Is It Time To Put Cold Starts In The Deep Freeze?

Carlos Segarra
cs1620@ic.ac.uk
Imperial College London

Ivan Durev
i.durev23@imperial.ac.uk
Imperial College London

Peter Pietzuch
prp@imperial.ac.uk
Imperial College London

ABSTRACT

Cold-start times have been the "end-all, be-all" metric for research in serverless cloud computing over the past decade. Reducing the impact of cold starts matters, because they can be the biggest contributor to a serverless function's end-to-end execution time. Recent studies from cloud providers, however, indicate that, in practice, a majority of serverless functions are triggered by non-interactive workloads. To substantiate this, we study the types of serverless functions used in 35 publications and find that over 80% of functions are not semantically latency sensitive. If a function is non-interactive and latency insensitive, is end-to-end execution time the right metric to optimize in serverless? What if cold starts do not matter that much, after all?

In this vision paper, we explore what serverless environments in which cold starts do not matter would look like. We make the case that serverless research should focus on supporting latency insensitive, i.e., *batch*, workloads. Based on this, we explore the design space for *DFaaS*, a serverless framework with an execution model in which functions can be arbitrarily delayed. DFaaS users annotate each function with a delay tolerance and, as long as the deadline has not passed, the runtime may interrupt or migrate function execution. Our micro-benchmarks suggest that, by targeting batch workloads, DFaaS can improve substantially the resource usage of serverless clouds and lower costs for users.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*.

KEYWORDS

cloud computing, serverless, function-as-a-service, cold-starts, batch processing, serverless accelerators

ACM ISBN 979-8-4007-1286-9/24/11…$15.00
https://doi.org/10.1145/3698038.3698527

ACM Reference Format:
Carlos Segarra, Ivan Durev, and Peter Pietzuch. 2024. Is It Time To Put Cold Starts In The Deep Freeze?. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3698038.3698527

1 INTRODUCTION

Prior research on serverless cloud computing has focused on reducing cold-start latencies [1, 4, 12, 18, 36, 38, 47, 51, 53, 59, 60, 68]. The cold-start latency refers to the time that it takes the serverless provider to provision an execution environment for a serverless function. Production traces from serverless environments in clouds show that cold start times can dominate the end-to-end execution time of serverless functions, measured as the time elapsed between function invocation and response [26, 40, 57]. By optimizing cold starts, prior work is, in fact, optimizing end-to-end execution time.

The same production traces, however, also reveal another trait of serverless functions in practice: most function invocations are triggered by non-interactive events from storage, queueing, or orchestration systems. For example, Azure reports that over 64% of function invocations are triggered by non-interactive events [57]; at Meta, this percentage is 85% [52]. If functions are triggered by non-interactive events, is end-to-end execution time the right metric to reduce?

Based on the hypothesis that serverless functions are used in non-interactive workloads, we classify the serverless functions in 35 research articles depending on their use case, and find that 80% of functions do not have a latency focus. We call these functions *latency insensitive*. For example, a function to compress an image [15, 28, 31, 68] for long-term archival storage does not have a strict latency requirement. If functions are latency insensitive, why reduce end-to-end execution time?

Based on an empirical analysis (§2), we conclude that, contrary to established wisdom, much serverless usage today is for non-interactive latency insensitive workloads, which do not benefit from cold-start optimizations. We do not know if this is due to the demands of real-world workloads, or cold-start optimizations in serverless infrastructure lagging behind research [16, 40].

Instead, we make the case that batch workloads are a genuinely good fit for a serverless model: in traditional cloud computing, the user controls the application (*what*), the server (*where*), and the request (*when*); in current serverless models, the user controls the *what* and the *when*, and the cloud provider controls the *where*. This separation of concerns leads to the ease-of-use of serverless and its cost savings. When serverless is applied to batch workloads, the user still controls the *what*, but the cloud provider can now control both the *when* and the *where*. This further separation of concerns is an important next step to fully realize the benefits of a serverless model [25].

