. Epsagon: Application Monitoring Built for Containers and Serverless. Retrieved May 10, 2022 from https://epsagon.com/.
[34] 2022. Google App Engine. Retrieved May 01, 2022 from https://cloud.google.com/appengine.
[35] 2022. Google Cloud Functions. Retrieved May 01, 2022 from https://cloud.google.com/functions.
[36] 2022. Google Cloud Operations. Retrieved October 01, 2022 from https://cloud.google.com/products/operations.
[37] 2022. Google Docs. Retrieved May 01, 2022 from https://google-docs.en.softonic.com/.
[38] 2022. Google Gmail. Retrieved May 01, 2022 from https://play.google.com/store/apps/details?id=com.google.android.gm&hl=en_US&gl=US.
[39] 2022. Google gVisor. Retrieved May 01, 2022 from https://github.com/google/gvisor.
[40] 2022. Infrastructure as Code. Retrieved May 01, 2022 from https://en.wikipedia.org/wiki/Infrastructure_as_code.
[41] 2022. Kafka. Retrieved May 01, 2022 from https://kafka.apache.org/.
[42] 2022. Kubernetes. Retrieved May 01, 2022 from https://kubernetes.io.
[43] 2022. LLVM. Retrieved May 01, 2022 from https://llvm.org/.
[44] 2022. Nginx. Retrieved May 01, 2022 from https://www.nginx.com/.
[45] 2022. OpenFaaS. Retrieved May 01, 2022 from https://www.openfaas.com/.
[46] 2022. OpenFaaS Function Store. Retrieved May 01, 2022 from https://github.com/openfaas/store.
[47] 2022. OpenLambda. Retrieved October 01, 2022 from https://github.com/open-lambda/open-lambda.
[48] 2022. Openwhisk. Retrieved May 01, 2022 from https://openwhisk.apache.org/.
[49] 2022. PCI Passthrough. Retrieved May 01, 2022 from https://pve.proxmox.com/wiki/Pci_passthrough.
[50] 2022. Prometheus. Retrieved May 01, 2022 from https://github.com/prometheus/prometheus.
[51] 2022. rCUDA. Retrieved May 01, 2022 from http://www.rcuda.net/.
[52] 2022. Serverless Architecture vs Traditional Architecture. Retrieved May 01, 2022 from https://networkinterview.com/serverless-architecture-vs-traditional-architecture/.
[53] 2022. Serverless Framework. Retrieved May 01, 2022 from https://www.serverless.com/.
[54] 2022. Top 10 Tools for IaaS Cloud Computing. Retrieved September 01, 2022 from https://a3logics.com/blog/tools-for-iaas-cloud-computing/.
[55] 2022. Unikernel. Retrieved May 01, 2022 from http://unikernel.org/.
[56] 2022. Visual Studio Code. Retrieved May 01, 2022 from https://code.visualstudio.com/.
[57] 2022. VMFUNC. Retrieved May 01, 2022 from felixcloutier.com/x86/vmfunc.

[58] 2022. WebAssembly. Retrieved May 01, 2022 from https://webassembly.org/.

[59] 2022. What are Durable Functions? Retrieved May 10, 2022 from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[60] 2022. What you Need to know Before Implementing Infrastructure as a Service (IaaS). Retrieved September 01, 2022 from http://novacontext.com/what-you-need-to-know-before-implementing-iaas/.

[61] 2022. Xcode. Retrieved May 01, 2022 from https://developer.apple.com/xcode/.

[62] Cristina L. Abad, Edwin F. Boza, and Erwin Van Eyk. 2018. Package-aware scheduling of FaaS functions. In *Proceedings of the Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 101–106.

[63] Gojko Adzic and Robert Chatley. 2017. Serverless computing: Economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 884–889.

[64] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. 419–434.

[65] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. Cose: Configuring serverless functions using statistical learning. In *Proceedings of the IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 129–138.

[66] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful functions as a service in action. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1890–1893.

[67] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 923–935.

[68] Zaid Al-Ali, Sepideh Goodarzy, Ethan Hunter, Sangtae Ha, Richard Han, Eric Keller, and Eric Rozner. 2018. Making serverless computing more serverless. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing*. IEEE, 456–459.

[69] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 185–199.

[70] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[71] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure serverless computing using dynamic information flow control. *Proceedings of the ACM on Programming Languages* 2, 118 (2018), 1–26.

