The Gap Between Serverless Research and Real-world Systems

Qingyuan Liu[1,2], Dong Du[1,2], Yubin Xia[1,2,3], Ping Zhang[4], Haibo Chen[1,2]
[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Engineering Research Center for Domain-specific Operating Systems (MoE)
[3]Shanghai AI Laboratory
[4]Huawei Cloud

ABSTRACT

With the emergence of the serverless computing paradigm in the cloud, researchers have explored many challenges of serverless systems and proposed solutions such as snapshot-based booting. However, we have noticed that some of these optimizations are based on oversimplified assumptions that lead to infeasibility and hide real-world issues. This paper aims to analyze the gap between current serverless research and real-world systems from a perspective of industry, and present new observations, challenges, opportunities, and insights that may address the discrepancies.

CCS CONCEPTS

• **Networks → Cloud computing**.

KEYWORDS

serverless, cloud computing, scheduling, sidecar

ACM Reference Format:
Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, Haibo Chen. 2023. The Gap Between Serverless Research and Real-world Systems. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3620678.3624785

1 INTRODUCTION

Serverless computing [40] has become an emerging paradigm of today's cloud and data center infrastructures [2, 4, 6, 8]. It uses one single-purpose service or function as the basic computation unit, which eases computing in several ways. First, it helps application developers focus on the core logic, and leaves infrastructure-related tasks like auto-scaling to the serverless platform. Second, it adopts the "pay-as-you-go" model with fine-grained charging granularity (e.g., 1ms [3]) so that users can save costs for unused computing resources. Third, serverless computing also benefits cloud providers such that they can manage their resources more efficiently.

There are already many optimizations of serverless systems. However, despite these efforts, there is still a significant gap between current solutions and efficient, practical serverless systems. For example, many researches have attempted to optimize instance initialization [5, 7, 9, 33, 39, 52, 59, 64, 66] to reduce cold start latency. Nevertheless, even if these optimizations successfully reduce the initialization overhead to <10ms, on real serverless systems such as Knative [13] on top of Kubernetes (K8s) [21], it still requires hundreds of milliseconds to start an instance. The reason for this is that cold start latency is not only made up of instance initialization costs but also other overheads that are overlooked by research works. For example, Knative relies on K8s state synchronization to deliver scale-out requests, which incurs non-trivial communication costs (§3). Additionally, before initialization, the K8s scheduler must choose a node for the new instance, which incurs scheduling costs (§4). These overlooked overheads represent gaps and highlight the discrepancy between common assumptions in research work and actual industry situations. It is essential to realize these gaps to make serverless research steps further.

This paper presents five open challenges observed in real-world serverless systems and industry practices to illustrate the gaps, as shown in Table 1. Firstly, we emphasize that instance cold starts are usually out of the critical path of end-to-end latency. Cold start latency does not directly affect end-to-end latency but does so indirectly through queuing, scheduling, and routing policies. Secondly, many serverless systems based on K8s take advantages of its declarative feature, which provides benefits such as cluster management simplicity. However, it can also introduce drawbacks like higher and uncertain end-to-end latency. Thirdly, scheduling costs can be significant in terms of latency and resource consumption, especially in large-scale clusters with more

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624785

Table 1: Gaps between research and real-world systems.

Challenges	Research works	Real-world systems
Cold start	Synchronous	Asynchronous
Declarative tax	Hardly consider	Non-trivial cost
Scheduling cost	<100ms (<120 nodes)	>10s (2K nodes)
Scheduling policy	Single policy	≥20 policies
Sidecar	Hardly consider	Non-trivial cost

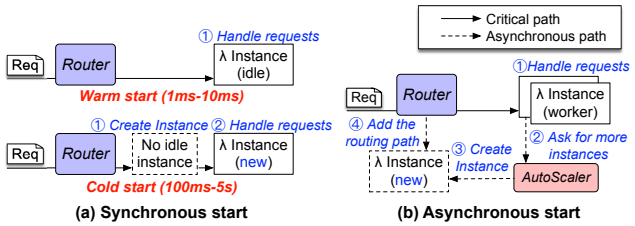


Figure 1: Synchronous start v.s. asynchronous start.

than 1,000 nodes. Therefore, an efficient scheduler for serverless platforms can be critical. Fourthly, designing a single scheduling policy may not be sufficient in real schedulers such as the K8s scheduler. Typically, more than 20 scheduling policies are used in conjunction and scheduling choices are considered holistically. However, due to a lack of balancing among these policies, the final scheduling results may be suboptimal. Fifthly, we focus on the sidecar component, which is often overlooked in research work but is very common in real-world serverless systems. We discuss the advantages of sidecars in terms of functionality and modularity, but also mention the additional overhead this component may bring.

By outlining these challenges, we hope to inspire further research in the area and enable researchers to advance state-of-the-art serverless systems. Also, since addressing these challenges necessitates cooperation between academia and industry, recognizing the challenges is an essential step, or the initial step, for such cooperation.

Methodology. The challenges and insights presented in this paper are drawn mainly from our experience of applying research optimizations to real-world serverless systems (FunctionGraph, Huawei Cloud). While research has made significant progress, we find that further steps are necessary to fully tackle real-world problems. To bridge the gap between research and industry, we examine common assumptions and issues in research work while also highlighting points easily overlooked from the industry perspective. This paper does not cover all prevalent topics in serverless computing, e.g., heterogeneous serverless [32]. Although we mostly use K8s and serverless platforms like Knative as examples, the challenges and gaps discussed are general for serverless computing.