In addition, many batch workloads, such as machine learning (ML), require hardware acceleration [27, 32, 58]. Accelerators are an expensive resource and hard to provision and configure [17]. We believe that – in the way how serverless achieved CPU and memory transparency for functions – serverless for batch workloads can achieve accelerator transparency. The corresponding improvements in cost and ease-of-use would make serverless an attractive execution model for accelerated batch workloads.

Since workloads such as ML inference must batch requests for cost efficiency, can serverless for batch workloads make them more cost effective? Less expensive, pay-per-use ML inference [20] would have a major impact on the popularity of serverless offerings, and prompts us to ask the question motivating this paper: *How can we better support latency-tolerant batch workloads through a serverless model?*

Our attempt at answering this question is the exploration of the design of *DFaaS*, a new serverless framework in which function execution can be *delayed* (§3.1). DFaaS builds on previous work that explores optimizations for serverless resource usage [62], including delay-tolerant (or time-shifted) computation [52, 54], access to accelerators [17, 39], and function multiplexing [62]. DFaaS bridges the gap between function-as-a-service (FaaS) and batch: it annotates each function with a latency deadline and, as long as the deadline has not passed, the function may be interrupted and migrated arbitrarily. Unlike best-effort computing [67, 71], DFaaS does not tolerate deadline violations and supports latency-sensitive functions, i.e., ones that have no slack in their deadlines. To optimize resource usage, function execution may be: (i) delayed to use warm resources (§3.2); (ii) fused with other functions to pipeline compute and I/O blocks (§3.3); and (iii) routed to dedicated per-language/accelerator queues to increase memory density and improve performance (§3.4).

DFaaS naturally applies to serverless workflows as long as chaining dependencies are preserved. In a serverless workflow, we consider the situation in which some functions are latency sensitive and others are not. For example, in an image processing workflow [15, 18, 31, 63, 65, 68, 69, 73], the first function may have to return a visual acknowledgment to the user, and thus is latency sensitive, but all subsequent functions can do the actual processing in the background, and thus become latency insensitive.

Our benchmarks show that DFaaS has potential to improve serverless infrastructure utilization: by delaying function execution, it reduces peak resource demand by up to 72% based on available production traces [26, 57]. By carefully fusing functions from the same workflow, it can improve CPU usage by up to 2×.

2 WHY COLD STARTS DO NOT MATTER

Next, we provide empirical evidence for the two claims as to why cold starts do not matter (that much): (C1) many serverless functions are triggered by non-interactive events; and (C2) many serverless functions are latency insensitive. We also claim that, as a consequence of cold-start optimizations, (C3) serverless infrastructure resource usage is low. We back these claims with empirical evidence from workload traces and reports (§2.1), academic surveys (§2.2), and a study of serverless research papers (§2.3).

2.1 Industry traces and reports

Meta, with its private serverless offering XFaaS [52], was the first major company to acknowledge most of our claims, and this work is inspired by their findings. Sahrei et al. claim that functions triggered non-interactively (i.e. not via direct HTTP requests) tend to be more delay tolerant. They report that at least 85% of their function invocations fall under this category (C1). They also claim that: *"[...] at Meta, serverless functions are rarely used to handle user-facing requests requiring sub-second response times"* [52]. This means that serverless functions at Meta are, by construction, latency insensitive (C2). We find this quote insightful, as it challenges the long-standing effort to reduce cold starts. The authors also report, after implementing a full set of infrastructure-wide optimizations, an average CPU utilization on their worker nodes of 66%, and claim that this could be several times higher than other industrial FaaS platforms (C3). Although they cannot back up this claim, they suggest focusing on high resource usage and function throughput in public clouds.