[72] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. 2021. Cloud-scale runtime verification of serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 92–107.

[73] Moayad Alshangiti, Hitesh Sapkota, Pradeep K. Murukannaiah, Xumin Liu, and Qi Yu. 2019. Why is developing machine learning applications challenging? A study on stack overflow posts. In *Proceedings of 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 1–11.

[74] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.

[75] Gabriel Aumala, Edwin Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. 2019. Beyond load balancing: Package-aware scheduling for serverless platforms. In *Proceedings of the 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 282–291.

[76] Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. 2020. BlastFunction: An FPGA-as-a-service system for accelerated serverless computing. In *Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 852–857.

[77] Timon Back and Vasilios Andrikopoulos. 2018. Using a microbenchmark to compare function as a service solutions. In *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer, 146–160.

[78] Deepali Bajaj, Urmil Bharti, Anita Goel, and S. C. Gupta. 2020. Partial migration for re-architecting a cloud native monolithic application into microservices and faas. In *Proceedings of the International Conference on Information, Communication and Computing Technology*. Springer, 111–124.

[79] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 89–103.

[80] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 1–31.

[81] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*. 41–54.

[82] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful serverless computing with crucial. *ACM Transactions on Software Engineering and Methodology* 31, 3 (2022), 1–38.

[83] Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga, Sam Guinea, and Giovanni Quattrocchi. 2019. A unified model for the mobile-edge-cloud continuum. *ACM Transactions on Internet Technology* 19, 2 (2019), 1–21.

[84] David Bermbach, Jonathan Bader, Jonathan Hasenburg, Tobias Pfandzelter, and Lauritz Thamsen. 2022. Auction-Whisk: Using an auction-inspired approach for function placement in serverless fog platforms. *Software: Practice and Experience* 52, 5 (2022), 1143–1169.

[85] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 134–143.

[86] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. 2010. Cloud computing: A study of infrastructure as a service (IAAS). *International Journal of Engineering and Information Technology* 2, 1 (2010), 60–63.

[87] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 153–167.

[88] Jonas Bonér. 2016. *Reactive Microservices Architecture*. O'Reilly Media, Incorporated.

[89] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the "micro" back in microservice. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 645–650.

[90] Edwin F. Boza, Cristina L. Abad, Mónica Villavicencio, Stephany Quimba, and Juan Antonio Plaza. 2017. Reserved, on demand or serverless: Model-based simulations for cloud budget planning. In *Proceedings of the 2017 IEEE 2nd Ecuador Technical Chapters Meeting*. IEEE, 1–6.

[91] Lukas Brand and Markus Mock. 2021. SFL: A compiler for generating stateful aws lambda serverless applications. In *Proceedings of the 17th International Workshop on Serverless Computing*. 29–35.

[92] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip redundant paths to make serverless fast. In *Proceedings of the 15th European Conference on Computer Systems*. 1–15.

[93] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: Leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 58–64.

[94] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 1–15.

[95] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. 2019. In search of a fast and efficient serverless dag engine. In *Proceedings of the 2019 IEEE/ACM 4th International Parallel Data Systems Workshop*. IEEE, 1–10.

[96] Mohak Chadha, Anshul Jindal, and Michael Gerndt. 2021. Architecture-specific performance optimization of compute-intensive faas functions. In *Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing*. IEEE, 478–483.

[97] Dheeraj Chahal, Manju Ramesh, Ravi Ojha, and Rekha Singhal. 2021. High performance serverless architecture for deep learning workflows. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 790–796.

[98] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 750–762.

[99] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. 2019. Fog function: Serverless fog computing for data intensive iot services. In *Proceedings of the 2019 IEEE International Conference on Services Computing*. IEEE, 28–35.

[100] Angelos Christidis, Sotiris Moschoyiannis, Ching-Hsien Hsu, and Roy Davies. 2020. Enabling serverless deployment of large-scale ai workloads. *IEEE Access* 8 (2020), 70150–70161.

[101] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[102] Robert Cordingly, Wen Shu, and Wes J. Lloyd. 2020. Predicting performance and cost of serverless computing functions with SAAF. In *Proceedings of the IEEE International Symposium on Dependable, Autonomic and Secure Computing*. IEEE, 640–649.