2 CHALLENGE I: ASYNCHRONOUS START

Optimizing cold start latency is crucial for serverless computing for two primary reasons. Firstly, serverless has a much higher frequency of cold starts than other cloud paradigms due to its on-demand provisioning nature. Secondly, cold starts can significantly increase the end-to-end latency of requests [33], which can negatively affect user experience. As a result, there is a lot of research work [5, 7, 9, 25, 26, 33, 39, 42, 44, 52, 59, 64, 66, 68, 76] being conducted to optimize cold start latency in serverless computing.

Synchronous and asynchronous starts. However, we observe a mismatch between the cold starts considered in research works and real-world systems. Specifically, most existing works only consider *synchronous starts*, where a request can be directly handled by an instance if there are idle instances available (warm start); otherwise, the request is blocked until a new instance is created and then handled by the new instance (cold start), as shown in Figure 1 (a). However, in industry, such as in our cloud, serverless platforms use *asynchronous starts*, as shown in Figure 1 (b). In this case, for every incoming request, the *router* always delivers the request to a function instance (if there are any) without explicit start operations. Instead, the system deploys a dedicated component, usually called *AutoScaler*, to monitor metrics (e.g., RPS, tail latency, etc.) and launch new instances asynchronously when existing instances are insufficient to handle loads. Serverless systems utilize a *queue* to buffer pending requests until they are handled by active instances.

Gaps & challenges. With asynchronous starts, cold start latency is not only related to instance creation, such as preparing a Docker container [52], but also other factors, such as the queue size and request rate. To illustrate the implications of asynchronous starts on end-to-end latency, let us consider the specific implementation of Knative [13], as shown in Figure 2. Knative uses a per-instance queue design, where the router delivers requests to instances consisting of a queue container and an application container. The queue container buffers incoming requests and delivers them one-by-one to the application container to handle. This per-instance queue design is common in reality, for example, our public serverless platform also applies the approach.

To illustrate how asynchronous starts can affect queuing and thus tail latency, we can consider a simplified scenario where only one instance exists in the cluster at the beginning. As requests continue to arrive, if the number of requests waiting in the queue exceeds a threshold value L, a *scale-out* command is triggered to create a new instance. For simplicity, let the execution latency of each request be d, and the internal arrival time (IAT) between two requests be a fixed value σd ($0 < \sigma < 1$). In this case, the queue will continue to grow

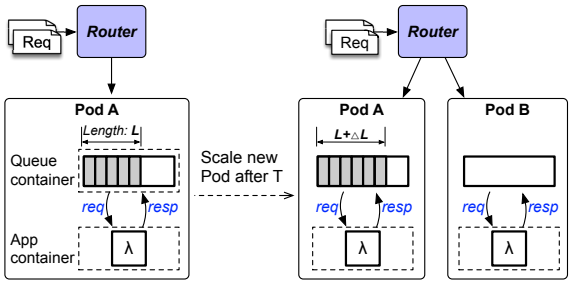


Figure 2: Knative with per-instance queue.

until it reaches length L at time t_{scale} as follows:

$$N(t_{scale}) = (\frac{1}{\sigma d} - \frac{1}{d})t_{scale} = L \Rightarrow t_{scale} = \frac{L\sigma d}{1-\sigma}$$

Suppose the instance initialization latency is T. Before the second instance is created, new requests continue to be sent to the first instance, and the number of requests in the instance's queue continues to grow. As a result, when the second instance is initialized, the latency of the last request in the queue is:

$$Tail\_e2e\_latency = N(T + t_{scale})d = Ld + T\frac{1-\sigma}{\sigma}$$

As per the example, the impact of the initialization latency on the end-to-end latency is magnified by other factors, including the queue size, execution time, and arrival rate of requests. Since all the factors affect the end-to-end latency, merely accelerating instance initialization may not be sufficient for achieving good performance. Therefore, it is essential to consider other factors, such as proactive auto-scaling policies and queue design, for reducing the end-to-end latency. Even though prior scheduling systems designed for serverless computing [37, 41, 60, 62] and other distributed systems [27, 38, 43, 49, 53, 58] have optimized end-to-end latency through special queuing or scheduling methods, However, few of them consider the impact of asynchronous starts, which is unique and common in the serverless scenario, so their design choices may have limitations in some cases.

Asynchronous starts could be influential with plentiful instance scaling, typically with load spikes. In some cases, for example, when the request interval (σ in the example) is small, the queuing latency can affect the tail latency more significantly than the instance initiation latency.

Opportunities & suggestions. Considering asynchronous startup, it is important to rethink design choices of prior works on various aspects, such as scheduling, routing, or queuing. Take queuing as an example, when constructing the queuing model to help design the queuing policy, the dynamic change in the number of queues, i.e., asynchronous starts, should be considered as a new dimension. With this new consideration, minor adjustments in design choices have the potential to yield significant performance improvements.

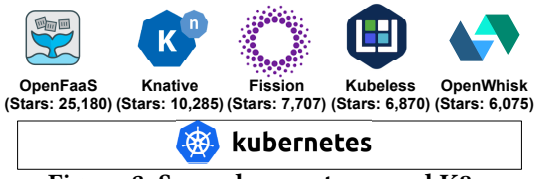


Figure 3: Serverless systems and K8s.