Alibaba presented traces from their public FaaS offering, Cloud Function Compute, as part of an effort to accelerate FaaS container image provisioning [66]. These traces include three representative FaaS applications: gaming, IoT, and video processing. Wang et al. do not clarify whether these applications are latency sensitive or not, but report that 57% of image pulls for their Beijing region see latencies longer than 45 seconds, and 86% of image pulls in their Shanghai region see latencies longer than 80 seconds. These latencies motivated Alibaba to optimize image pull times. We observe that, if the original FaaS applications experienced these latencies, they were most likely already used in a latency-insensitive context (C2). In addition, they report a peak-to-trough load ratio of more than 500×. This means that, in their effort to mitigate cold starts and not allowing functions to be delayed, they suffer from high resource over-provisioning and sandbox churn, known to introduce severe resource underutilization [16] (C3).

Huawei presented traces from both their public and private FaaS offerings [26]. From these traces, we cannot infer if functions are triggered by interactive or non-interactive events. However, they study the periodicity of function invocations and conclude that: *"A significant number of functions in both platforms have strong periodicity [...]. Periodicity is especially significant for more highly requested functions"* [26]. We claim that if a function is highly periodic, it is likely to be triggered by non-interactive events such as cron jobs (C1), and probably latency insensitive (C2). Even if a periodic function was latency sensitive, its periodicity and predictability make it likely to be scheduled in advance, effectively hiding its latency requirements. In terms of resource usage, they also report a high peak-to-trough ratio (which we also plot in Fig. 2b). The authors highlight that another big source of resource underutilization is that functions over-provision resources, with 60% of allocations using less than an order of magnitude of their requested CPU resources (C3).

Azure was the first cloud provider to characterize a public FaaS offering, Azure Functions, and make traces available to the community [57]. While this motivated much serverless research, four years later, we look back at it under a different light. Shahrad et al. report that only 36% of function invocations come from HTTP requests. This means that 64% of function invocations are triggered by non-interactive events (C1). From the original traces, we cannot infer whether the functions are latency sensitive. However, if we consider the follow-on work on Durable Functions, Azure's offering for serverless workflows, we see that all examples cited, implemented and evaluated are latency insensitive [11]. Albeit potentially anecdotal, this makes us believe that a relevant portion of the functions fall under this category (C2). In terms of resource usage, we see Huawei's claims confirmed: high peak-to-trough load ratios and low intra-function resource usage, which potentially result in infrastructure-wide resource underutilization (C3).

Other self-reported usage of serverless at scale without production traces or peer review also exist. A study on the AWS marketplace shows that, semantically, only 1% of functions deployed from the serverless marketplace are latency sensitive [61] (C2). These correspond largely to microservice HTTP endpoint functions. Similarly, a series of articles on Netflix's video processing pipeline shows that they use *"microservices that orchestrate serverless functions"* [46] (C1) to balance latency sensitive tasks (handled by always-on microservices) with latency insensitive tasks (handled by cheap on-demand serverless) with latency deadlines in the order of days [45, 46] (C2).

2.2 Research surveys

Several surveys have studied the use of serverless in practice. Eismann et al. [19] report that, across all their studied scenarios, 45% of function invocations are non-interactive (C1). The authors also analyze the relevance of latency for serverless functions and claim that, for 36% of functions, latency is irrelevant. Their distribution of application types suggests that only 20% of functions are latency sensitive (C2). These results coincide with our field study (Tab. 1). Hassan et al. [21] also analyze the types of serverless applications used in existing literature and reach similar conclusions: most serverless applications do background tasks such as file processing or information retrieval and are pervasive in edge or IoT scenarios (C2). Mampage et al. [44] confirm this.

2.3 Field study

To substantiate our claims, we analyze the serverless functions used in the research literature. For each function or workflow used in a research paper, we decide if it is semantically latency sensitive or not. We classify a function as semantically latency *insensitive* if the task performed *can* tolerate delays. File encryption, compression, or batch analytics are, for example, considered to tolerate delays, whereas functions serving API requests in a web server are not. For serverless workflows, we classify workflows where only the first function is latency sensitive as latency *insensitive.* This is a common situation in practice, and could be treated as a special case in an otherwise latency insensitive (i.e., delayable) architecture. Most microservice-like workflows, where the first function returns an acknowledgment and the rest perform a task in the background, fall under this category.