[103] Domenico Cotroneo, Flavio Frattini, Roberto Pietrantuono, and Stefano Russo. 2015. State-based robustness testing of IaaS cloud platforms. In *Proceedings of the 5th International Workshop on Cloud Data and Platforms*. 1–6.

[104] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-mei Hwu. 2019. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. IEEE, 372–382.

[105] Anirban Das, Andrew Leaf, Carlos A. Varela, and Stacy Patterson. 2020. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing*. IEEE, 609–618.

[106] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of the Web Conference 2020*. 939–950.

[107] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.

[108] Leonardo Rebouças de Carvalho and Aletéia Patrícia Favacho de Araújo. 2019. Framework Node2FaaS: Automatic NodeJS application converter for function as a service.. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. 271–278.

[109] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[110] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 45–59.

[111] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*. 248–259.

[112] Simon Eismann, Diego Elias Costa, Lizhi Liao, Cor-Paul Bezemer, Weiyi Shang, André van Hoorn, and Samuel Kounev. 2022. A case study on the stability of performance tests for serverless applications. *Journal of Systems and Software* 189 (2022), 111294.

[113] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 265–276.

[114] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. 2021. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4152–4166.

[115] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2020. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (2020), 32–39.

[116] Adam Eivy and Joe Weinman. 2017. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing* 4, 2 (2017), 6–12.

[117] Tarek Elgamal. 2018. Costless: Optimizing cost of serverless computing through function fusion and placement. In *Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing*. IEEE, 300–312.

[118] Unai Elordi, Luis Unzueta, Jon Goenetxea, Sergio Sanchez-Carballido, Ignacio Arganda-Carreras, and Oihana Otaegui. 2020. Benchmarking deep neural network inference performance on serverless environments with MLPerf. *IEEE Software* 38, 1 (2020), 81–87.

[119] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. 2020. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems* 105 (2020), 361–379.

[120] Nafise Eskandani and Guido Salvaneschi. 2021. The wonderless dataset for serverless computing. In *Proceedings of 2021 IEEE/ACM 18th International Conference on Mining Software Repositories*. IEEE, 565–569.

[121] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. USETL: Unikernels for serverless extract transform and load why should you settle for less?. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. 23–30.

[122] Patrik Fortier, Frédéric Le Mouël, and Julien Ponge. 2021. Dyninka: A FaaS framework for distributed dataflow applications. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. 2–13.

[123] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 475–488.

[124] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. 363–376.

[125] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.

[126] Salvador Garcia, Julian Luengo, José Antonio Sáez, Victoria Lopez, and Francisco Herrera. 2012. A survey of dis-
cretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Transactions on Knowledge
and Data Engineering* 25, 4 (2012), 734–750.

[127] Nikita Gerasimov. 2019. The DSL for composing functions for FaaS platform. In *Proceedings of the 4th Conference on
Software Engineering and Information Management.*

[128] Alim Ul Gias and Giuliano Casale. 2020. Cocoa: Cold start aware capacity planning for function-as-a-service plat-
forms. In *Proceedings of the 2020 28th International Symposium on the Modeling, Analysis, and Simulation of Computer
and Telecommunication Systems.* IEEE, 1–8.

[129] Vicent Giménez-Alventosa, Germán Moltó, and Miguel Caballer. 2019. A framework and a performance assessment
for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems* 97 (2019), 259–274.

[130] Jake Grogan, Connor Mulready, James McDermott, Martynas Urbanavicius, Murat Yilmaz, Yalemisew Abgaz,
Andrew McCarren, Silvana Togneri MacMahon, Vahid Garousi, Peter Elger, et al. 2020. A multivocal literature re-
view of function-as-a-service (faas) infrastructures and implications for software developers. In *Proceedings of the
European Conference on Software Process Improvement.* Springer, 58–75.

[131] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis,
and Chita Das. 2019. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public
cloud. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing.* IEEE, 199–208.

[132] Adam Hall and Umakishore Ramachandran. 2019. An execution model for serverless functions at the edge. In *Pro-
ceedings of the International Conference on Internet of Things Design and Implementation.* 225–236.

[133] Tracy Hall, Nathan Baddoo, Sarah Beecham, Hugh Robinson, and Helen Sharp. 2009. A systematic review of theory
use in studies investigating the motivations of software engineers. *ACM Transactions on Software Engineering and
Methodology* 18, 3 (2009), 1–29.