For instance, Hermod [41] notices and discusses the performance of different queuing choices, such as late binding or early binding. Late binding has a centralized gateway or router with a queue, while early binding has per-instance queues (e.g., Knative). Hermod chooses early binding, which employs a processor-sharing mechanism to execute functions for better end-to-end latency. The choice achieves great performance under its scope, where the number of instances is fixed regardless of scaling. However, there are trade-offs that in cases where asynchronous cold starts are frequent, late binding can better balance the queuing sizes of new and old instances. Continuing with the prior example, if we use late binding with a centralized queue, when the second instance is created, the requests in the queue can be consumed by both instances rather than a single instance (i.e., the first instance), resulting in halved tail latency, i.e., about 2x better:

$$Tail\_e2e\_latency = \frac{Ld + T\frac{1-\sigma}{\sigma}}{2}$$

Insight I

> *With the introduction of asynchronous cold starts, there are novel opportunities for specialized designs for systems such as scheduling or queuing systems to further optimize end-to-end latencies in serverless computing. Design choices made based on assumptions of synchronous start may need to be reevaluated in the context of asynchronous start.*

3 CHALLENGE II: DECLARATIVE TAX

Serverless systems often utilize a low-level infrastructure to manage instances in a large-scale cluster, with K8s [21] being the most widely-used infrastructure today, as shown in Figure 3. However, it is important to consider *whether K8s is the right system* for serverless computing. In practice, we observe that K8s' *declarative* approach brings both benefits and challenges to serverless platforms.

The declarative approach of K8s. K8s adopts a declarative approach to manage resources in the cluster, where any component can declare its "expected state" of some resources, and the corresponding *controllers* (a term in K8s) are responsible for making the resource meet the expected state. Some serverless platforms [1, 12, 13, 18, 19, 23], for example OpenFaaS, utilize this feature to implement auto-scalability, as

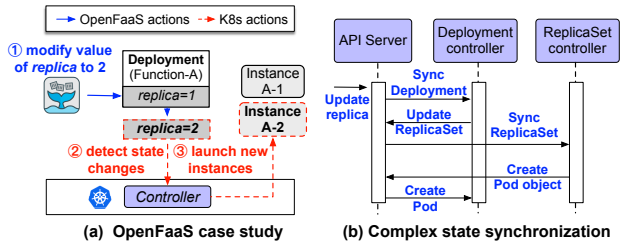


Figure 4: OpenFaaS case study.

shown in Figure 4 (a). OpenFaaS implements serverless functions using K8s *Deployment*, which has a state named *replica* to represent the number of instances of the function. To scale-out, OpenFaaS simply modifies the value of *replica* instead of manually creating new instances, and then the infrastructure will eventually launch new instances as expected.

While the declarative approach simplifies the management of resources and enables features like auto-scalability, it also introduces challenges for serverless platforms.

Gaps & challenges. The declarative method used by K8s requires many communications and synchronization between multiple components, incurring non-trivial overheads. Each operation through API server takes a few milliseconds [20], while a complete state synchronization process requires multiple communications through the API server. For example, Figure 4 (b) shows the essential synchronization messages required to create a new instance, which includes at least six messages through API server. In serverless systems (such as Knative) which extend K8s with customized objects, synchronization round trips become even more complex. With current research demonstrating sub-millisecond instance initialization times during cold start (as seen in works like Catalyzer [33] and Faasm [59]), the declarative tax has the potential to become the newest performance constraint.

The declarative method also leads to serious uncertainty and makes real-time operations or microsecond-scale latency difficult. Taking OpenFaaS as an example (Figure 4), OpenFaaS has no idea which component in the cluster will handle the scaling request and how fast they can handle it. This makes it challenging for the system to provide deterministic performance guarantees.

Furthermore, programming the controller is difficult. Tracing each invocation chain requires understanding the correspondence between various events and operations of multiple components.

In conclusion, the declarative tax could be the bottleneck when the cloud application has the requirements of low or real-time latencies. It could be increasingly influential as the complexity of applications enhances, involving more cloud/K8s resources, states and controllers.

Opportunities & suggestions. Solving the challenge brings new opportunities for optimizing low-level infrastructure system design. It requires achieving the goals simultaneously: *achieving great performance* while *providing an easy-to-use interface* (such as declarative APIs) with *highly modular infrastructure* to simplify development for both application and infrastructure developers. The suggestion is that, researchers can explore optimizing individual components, for example, speeding up the synchronization via API server/etcd, or adjusting the queuing mechanism within controllers to reduce the latency variation caused by them, etc. Moreover, with modularity carefully ensured, optimizing multiple components jointly can also be effective in improving performance, for example, providing fast paths for state synchronization, or exploring hardware-software co-design, etc. By addressing these challenges and pursuing these opportunities, we can improve the performance and usability of serverless platforms, making them more efficient and effective for a broad range of applications and use cases.

Insight II

> *Existing infrastructure systems used by serverless platforms, such as K8s, often use a declarative approach, which can result in higher and uncertain end-to-end latency. Designing a new mechanism to optimize the costs of the declarative approach without losing the benefits is an important challenge in the field.*

4 CHALLENGE III: SCHEDULING COST

The scheduling cost is part of cold start latency. Serverless schedulers in modern cloud platforms generally refer to control plane components that select a node for a new instance to run, as shown in Figure 5. For every instance's cold start, the scheduler must first determine where the instance can be deployed using a series of scheduling policies before the instance can be initialized on the target node. The costs involved in a serverless scheduler making a scheduling decision are known as *scheduling costs*.

Gaps & challenges. Although the scheduling costs are often treated as minor, we observe that they can be critical in large-scale clusters. The problem can be illustrated with a quantitative analysis, evaluating the scheduling overhead of K8s scheduler [15] on a large scale cluster using Kwok [17]. K8s scheduler is widely used by serverless platforms such as Knative [13], OpenFaaS [19], and OpenWhisk [1], making it a representative model for serverless schedulers.