Tab. 1 shows our results: most serverless functions perform background, batch-processing, or periodic jobs that can tolerate delays and are thus *latency insensitive.* In particular, from 35 papers, 80% of functions and 71% of workflows are latency insensitive (C2).

2.4 Summary

In this section, we have presented supporting evidence for two claims about a relevant portion of serverless functions in practice: (C1) functions are triggered by non-interactive events and (C2) functions are latency insensitive. These functions do not, as a consequence, benefit from cold-start optimizations which, we claim, (C3) are negatively affecting serverless infrastructure resource utilization.

It is not possible for us to know if serverless functions are latency insensitive because existing infrastructure does not support fast-enough instantiation times [16, 40], or it is

Name	Used In	Latency Sensitive
	Functions	
Encryption	FunctionBench [28], Medes [53], vHive [65], Desiccant [73], Ignite [14], Pronghorn [31], Ensure [63]	Yes
Complex Math	Cirrus [13], DataFlower [37], Desiccant [73], Ensure [63], FaaSdom [43], Medes [53], MuBenchmark [8], Pagurus [36], Netherite [11]	No
E-mail Notification	Ensure [63], Fusionize [56], ProFaaStinate [54]	No
File Compression	FunctionBench [28], Pronghorn [31], RainbowCake [68], SeBS [15]	No
File Upload	Pronghorn [31], RainbowCake [68], SeBS [15], ProFaaStinate [54]	No
Filesystem Access	Desiccant [73], FaaSdom [43], SEUSS [12]	No
Generic Test	Boki [23], Catalyzer [18], FUYAO [39], Netherite [11], SEUSS [12], SOCK [47]	No
Graph Algorithms	Desiccant [73], Ensure [63], Pronghorn [31], RainbowCake [68], SeBS [15]	No
Log Serving	Golgi [35], Owl [64]	Yes
ML Inf/Train	Faasm [59] (×2)	No
Matmul	Desiccant [73], Ensure [63], FaaSdom [43], MuBenchmark [8], Pronghorn [31]	No
OCR	ProFaaStinate [54], RainbowCake [68]	No
PageRank	Pronghorn [31], SeBS [15]	Yes
Sorting / Scanning	Desiccant [73], Ensure [63], Pocket [29], ProFaaStinate [54], MuBenchmark [8], SEUSS [12]	No
Web Server	Catalyzer [18], Desiccant [73] (×2), FaaSdom [43], SEUSS [12]	Yes
	Percentage non latency-sensitive:	80%
	Workflows	
Bank / ChatBot	Netherite [11], Orion [42], Pagurus [36]	No
Data Analysis	Desiccant [73], Ensure [63], Faastlane [33], FINRA [2], Fusionize [56], FUYAO [39], Golgi [35], Owl [64], Pagurus [36], ProFaaStinate [54], RainbowCake [68], RMMap [41], ServerlessBench [69]	No
DNA Processing	RainbowCake [68], SeBS [15]	No
HTML Generation	Desiccant [73], Faa$T [50], FunctionBench [28], Ignite [14], Medes [53], Pronghorn [31], RainbowCake [68], SeBS [15], vHive [65]	Yes
Image Processing	DataFlower [37], Catalyzer [18], Desiccant [73], Ensure [63], Faastlane [33] (×2), Golgi [35], Medes [53], MXFaaS [62], Owl [64] (×2), Pronghorn [31], RainbowCake [68] (×2), SeBS [15] (×2), ServerlessBench [69], SOCK [47], vHive [65] (×2)	No
JSON Operations	FunctionBench [28], vHive [65]	Yes
Media Streaming	Beldi [70], Boki [23], FUYAO [39], Sprocket [3]	Yes*
ML Inference	Cirrus [13], Ensure [63], Faa$T [50], Faastlane [33], Medes [53], MXFaaS [62], Pagurus [36] , RainbowCake [68], RMMap [41], vHive [65]	No
ML Training	Cirrus [13], Medes [53], MXFaaS [62], Orion [42], RainbowCake [68], RMMap [41], vHive [65]	No
Online Compilation	Faa$T [50], Pocket [29], RainbowCake [68], ServerlessBench [69]	No
Device Control	Desiccant [73], Golgi [35], Owl [64], ServerlessBench [69]	Yes
Social Network	Beldi [70], Boki [23], Catalyzer [18], Pagurus [36]	Yes*
Store / Search	Catalyzer [18], Desiccant [73] (×2), Fusionize [56], MXFaaS [62], Pagurus [36]	Yes*
Travel Reservation	Beldi [70], Boki [23], MXFaaS [62]	Yes*
A/V Processing	DataFlower [37], Medes [53], MXFaaS [62], Owl [64], Pronghorn [31], RainbowCake [68], SeBS [15], Sprocket [3] (×2), vHive [65], Pocket [29], Orion [42]	No
WordCount	DataFlower [37], Desiccant [73], FunctionBench [28], Ignite [14], Medes [53], Netherite [11], Pronghorn [31], RMMap [41]	No
	Percentage non latency-sensitive:	71 %