[134] Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. 2021. Survey on serverless computing. *Journal of Cloud
Computing* 10, 1 (2021), 1–29.

[135] Shay Horovitz, Roei Amos, Ohad Baruch, Tomer Cohen, Tal Oyar, and Afik Deri. 2018. Faastest-machine learning
based cost and performance faas optimization. In *Proceedings of the International Conference on the Economics of
Grids, Clouds, Systems, and Services.* Springer, 171–186.

[136] MohammadReza HoseinyFarahabady, Young Choon Lee, Albert Y. Zomaya, and Zahir Tari. 2017. A qos-aware re-
source allocation controller for function as a service (faas) platform. In *Proceedings of the International Conference on
Service-Oriented Computing.* Springer, 241–255.

[137] M. Reza HoseinyFarahabady, Albert Y. Zomaya, and Zahir Tari. 2017. A model predictive controller for managing
QoS enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Transactions on Parallel and
Distributed Systems* 29, 7 (2017), 1442–1455.

[138] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020.
Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on
Software Engineering.* 1110–1121.

[139] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing.
Proceedings of the ACM on Programming Languages 3, 149 (2019), 1–26.

[140] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2021. AMPS-Inf: Automatic model partitioning for serverless
inference with cost efficiency. In *Proceedings of the 50th International Conference on Parallel Processing.* 1–12.

[141] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful serverless computing with shared logs. In *Proceedings of the
ACM SIGOPS 28th Symposium on Operating Systems Principles.* 691–707.

[142] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and scalable serverless computing for latency-sensitive,
interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Pro-
gramming Languages and Operating Systems.* 152–166.

[143] Anshul Jindal, Michael Gerndt, Mohak Chadha, Vladimir Podolskiy, and Pengfei Chen. 2021. Function delivery net-
work: Extending serverless computing for heterogeneous platforms. *Software: Practice and Experience* 51, 9 (2021),
1936–1963.

[144] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed
computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing.* 445–451.

[145] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar,
Joao Carreira, Karl Krauth, Neeraja Yadawkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A.
Patterson. 2019. Cloud programming simplified: A berkeley view on serverless computing. arXiv:1902.03383. Re-
trieved from https://arxiv.org/abs/1902.03383.

[146] Kostis Kaffes, Neeraja J. Yadawkar, and Christos Kozyrakis. 2019. Centralized core-granular scheduling for server-
less functions. In *Proceedings of the ACM Symposium on Cloud Computing.* 158–164.

[147] Alex Kaplunovich. 2019. ToLambda–automatic path to serverless architectures. In *Proceedings of the 2019 IEEE/ACM 3rd International Workshop on Refactoring*. IEEE, 1–8.

[148] Staffs Keele et al. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report. Technical report, ver. 2.3 ebse technical report. ebse.

[149] Joanna Kijak, Piotr Martyna, Maciej Pawlik, Bartosz Balis, and Maciej Malawski. 2018. Challenges for scheduling scientific workflows on cloud functions. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing*. IEEE, 460–467.

[150] Dong Kyoung Kim and Hyun-Gul Roh. 2021. Scheduling containers rather than functions for function-as-a-service. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 465–474.

[151] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. 2018. Gpu enabled serverless computing framework. In *Proceedings of the 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 533–540.

[152] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. IEEE, 502–504.

[153] Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, and Albert Y. Zomaya. 2020. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems* 31, 10 (2020), 2289–2301.

[154] Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, Albert Y. Zomaya, and Raja Jurdak. 2018. Dynamic control of CPU usage in a lambda platform. In *Proceedings of the 2018 IEEE International Conference on Cluster Computing*. IEEE, 234–244.

[155] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 427–444.

[156] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating function-as-a-service workflows. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 805–820.

[157] Kyriakos Kritikos, Pawel Skrzypek, Alexandru Moga, and Oliviu Matei. 2019. Towards the modelling of hybrid cloud applications. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. IEEE, 291–295.

[158] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174.

[159] Seungjun Lee, Daegun Yoon, Sangho Yeo, and Sangyoon Oh. 2021. Mitigating cold start problem in serverless computing with function fusion. *Sensors* 21, 24 (2021), 8416.

[160] Youngsoo Lee and Sunghee Choi. 2021. A greedy load balancing algorithm for faaS platforms. In *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing*. 63–69.