Figure 6 presents the average costs to schedule each Pod with various numbers of concurrently created Pods. As the number of Pods increases from 2,000, the average scheduling cost or latency increases to ~14.5 seconds. Comparing these scheduling costs to the instance start cost (usually 10ms~3s), scheduling overhead (measured in seconds or tens of seconds) is orders of magnitude higher than start-up overhead

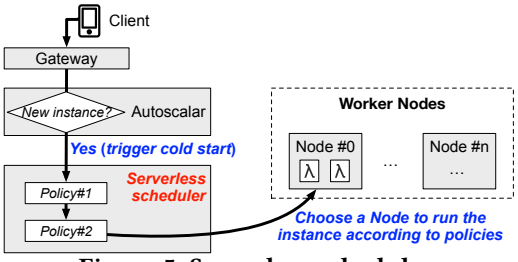


Figure 5: Serverless scheduler.

(~100x). Furthermore, when the number of Pods grows from 1,000 to 10,000, the memory cost of the scheduler can increase significantly from about 1 to 10 GB. The memory cost mostly comes from the cache of the scheduler, which caches detailed information about nodes and Pods in memory.

According to the analysis, the gaps can be exhibited in several ways. To begin with, existing scheduling works mostly consider scheduling costs on a small scale, leading to a failure to recognize the unacceptable growth of scheduling overheads in large-scale situations, as shown in Table 2. Based on our evaluation (Figure 6), scheduling costs are negligible when the number of concurrent Pods is less than 200, taking less than 10 milliseconds. However, scheduling costs grow proportionally with the number of concurrent Pods. In view of modern serverless platforms [28] (like clusters that can support up to 2,000 nodes and deploy up to 250 Pods per node in our public serverless platform), large-scale scenarios are increasingly critical considering scheduling costs.

Moreover, it shows that prior scheduling policies with complex calculations [47, 48, 69, 71, 77] are often infeasible for real-world platforms. These policies utilize machine learning models to quantify resource interference and achieve targets like maximizing resource utilization, as in the case of Gsight [77]. Here, a Random Forest Regression model predicts the performance of serverless functions to optimize function density while guaranteeing QoS. However, according to its evaluation, it requires costly model inference (additional tens of milliseconds) when scheduling every instance. When numerous concurrent Pods are waiting to be scheduled, Gsight scheduler predicts their performance one-by-one, making it unacceptable in large-scale scenarios.

In a nutshell, the analysis shows that scheduling costs play an important role, especially in large-scale scenarios with numerous concurrently scheduled instances, for applications that are sensitive to cold start latencies.

Opportunities & suggestions. Firstly, the gap shows the value of research in designing scalable scheduling policies. Moreover, it also motivates optimizing the scalability of scheduling systems. For example, even though the scheduling policy used in our evaluation (Figure 6) has relatively low computational cost [11] (around 4.55ms in our evaluation), it still results in high scheduling overhead as the

Table 2: The scale (number of nodes) of prior serverless scheduler researches.

Schedulers	Gsight [77]	Hermod [41]	Fifer [37]	Owl [63]
Scale	8	9–100	80	120

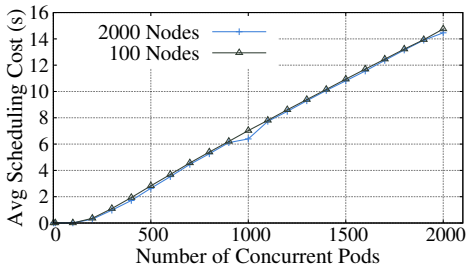


Figure 6: Average Scheduling Overhead.

scale of the cluster increases. So, rather than the policy, the unscalable design of K8s, i.e., the binding process in this example, can also become the bottleneck. Thus, eliminating the unscalable design in K8s and its scheduler, and applying mechanisms such as parallel binding, becomes essential to avoid non-trivial costs at large scales.

In addition, since large-scale cluster testing can be cost-prohibitive, we recommend using simulation tools such as Kwok [17] as an alternative, which can model thousands of virtual nodes and Pods using only a few real nodes. Industry professionals have used Kwok for large-scale evaluations. Additionally, innovative simulation techniques offer promising avenues for advancing our understanding and capabilities of serverless and cloud-native systems. These techniques can provide valuable insights into system behaviors under different conditions and help develop robust and efficient scheduling policies. Therefore, we encourage further exploration of simulation techniques, which is a cost-effective mean of advancing research in this area.

In conclusion, both scheduling policies and frameworks cannot ignore scalability requirements. Addressing the challenge requires innovative solutions that can provide efficient and scalable scheduling mechanisms for serverless platforms.

Insight III

> *Scheduling costs are often overlooked but can be non-trivial, particularly in large-scale scenarios. It is critical to consider the costs associated with scheduling decisions, as they can significantly impact system performance and overall efficiency.*

5 CHALLENGE IV: BALANCING SCHEDULING POLICIES

We observe that real-world serverless platforms typically deploy multiple serverless policies. For instance, K8s-based serverless platforms (such as Knative [13], OpenFaaS [19], Kubeless [14], and vHive [64]) come with 20 different policies

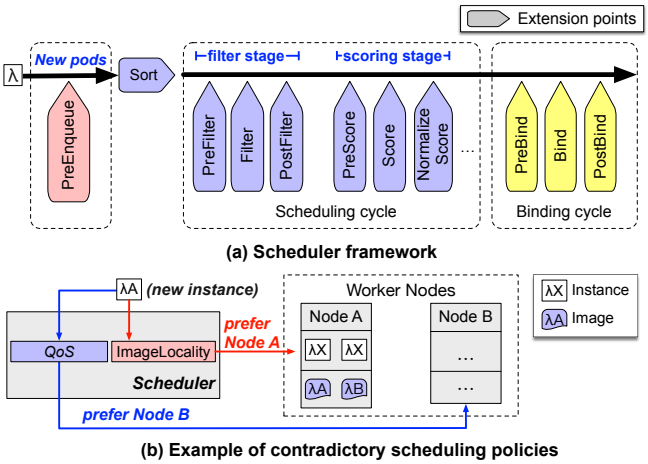


Figure 7: Scheduler policies. ***(a) The extensible scheduler used in K8s-based serverless systems. (b) An example of contradictory scheduling policies.***

(implemented as plugins) by default [16, 22]. This fact illustrates the significant challenge of *balancing different policies within a serverless system*, which is crucial for the effective management and optimal utilization of such systems.