Table 1: Field study of the type of serverless functions used for serverless research (For workflows, Yes* means that most functions in the workflow are latency sensitive.)

the genuine demand for serverless. We make the point that, in any case, non-interactive latency insensitive (i.e., *batch*) workloads are indeed a good fit for the serverless execution model. We believe that an undue emphasis has been placed on the cold start problem, which has happened to the detriment of supporting batch workloads. We explore next how, by ignoring cold starts, we can offer a serverless environment that is cheaper and more resource efficient.

3 DELAYABLE FAAS

In this section, we describe DFaaS, our vision for a serverless environment in which functions can be delayed arbitrarily (§3.1). DFaaS aims for high resource usage and function throughput by using three techniques: function delay (§3.2), function fusion (§3.3), and semantic scheduling (§3.4).

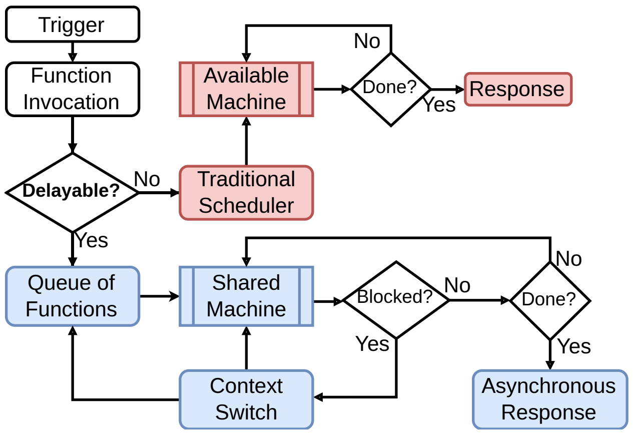


Figure 1: Overview of a DFaaS deployment (In a DFaaS deployment, a runtime for non-delayable, i.e., traditional, functions co-exists with a runtime for delayable ones.)

3.1 Overview

DFaaS bridges the gap between a serverless execution model and batch workloads by allowing functions to be arbitrarily *delayed*. Serverless functions in DFaaS are annotated with a *delay tolerance*. In a DFaaS deployment (Fig. 1), the delayable serverless backend co-exists with a traditional backend to provide backwards-compatibility with latency sensitive functions. A delay tolerance of 0 indicates that a function is latency sensitive and will be routed to the traditional backend; otherwise, the function will be routed to the delayable backend. A delay tolerance of -1 means that the function can be arbitrarily delayed as long as it eventually runs.

When a latency sensitive function is invoked, the runtime will execute it immediately, hopefully on a pre-initialized (warm) execution environment. Conversely, when a latency insensitive function is invoked, it may wait in a queue of functions. Latency insensitive functions can be preempted during execution and returned to the queue and, to optimize resource usage, may execute on shared machines. Latency insensitive functions are guaranteed the same isolation as latency sensitive ones.