[161] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software* 149 (2019), 340–359.

[162] Valentina Lenarduzzi, Jeremy Daly, Antonio Martini, Sebastiano Panichella, and Damian Andrew Tamburri. 2020. Toward a technical debt conceptualization for serverless computing. *IEEE Software* 38, 1 (2020), 40–47.

[163] Valentina Lenarduzzi and Annibale Panichella. 2020. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software* 38, 1 (2020), 54–60.

[164] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Cheng-Zhong Xu. 2022. Serverless computing: State-of-the-art, challenges and opportunities. *IEEE Transactions on Services Computing* (2022).

[165] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. 2021. The serverless computing survey: A technical primer for design architecture. *Computing Surveys* 54, 10s (2021), 1–34.

[166] Bin Lin, Nathan Cassee, Alexander Serebrenik, Gabriele Bavota, Nicole Novielli, and Michele Lanza. 2022. Opinion mining for software development: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 31, 3 (2022), 1–41.

[167] Changyuan Lin and Hamzeh Khazaei. 2020. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 615–632.

[168] Wei Ling, Lin Ma, Chen Tian, and Ziang Hu. 2019. Pigeon: A dynamic and efficient serverless and FaaS framework for private cloud. In *Proceedings of the 2019 International Conference on Computational Science and Computational Intelligence*. 1416–1421.

[169] Xuanzhe Liu, Gang Huang, Qi Zhao, Hong Mei, and M. Brian Blake. 2014. iMashup: A mashup-based framework for service composition. *Science China Information Sciences* 57, 1 (2014), 1–20.

[170] Xueqing Liu, Yangqiu Song, Shixia Liu, and Haixun Wang. 2012. Automatic taxonomy construction from keywords. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1433–1441.

[171] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *Proceedings of the 2018 IEEE International Conference on Cloud Engineering*. IEEE, 159–169.

[172] Sin Kit Lo, Qinghua Lu, Chen Wang, Hye-Young Paik, and Liming Zhu. 2021. A systematic literature review on federated machine learning: From a software engineering perspective. *ACM Computing Surveys* 54, 5 (2021), 1–39.

[173] Ju Long, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. 2020. A lightweight design for serverless function as a service. *IEEE Software* 38, 1 (2020), 75–80.

[174] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: Symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 617–628.

[175] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. 2021. FaaSTCC: Efficient transactional causal consistency for serverless computing. In *Proceedings of the 22nd International Middleware Conference*. 159–171.

[176] Kunal Mahajan, Daniel Figueiredo, Vishal Misra, and Dan Rubenstein. 2019. Optimal pricing for serverless computing. In *Proceedings of the 2019 IEEE Global Communications Conference*. IEEE, 1–6.

[177] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. SONIC: Application-aware data passing for chained serverless applications. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 285–301.

[178] Nima Mahmoudi and Hamzeh Khazaei. 2020. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing* 10, 4 (2020), 2834–2847.

[179] Nima Mahmoudi and Hamzeh Khazaei. 2020. Temporal performance modelling of serverless computing platforms. In *Proceedings of the 2020 6th International Workshop on Serverless Computing*. 1–6.

[180] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. 2019. Optimizing serverless computing: Introducing an adaptive function placement algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 203–213.

[181] Marcin Majewski, Maciej Pawlik, and Maciej Malawski. 2021. Algorithms for scheduling scientific workflows on serverless architecture. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 782–789.

[182] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2021. Deadline-aware dynamic resource management in serverless computing environments. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 483–492.

[183] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2021. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys* 54, 11s (2021), 1–36.

[184] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering* 43, 9 (2016), 817–847.

[185] Vadym Maslov and Andriy Petrashenko. 2021. Distributed serverless computing orchestration based on finite automaton. In *Proceedings of the International Conference on Computer Science, Engineering and Education Applications*. Springer, 290–303.

[186] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. 2018. Retro-λ: An event-sourced platform for serverless applications with retroactive computing support. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. 76–87.

[187] Peter Mell and Timothy Grance. 2011. The NIST definition of cloud computing. *Special Publication* 800 (2011), 145.

[188] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*.

[189] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. 2018. Cognitive biases in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1318–1339.

[190] Ajdin Mujezinović and Vedran Ljubović. 2019. Serverless architecture for workflow scheduling with unconstrained execution environment. In *Proceedings of the 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, 242–246.