Supports for multiple policies by serverless schedulers. As many serverless platforms directly reuse K8s scheduler, we take it as a representative example. The K8s scheduler provides native extensibility, with the scheduling process consisting of several stages that include extension points, as shown in Figure 7 (a). This flexibility allows developers to implement plugins that can support customized scheduling policies. The most critical stages of the scheduling process are the *filter* and *scoring* stages. During the filter stage, each enabled filter plugin filters out improper nodes, finally leaving nodes that satisfy all filter policies. The scheduler then proceeds to the scoring stage, where each score plugin assigns a score to each node. Ultimately, the scheduler selects the most suitable node based on the weighted average of the scores.

For example, one of the default score plugins, "ImageLocality", prioritizes nodes based on the readiness of their container images, effectively mitigating the costs of image pulling, which can take between 3~80 seconds [65, 72]. As cloud vendors often require customized scheduling policies to meet their specific needs, it is common practice for them to design their own customized schedulers. Therefore, we believe that an extensible serverless scheduler that can support great customizability is essential for both the research community and industry.

Gaps & challenges. While K8s scheduler plugins provide extensibility, making proper use of this feature can be challenging. One significant challenge is balancing the multiple plugins properly. Scheduler plugins [16] are diverse, representing various scheduling requirements. Each stage of the K8s scheduler's scheduling process can simultaneously apply *multiple* plugins, enabling the selection of nodes that result from comprehensive consideration of the scheduling requirements. However, there is currently no reasonable approach to balance the requirements.

As an example, consider the scoring step. The weights of multiple scoring plugins are uniform and fixed for various Pods. This approach can lead to suboptimal scheduling results when the results of different plugins contradict each other. Figure 7 (b) shows an example where the scheduler adopts two plugins during the scoring stage, "ImageLocality" and "QoS". The ImageLocality plugin prefers to deploy a new instance on Node-A because the required container images of the instance are ready there, while the QoS plugin prefers Node-B because it has fewer running instances and less interference, making it better for QoS guarantee. In this case, the two plugins have conflicting results, so the final choice between Node A and Node B requires balancing these two results properly based on the characteristics of the Pod. Based on our production experience, similar phenomena, i.e., different plugins causing different scheduling results, are quite common. However, there is currently a lack of appropriate systematic approaches to analyze and balance multiple policies, without which the final scheduling result may be suboptimal.

Current methods (e.g., scoring mechanisms) cannot properly balance the policies for at least the following reasons. Firstly and fundamentally, there is a lack of specific criteria to define and assess optimal scheduling. Hence, the scheduler lacks knowledge regarding "what is optimal", let alone the ways to achieve optimal results. Secondly, there is a deficiency in efficient weight configuration methods, and currently the weights are manually configured by engineers in production environments. Furthermore, the plugin weights are fixed and cannot be customized based on specific scenarios or Pod/Node configurations.

Although existing scheduler works [24, 30, 31, 34–36, 46, 50, 51, 54–57, 63, 67, 70, 73–75, 77, 78] can achieve great results in the specific problem they address, such as resource interference or QoS violation, they often ignore the existence of other policies, making the designs infeasible in real-world platforms. For example, Gsight [77] and Owl [63] use prediction methods to quantify the risk of QoS violation caused by resource interference between co-located instances on a node. They attempt to deploy more instances without violating QoS for better resource utilization. However, besides resource interference, QoS violation may be caused by other factors such as long image pulling latency, which can be addressed by the ImageLocality policy. Since these ignored factors may have already been considered by other existing

plugins or policies, balancing the cutting-edge scheduling policies with existing ones in a reasonable way can potentially improve the adaptation of scheduler policies to complex real-world cloud environments.

In conclusion, the intent of K8s scheduler's support for multiple plugins is to be able to take into account multiple scheduling requirements and thus make comprehensive, fully considered optimal scheduling decisions. However, due to the lack of proper balancing approaches, such a mechanism may instead lead to suboptimal results. It could have an increasingly significant impact on every Pod as the number of plugins increases.

Opportunities & suggestions. Addressing this challenge presents an opportunity for further research on how to balance these requirements effectively to enable an optimized scheduling process. Designing a balancing mechanism requires joint efforts of industry and academia, since the former is familiar with the requirements and workloads, while the latter is knowledgeable about specific methodologies. Industry opening their data as traces could be helpful for academia to do simulations, analysis and optimizations. The comprehensive definition of optimal scheduling is also preferably defined by industry and academia collaboratively. Several methods can be applied for reaching the optimal, e.g, using reinforcement learning to dynamically adjust the weights of various plugins.

Insight IV

> *Real-world serverless platforms will deploy multiple scheduling policies simultaneously. Designing effective mechanisms for balancing multiple scheduler policies is crucial for the management and utilization of serverless platforms.*

6 CHALLENGE V: COSTS OF SIDECAR

On the implementation of function instances, people often embark on a misunderstanding — one function instance means one container. However, modern cloud-native platforms tend to utilize a higher level of abstraction, such as K8s Pod, to represent serverless instances that may consist of one or multiple containers. For example, each Knative Pod includes a function container and a queue container. The sidecar is widely adopted by real-world serverless platforms, especially by public cloud serverless platforms such as our platform. A comparison between simplified serverless systems (no sidecars) and real-world serverless systems (with sidecars) is shown in Figure 8.