Adding delay tolerance to a serverless function's definition is a simple, yet powerful, abstraction [52, 54]. Next, we introduce the techniques for high-level function orchestration enabled by this new abstraction, and we validate their potential benefits. In §4, we discuss further techniques and their implementation challenges.

3.2 Function delay

We first consider function delay in DFaaS, which involves postponing function execution to a later time point. It has been used in related work to homogenize function invocation patterns [52, 54]. Meta's internal FaaS offering, for example, exhibits strong diurnal periodicity, with many analytics jobs being triggered at midnight. These batch jobs can instead be delayed to other times of the day, when demand for computing resources is lower [52].

Fig. 2 shows the request arrival pattern from the Azure trace over 14 days [57], and the Huawei trace over 26 days [26]. Both traces show strong diurnal periodicity and high churn. In the same figure, we show a horizontal line, indicating the number of sandboxes required to serve the same number of function invocations with a uniform arrival rate. Flattening the arrival rate requires perfect knowledge of the function invocation distribution, but it gives an upper bound on how much it is possible to reduce sandbox creation churn. In the case of Azure, the difference between peak and flattened requests is 20%; in the case of Huawei, it is 72%. Reducing sandbox creation churn would mitigate orchestration overheads, known to be large at scale [16].

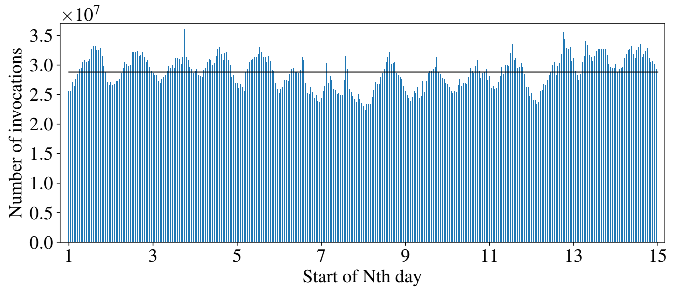
3.3 Function fusion

The second technique that we consider is function fusion, which is a form of resource over-commitment [35, 64]. The observation behind function fusion is that serverless functions spend much time blocked on I/O requests, waiting for data from external storage [22, 24, 33, 41]. By fusing functions together, when one function blocks for I/O, DFaaS can switch context to another function, increasing throughput and CPU usage. Such a fusion approach allows DFaaS to embrace a true *pay-per-use* billing model in which serverless users are only charged for used CPU time and not for blocked I/O time. Our experiments with Azure Functions confirm that users are, indeed, charged for CPU time when functions are blocked on I/O, even when using the recommended asynchronous API [7].

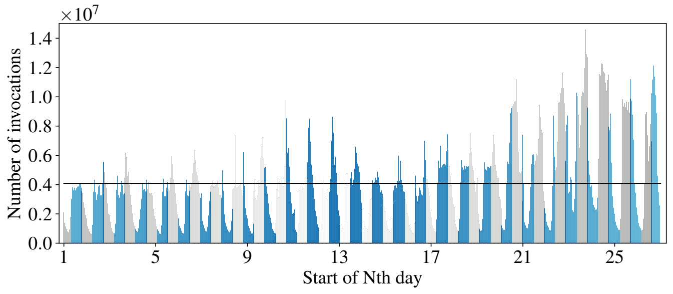
In addition, function fusion addresses the long-standing double-billing problem in serverless, which makes nested chaining a design anti-pattern [5, 9, 55]. Today's serverless platforms charge users for the time functions wait for nested calls [6],[1] whereas DFaaS avoids this due to function fusion.