[191] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. 2021. OFC: An opportunistic caching system for FaaS platforms. In *Proceedings of the 16th European Conference on Computer Systems*. 228–244.

[192] Shripad Nadgowda, Nilton Bila, and Canturk Isci. 2017. The less server architecture for cloud functions. In *Proceedings of the 2nd International Workshop on Serverless Computing*. 22–27.

[193] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *Journal of Parallel and Distributed Computing* 139 (2020), 32–42.

[194] Sam Newman. 2021. *Building Microservices*. "O'Reilly Media, Inc.".

[195] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*. 1–6.

[196] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Pipsqueak: Lean lambdas with large libraries. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems Workshops*. IEEE, 395–400.

[197] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 57–70.

[198] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static call graph construction in aws lambda serverless applications. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*.

[199] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. 2020. Allocation priority policies for serverless function-execution scheduling optimisation. In *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 416–430.

[200] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–74.

[201] Maciej Pawlik, Pawel Banach, and Maciej Malawski. 2019. Adaptation of workflow application scheduling algorithm to serverless infrastructure. In *Proceedings of the European Conference on Parallel Processing*. Springer, 345–356.

[202] KJPG Perera and I. Perera. 2018. A rule-based system for automated generation of serverless-microservices architecture. In *Proceedings of the IEEE International Systems Engineering Symposium*. IEEE, 1–8.

[203] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. 2018. Serverless computing for container-based architectures. *Future Generation Computer Systems* 83 (2018), 50–59.

[204] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. 2019. A programming model and middleware for high throughput serverless computing applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 106–113.

[205] Tobias Pfandzelter and David Bermbach. 2020. tinyfaas: A lightweight faas platform for edge environments. In *Proceedings of the 2020 IEEE International Conference on Fog Computing*. IEEE, 17–24.

[206] Bartłomiej Przybylski, Paweł Żuk, and Krzysztof Rzadca. 2021. Data-driven scheduling in serverless computing to reduce response time. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 206–216.

[207] Alexandros Psychas, John Violos, Fotis Aisopos, Athanasia Evangelinou, George Kousiouris, Ioannis Bouras, Theodora Varvarigou, Gerasimos Xidas, Dimitrios Charilas, and Yiannis Stavroulas. 2020. Cloud toolkit for provider assessment, optimized application cloudification and deployment on IaaS. *Future Generation Computer Systems* 109 (2020), 657–667.

[208] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*. 193–206.

[209] Chen Qian and Wenjing Zhu. 2020. F(X)-MAN: An algebraic and hierarchical composition model for function-as-a-service. In *Proceedings of the 32nd International Conference on Software Engineering and Knowledge Engineering*. 210–215.

[210] Weizhong Qiang, Zezhao Dong, and Hai Jin. 2018. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *Proceedings of the International Conference on Security and Privacy in Communication Systems*. Springer, 451–470.

[211] Shijun Qin, Heng Wu, Yuewen Wu, Bowen Yan, Yuanjia Xu, and Wenbo Zhang. 2020. Nuka: A generic engine with millisecond initialization for serverless computing. In *Proceedings of the 2020 IEEE International Conference on Joint Cloud Computing*. IEEE, 78–85.

[212] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. 2021. A case for function-as-a-service with disaggregated FPGAs. In *Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing*. IEEE, 333–344.

[213] Sasko Ristov, Stefan Pedratscher, Jakob Wallnoefer, and Thomas Fahringer. 2020. Daf: Dependency-aware faasifier for node.js monolithic applications. *IEEE Software* 38, 1 (2020), 48–53.

[214] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. FaaT: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 122–137.

[215] Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, and Antonio Corradi. 2021. A shared memory approach for function chaining in serverless platforms. In *Proceedings of the 2021 IEEE Symposium on Computers and Communications*. IEEE, 1–6.

[216] Aakanksha Saha and Sonika Jindal. 2018. EMARS: Efficient management and allocation of resources in serverless. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing*. IEEE, 827–830.

[217] Fatima Samea, Farooque Azam, Muhammad Waseem Anwar, Mehreen Khan, and Muhammad Rashid. 2019. A UML profile for multi-cloud service configuration (UMLPMSC) in event-driven serverless applications. In *Proceedings of the 2019 8th International Conference on Software and Computer Applications*. 431–435.