Benefits of Sidecar. Sidecar containers have become increasingly popular in serverless computing due to their ability to provide additional features, such as logging, monitoring, security, and proxying, for serverless functions. For instance, Knative's queue container, when used as a sidecar, can queue

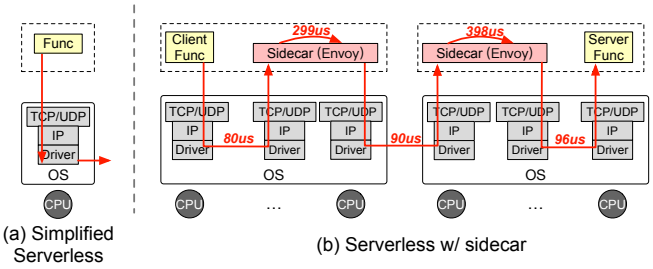
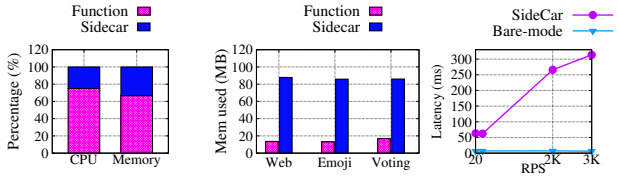


Figure 8: Comparison between serverless with and without sidecars. ***We label the latency for an emoji-vote application using Istio/Envoy as our sidecar. Client and server functions are running in the same machine in this case.***



(a) Allocated resources. **(b) Used resources.** **(c) Latency.**

Figure 9: Costs of sidecar. ***(a) About 25% CPU and 33% memory resources are allocated for sidecar containers. (b) Sidecar containers may use more memory resources when request per second (RPS) is low. (c) Sidecar brings higher end-to-end latency.***

all incoming requests and deliver them to the function container when it becomes available to handle new requests. This simplifies the implementation of function containers and enables independent management of both containers, avoiding any interference between them. Furthermore, the sidecar design allows developers to add or remove functionality as required, minimizing the chances of introducing bugs or other issues into the function's core code.

The separation of the function and sidecar containers also facilitates dynamic CPU resource allocation by the cloud platform based on the load of both containers. This means that the resources can be scaled independently, leading to better resource utilization overall. Additionally, the modular design of sidecar containers makes it easier to manage and update them independently, without having to modify the function's core code. This provides greater flexibility and agility in managing serverless systems, reducing the time and effort required to update and maintain the system.

Gaps & challenges. Despite the benefits of using a sidecar container in serverless computing environments, it can also cause non-trivial costs. Firstly, sidecar containers can occupy significant resources as shown in Figure 9 (a) and (b), which may affect function density. For example, our public serverless platform uses a worker as a sidecar to handle operations such as pulling source code. Depending on the size

of the instance, the sidecar may consume 0.3~1 vCPU and 300~800MB memory resources. We also conduct another experiment using Envoy as the sidecar, where a sidecar container can use up to 25% CPU and 33% memory resources. Moreover, the runtime costs associated with managing sidecar containers can also increase as compared to managing function instances, particularly when the RPS is low.

Another significant cost of using sidecar containers is longer communication latency, which arises due to the longer communication path illustrated in Figure 8 (b). In our experimental environment (Figure 9 (c)), we have observed that the end-to-end latency can increase by 9.5x (RPS=20) and up to 49.8x (RPS=3,000). These costs underscore the importance of carefully considering the deployment and management of sidecar containers and optimizing the resource allocation to reduce the overall costs of the system.

While sidecar containers can offer benefits, they also introduce non-trivial costs that researchers must evaluate and optimize. Firstly, many existing research works [25, 39, 42, 45, 59, 61] have optimized DAG communication, most of which utilize local bus or IPC to optimize communication within the same machine. However, few of them consider the sidecar containers. As shown in Figure 8 (b) (latency breakdown on the same machine), sidecars introduce non-trivial cost, which is contributed by more context switches, network protocol cost, the sidecar processing logic, and even the microarchitecture states. It also encounters a contradiction between performance and resource allocation, as sidecar resources are usually limited. Thus, it remains an open question of how to reduce communication latency when sidecar containers are present. Secondly, prior works have explored optimizing the cold start latency using fork or checkpoint-restore techniques [29, 32, 33, 52, 64, 66]. However, they typically assume a single container instance based on Linux containers or VMs and do not account for the complexities introduced by multiple container instances with sidecars. This is one reason why fork-based optimizations have not been widely adopted by the industry. Lastly, sidecar containers can lead to serious resource interference in addition to increased resource consumption. For example, cache and TLB may need to be frequently switched for network-intensive applications, which are common in serverless systems. This raises questions on how to design system software, such as OS kernels and hardware, including new architecture extensions, to effectively support serverless with sidecars.

In summary, as deploying the sidecar with each application instance becomes the norm, the resource and performance overheads it imposes will affect all scenarios.

Opportunities & suggestions. Addressing these challenges presents opportunities for further research on how to optimize and design serverless systems with sidecar containers more efficiently. Several ways may be helpful for optimizations. For example, new hardware can be utilized for offloading sidecar logic, e.g., smartNICs (DPUs). Moreover, since sidecars are usually applied by multiple Pods on a node, decoupling the processing logics and sharing some of them can be helpful for reducing resource consumption.

Furthermore, we do not recommend utilizing the wrapper of the function, which is compiled with the function code during deployment, to support all functionalities of the sidecar pattern, such as proxying and queuing. Besides worse modularity, there is another important reason why this approach may not be feasible. Modern serverless platforms often support deploying functions using container images, which means the platform cannot inject code into the binary of serverless functions. For example, AWS began supporting the packaging and deployment of Lambda functions as Docker container images in 2020, and about 20% of Lambda users have chosen this method to deploy functions as of 2022 [10]. This trend makes the use of the sidecar pattern more necessary, as it enables the deployment of platform-level functionalities without modifying the application code (which may be closed-source).

Insight V

> *The sidecar pattern, running one or more sidecar containers alongside an application container, is becoming a standard in serverless computing systems. However, few of the existing works or optimizations consider the impact of sidecar containers. As a result, the question of how to design efficient and lightweight sidecar containers for serverless remains an open challenge.*

7 CONCLUSION

This paper has outlined five open challenges to bridge the gap between existing research and real-world issues faced by the industry. We believe that the presented observations, challenges, opportunities, and insights can aid in addressing these discrepancies, (hopefully) creating momentum towards improved serverless platforms.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Yingjun Wu and the anonymous SoCC'23 reviewers for their insightful suggestions. This work was supported in part by National Key Research and Development Program of China (No. 2022YFB4502003), National Natural Science Foundation of China (No. 62302300, 62132014, 61925206), and Startup Fund for Young Faculty at SJTU (SFYF at SJTU). This work was also supported by Huawei Cloud Inc. Corresponding author: Dong Du (dd_nirvana@sjtu.edu.cn).

REFERENCES

[1] 2021. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Referenced 2021.

[2] 2021. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/. Referenced Jan. 2021.

[3] 2021. AWS LambdaEdge changes duration billing granularity from 50ms down to 1ms. https://aws.amazon.com/about-aws/whats-new/2021/03/cloudfront-lambda-at-edge-billing-granularity/. Referenced July 2021.

[4] 2021. Azure Functions Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/. Referenced Jan. 2021.

[5] 2021. Checkpoint/Restore in gVisor. https://gvisor.dev/docs/user_guide/checkpoint_restore/. Referenced April 2021.

[6] 2021. Cloud Functions - Overview | IBM. https://www.ibm.com/cloud/functions. Referenced Jan. 2021.

[7] 2021. documentation/Limitations.md at master - kata-containers/documentation. https://github.com/kata-containers/documentation/blob/master/Limitations.md. Referenced April 2021.

[8] 2021. Google Cloud Function. https://cloud.google.com/functions/. Referenced Jan. 2021.

[9] 2021. [Snaps] Full snapshot + restore, firecracker-microvm/firecracker. https://github.com/firecracker-microvm/firecracker/issues/1184. Referenced April 2021.

[10] 2022. The State of Serverless (2022 Version). https://www.datadoghq.com/state-of-serverless/. Referenced Jun. 2023.

[11] 2023. Crane-scheduler. https://github.com/gocrane/crane-scheduler.

[12] Fission 2023. Fission. Fission. https://fission.io/

[13] 2023. Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/docs/.

[14] 2023. Kubernetes Native Serverless Framework. https://github.com/vmware-archive/kubeless.

[15] 2023. Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[16] 2023. kubernetes-sigs/scheduler-plugins. https://github.com/kubernetes-sigs/scheduler-plugins.

[17] 2023. KWOK stands for Kubernetes WithOut Kubelet. https://kwok.sigs.k8s.io/.

[18] nuclio 2023. Nuclio. nuclio. https://nuclio.io/

[19] 2023. OpenFaaS. https://www.openfaas.com/. Referenced May 2023.

[20] 2023. Results of measuring of API performance of Kubernetes. https://docs.openstack.org/developer/performance-docs/test_results/container_cluster_systems/kubernetes/API_testing/index.html. Referenced May 2023.

[21] 2023. Scheduler Configuration, Kubernetes. https://kubernetes.io/.

[22] 2023. Scheduler Configuration, Kubernetes. https://kubernetes.io/docs/reference/scheduling/config/.

[23] VMware Archive 2023. Vmware-Archive/Kubeless. VMware Archive. https://github.com/vmware-archive/kubeless

[24] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In IEEE INFOCOM 2020 - IEEE Conference on Computer Communications. 129–138. https://doi.org/10.1109/INFOCOM41043.2020.9155363

[25] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. *SAND*: Towards High-Performance Serverless Computing. In 2018 *USENIX* Annual Technical Conference (*USENIX ATC* 18). 923–935.

[26] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In Proceedings of the Seventeenth European Conference on Computer Systems (Rennes, France) (EuroSys '22). Association for Computing Machinery, New York, NY, USA, 730–746. https://doi.org/10.1145/3492321.3524270

[27] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (Broomfield, CO) (OSDI'14). USENIX Association, USA, 285–300.

[28] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In 2023 USENIX Annual Technical Conference (USENIX ATC 23). USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/atc23/presentation/brooker

[29] 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 Fifteenth European Conference on Computer Systems (Heraklion, Greece) (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[30] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. 2019. Avalon: Towards QoS Awareness and Improved Utilization through Multi-Resource Management in Datacenters. In Proceedings of the ACM International Conference on Supercomputing (Phoenix, Arizona) (ICS '19). Association for Computing Machinery, New York, NY, USA, 272–283. https://doi.org/10.1145/3330345.3330370

[31] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Providence, RI, USA) (ASPLOS '19). Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[32] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless Computing on Heterogeneous Computers. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS 2022). Association for Computing Machinery, New York, NY, USA, 797–813. https://doi.org/10.1145/3503222.3507732

[33] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[34] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. USENIX Association, USA.