[218] Josep Sampé, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2020. Toward multicloud access transparency in serverless computing. *IEEE Software* 38, 1 (2020), 68–74.

[219] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-driven serverless functions for object storage. In *Proceedings of the 18th International Middleware Conference*. 121–133.

[220] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. 2020. Primula: A practical shuffle/sort operator for serverless computing. In *Proceedings of the 21st International Middleware Conference Industrial Track*. 31–37.

[221] Arnav Sankaran, Pubali Datta, and Adam Bates. 2020. Workflow integration alleviates identity and access management in serverless computing. In *Proceedings of the Annual Computer Security Applications Conference*. 496–509.

[222] Lucia Schuler, Somaya Jamil, and Niklas Kühl. 2021. AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. In *Proceedings of 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 804–811.

[223] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572.

[224] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 205–218.

[225] Jingbo Shang, Xinyang Zhang, Liyuan Liu, Sha Li, and Jiawei Han. 2020. Nettaxo: Automated topic taxonomy construction from text-rich network. In *Proceedings of the Web Conference 2020*. 1908–1919.

[226] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 281–295.

[227] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2021. Defuse: A dependency-guided function scheduler to mitigate cold starts on faaS platforms. In *Proceedings of the 2021 IEEE 41st International Conference on Distributed Computing Systems*. IEEE, 194–204.

[228] K. R. Sheshadri and J. Lakshmi. 2021. QoS aware FaaS platform. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 812–819.

[229] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 419–433.

[230] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*. 138–152.

[231] Dalia Sobhy, Rami Bahsoon, Leandro Minku, and Rick Kazman. 2021. Evaluation of software architectures under uncertainty: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 30, 4 (2021), 1–50.

[232] Khondokar Solaiman and Muhammad Abdullah Adnan. 2020. WLEC: A not so cold architecture to mitigate cold start problem in serverless computing. In *Proceedings of the 2020 IEEE International Conference on Cloud Engineering*. IEEE, 144–153.

[233] Boubaker Soltani, Afifa Ghenai, and Nadia Zeghib. 2018. Towards distributed containerized serverless architecture in multi cloud environment. *Procedia Computer Science* 134 (2018), 121–128.

[234] Josef Spillner. 2020. Resource management for cloud functions with memory tracing, profiling and autotuning. In *Proceedings of the 2020 6th International Workshop on Serverless Computing*. 13–18.

[235] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A fault-tolerance shim for serverless computing. In *Proceedings of the 15th European Conference on Computer Systems*. 1–15.

[236] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2438–2452.

[237] Adam Stafford, Farshad Ghassemi Toosi, and Anila Mjeda. 2021. Cost-aware migration to functions-as-a-service architecture. In *Proceedings of the ECSA 2021 Companion Volume, Virtual*.

[238] R. Sujatha, R. Bandaru, and R. Rao. 2011. Taxonomy construction techniques–issues and challenges. *Indian Journal of Computer Science and Engineering* 2, 5 (2011), 661–671.

[239] Kun Suo, Junggab Son, Dazhao Cheng, Wei Chen, and Sabur Baidya. 2021. Tackling cold start of serverless applications by efficient and adaptive container runtime reusing. In *Proceedings of the 2021 IEEE International Conference on Cluster Computing*. IEEE, 433–443.

[240] Amoghvarsha Suresh and Anshul Gandhi. 2019. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*. 19–24.

[241] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems*. IEEE, 1–10.

[242] Davide Taibi, Josef Spillner, and Konrad Wawruch. 2020. Serverless computing-where are we now, and where are we heading? *IEEE Software* 38, 1 (2020), 25–31.

[243] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. 2020. Towards lightweight serverless computing via unikernel as a function. In *Proceedings of the 2020 IEEE/ACM 28th International Symposium on Quality of Service*. IEEE, 1–10.

[244] Yang Tang and Junfeng Yang. 2020. Lambdata: Optimizing serverless computing by making data intents explicit. In *Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing*. IEEE, 294–303.

[245] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 311–327.

[246] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[247] Erwin Van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2019. The spec-rg reference architecture for faas: From microservices and containers to serverless platforms. *IEEE Internet Computing* 23, 6 (2019), 7–18.