[35] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Virtual, USA) (ASPLOS '21). Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700

[36] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages

and Operating Systems (Providence, RI, USA) (ASPLOS '19). Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3297858.3304004

[37] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In Proceedings of the 21st International Middleware Conference (Delft, Netherlands) (Middleware '20). Association for Computing Machinery, New York, NY, USA, 280–295. https://doi.org/10.1145/3423211.3425683

[38] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[39] 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 Programming Languages and Operating Systems (Virtual, USA) (ASPLOS 2021). Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[40] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383 (2019).

[41] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: Principled and Practical Scheduling for Serverless Functions. In Proceedings of the 13th Symposium on Cloud Computing (San Francisco, California) (SoCC '22). Association for Computing Machinery, New York, NY, USA, 289–305. https://doi.org/10.1145/3542929.3563468

[42] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In 2021 USENIX Annual Technical Conference (USENIX ATC 21). USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[43] Wenzhuo Li, Chuang Lin, Puheng Zhang, and Mao Miao. 2017. Probe sharing: A simple technique to improve on sparrow. In 2017 IEEE Symposium on Computers and Communications (ISCC). 863–870. https://doi.org/10.1109/ISCC.2017.8024635

[44] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. FaaSLight: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing. ACM Trans. Softw. Eng. Methodol. (feb 2023). https://doi.org/10.1145/3585007 Just Accepted.

[45] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In 2021 USENIX Annual Technical Conference (USENIX ATC 21). USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[46] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating Interference in Cloud Services by Middleware Reconfiguration. In Proceedings of the 15th International Middleware Conference (Bordeaux, France) (Middleware '14). Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/2663165.2663330

[47] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (Porto Alegre, Brazil) (MICRO-44). Association for Computing Machinery, New York, NY, USA, 248–259. https://doi.org/10.1145/2155620.2155650

[48] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. 2017. ESP: A Machine Learning Approach to Predicting Application Interference. In 2017 IEEE International Conference on Autonomic Computing (ICAC). 125–134. https://doi.org/10.1109/ICAC.2017.29

[49] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. IEEE Transactions on Parallel and Distributed Systems 12, 10 (2001), 1094–1104. https://doi.org/10.1109/71.963420

[50] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. 2020. Auto-Scaling Cloud-Based Memory-Intensive Applications. In 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). 229–237. https://doi.org/10.1109/CLOUD49709.2020.00042

[51] Joe H. Novak, Sneha Kumar Kasera, and Ryan Stutsman. 2019. Cloud Functions for Fast and Robust Resource Auto-Scaling. In 2019 11th International Conference on Communication Systems & Networks (COMSNETS). 133–140. https://doi.org/10.1109/COMSNETS.2019.8711058

[52] 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 2018 *USENIX* Annual Technical Conference (*USENIX ATC* 18). 57–70.

[53] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (Farminton, Pennsylvania) (SOSP '13). Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[54] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[55] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. USENIX Association, USA.

[56] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In Proceedings of the Fifteenth European Conference on Computer Systems (Heraklion, Greece) (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[57] Aakanksha Saha and Sonika Jindal. 2018. EMARS: Efficient Management and Allocation of Resources in Serverless. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). 827–830. https://doi.org/10.1109/CLOUD.2018.00113

[58] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In Proceedings of the 8th ACM European Conference on Computer Systems (Prague, Czech Republic) (EuroSys '13). Association for Computing Machinery, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[59] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In 2020 *USENIX* Annual Technical Conference (*USENIX ATC* 20). 419–433.

[60] 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 (Seattle, WA, USA) (SoCC '21). Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981

[61] 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. Proc. VLDB Endow. 13, 12 (jul 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[62] 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 (Virtual Event, USA) (SoCC '20). Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/3419111.3421306

[63] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud. In Proceedings of the 13th Symposium on Cloud Computing (San Francisco, California) (SoCC '22). Association for Computing Machinery, New York, NY, USA, 78–93. https://doi.org/10.1145/3542929.3563470

[64] 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 (Virtual, USA) (ASPLOS 2021). Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[65] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[66] 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 Fourteenth EuroSys Conference 2019 (Dresden, Germany) (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 39, 16 pages. https://doi.org/10.1145/3302424.3303978

[67] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. 2022. DeepScaling: Microservices Autoscaling for Stable CPU Utilization in Large Scale Cloud Systems. In Proceedings of the 13th Symposium on Cloud Computing (San Francisco, California) (SoCC '22). Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3542929.3563469

[68] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[69] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. 2018. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-Located Workloads. In Proceedings of the 19th International Middleware Conference (Rennes, France) (Middleware '18). Association for Computing Machinery, New York, NY, USA, 146–160. https://doi.org/10.1145/3274808.3274820

[70] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. SIGARCH Comput. Archit. News 41, 3 (jun 2013), 607–618. https://doi.org/10.1145/2508148.2485974

[71] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[72] 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 ACM Symposium on Cloud Computing (SoCC '20). Association for Computing Machinery. https://doi.org/10.1145/3419111.3421280

[73] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In Proceedings of the 8th ACM European Conference on Computer Systems (Prague, Czech Republic) (EuroSys '13). Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[74] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Virtual, USA) (ASPLOS 2021). Association for Computing Machinery, New York, NY, USA, 167–181. https://doi.org/10.1145/3445814.3446693

[75] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (Cambridge, United Kingdom) (MICRO-47). IEEE Computer Society, USA, 406–418. https://doi.org/10.1109/MICRO.2014.53

[76] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In Proceedings of the Sixteenth European Conference on Computer Systems (Online Event, United Kingdom) (EuroSys '21). Association for Computing Machinery, New York, NY, USA, 540–555. https://doi.org/10.1145/3447786.3456258

[77] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (St. Louis, Missouri) (SC '21). Association for Computing Machinery, New York, NY, USA, Article 22, 15 pages. https://doi.org/10.1145/3458817.3476215

[78] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-Stage Serverless Workflows. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3567955.3567960