[248] Erwin Van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, and Alexandru Iosup. 2018. Serverless is more: From PaaS to present cloud computing. *IEEE Internet Computing* 22, 5 (2018), 8–17.

[249] Adbys Vasconcelos, Lucas Vieira, Ítalo Batista, Rodolfo Silva, and Francisco V Brasileiro. 2019. DistributedFaaS: Execution of containerized serverless applications in multi-cloud infrastructures.. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. 595–600.

[250] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaS-Net: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 443–457.

[251] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.

[252] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the 14th EuroSys Conference 2019*. 1–16.

[253] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 133–146.

[254] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 1–58.

[255] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data analytics on network-enabled serverless platforms. In *Proceedings of the 11th Annual Conference on Innovative Data Systems Research*.

[256] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 416–428.

[257] Jinfeng Wen and Yi Liu. 2021. A measurement study on serverless workflow services. In *Proceedings of the 2021 IEEE International Conference on Web Services*. IEEE, 741–750.

[258] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Junkai Chen, and Yun Ma. 2021. Characterizing commodity serverless computing platforms. *Journal of Software: Evolution and Process* (2021), e2394.

[259] Stefan Winzinger and Guido Wirtz. 2021. Data flow testing of serverless functions. In *Proceedings of the International Conference on Cloud Computing and Services Science*. 56–64.

[260] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. 2019. Cspot: Portable, multi-scale functions-as-a-service for IOT. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 236–249.

[261] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional causal consistency for serverless computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 83–97.

[262] Song Wu, Zhiheng Tao, Hao Fan, Zhuo Huang, Xinmin Zhang, Hai Jin, Chen Yu, and Chun Cao. 2022. Container lifecycle-aware scheduling for serverless computing. *Software: Practice and Experience* 52, 2 (2022), 337–352.

[263] Michael Wurster, Uwe Breitenbücher, Kálmán Képes, Frank Leymann, and Vladimir Yussupov. 2018. Modeling and automated deployment of serverless applications using TOSCA. In *Proceedings of the 2018 IEEE 11th Conference on Service-oriented Computing and Applications*. IEEE, 73–80.

[264] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. 2019. Adaptive function launching acceleration in serverless computing platforms. In *Proceedings of the IEEE 25th International Conference on Parallel and Distributed Systems*. 9–16.

[265] Bowen Yan, Heran Gao, Heng Wu, Wenbo Zhang, Lei Hua, and Tao Huang. 2020. Hermes: Efficient cache management for container-based serverless computing. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. 136–145.

[266] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating serverless computing by harvesting idle resources. In *Proceedings of the ACM Web Conference*. 1741–1751.

[267] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proceedings of the 2021 IEEE 41st International Conference on Distributed Computing Systems*. IEEE, 138–148.

[268] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 2020 ACM Symposium on Cloud Computing*. 30–44.

[269] Vladimir Yussupov, Uwe Breitenbücher, Ayhan Kaplan, and Frank Leymann. 2020. SEAPORT: Assessing the portability of serverless applications. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. 456–467.

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

[271] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, and Frank Leymann. 2022. Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA. *Software: Practice and Experience* 52, 6 (2022), 1454–1495.

[272] N. Yuvaraj, T. Karthikeyan, and K. Praghash. 2021. An improved task allocation scheme in serverless computing using gray wolf Optimization (GWO) based reinforcement learning (RIL) approach. *Wireless Personal Communications* 117, 3 (2021), 2403–2421.

[273] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 1187–1204.

[274] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE task scheduling for serverless analytics. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*. 653–669.

[275] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.

[276] Michael Zhang, Chandra Krintz, and Rich Wolski. 2021. Edge-adaptable serverless acceleration for machine learning Internet of Things applications. *Software: Practice and Experience* 51, 9 (2021), 1852–1867.

[277] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–12.

[278] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 328–343.

[279] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 724–739.

[280] Ge Zheng and Yang Peng. 2019. GlobalFlow: A cross-region orchestration service for serverless computing services. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. IEEE, 508–510.

[281] Lulai Zhu, Giorgos Giotis, Vasilis Tountopoulos, and Giuliano Casale. 2021. RDOF: Deployment optimization for function as a service. In *Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing*. IEEE, 508–514.

[282] Pawel Zuk and Krzysztof Rzadca. 2020. Scheduling methods to reduce response latency of function as a service. In *Proceedings of the 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing*. 132–140.