There are two reasons why function fusion has seen limited adoption: (1) if optimizing for end-to-end execution time, it is challenging for a serverless scheduler to know when the overheads of context-switching do not outweigh the benefits of increased throughput. In DFaaS, given that we do not optimize for end-to-end execution time, functions may wait after performing I/O until the scheduler decides to context switch again; and (2) function fusion assumes that there is always another function to be context-switched to. In general, this assumption does not hold in traditional serverless environments, but it holds in DFaaS, as functions are delayed and batched together.

[1] In Azure Functions, orchestrator functions are the only exception to this.

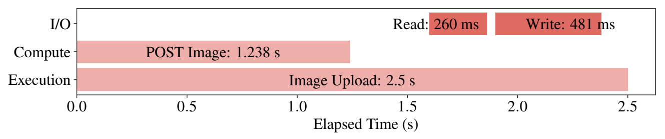


(a) Azure [57]

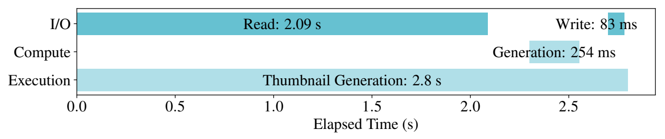


(b) Huawei [26]

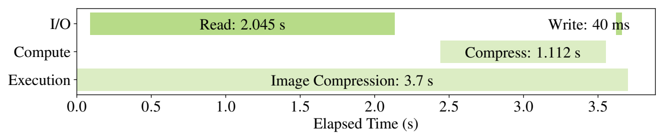
Figure 2: Hourly function invocation rates from industry traces (We present consecutive days to show the diurnal periodicity as well as the peak-to-average difference. The black line shows the number of invocations served per hour with the arrival rate flattened.)



(a) Image upload

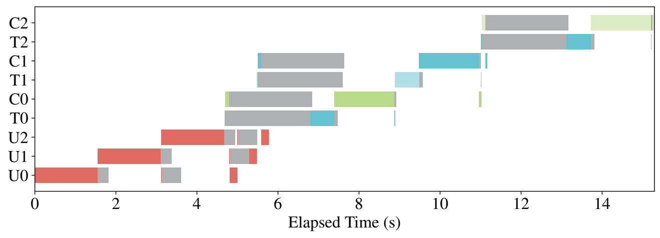


(b) Thumbnail generation

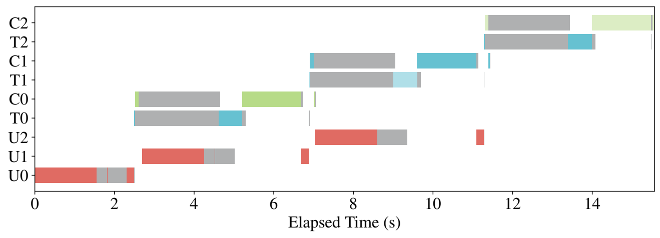


(c) Compression

Figure 3: Image processing workflow (The image upload function chains the thumbnail generation and compression.)



(a) Image upload is latency *sensitive*



(b) Image upload is latency *insensitive*

Figure 4: Fusion of three workflows (In gray, we show time blocked on I/O. We focus on single CPU core execution.)

To show the benefits of function fusion, we implement an image processing workflow [15, 18, 31, 63, 65, 68, 69, 73] in Azure Functions. Fig. 3 shows the execution time of each function in the workflow, differentiating between processing time and I/O time. The gaps in-between correspond to I/O set-up time. The upload function (Fig. 3a) is the first function in the workflow, and it chains to the other two in parallel (Fig. 3b and Fig. 3c). This example workflow spends 50% of its execution time blocked on I/O.

In Fig. 4, we present two possible execution plans for the workflow after applying function fusion, depending on whether the first function in the workflow is latency sensitive (Fig. 4a) or not (Fig. 4b). By interleaving different steps of the workflow and assuming perfect context switches, we can reduce the time blocked on I/O to 15% and 12%, respectively, concentrating execution on just 1 core CPU instead of 3.

We prototype function fusion in OpenWhisk [48]. We modify the scheduler, storage server, and Node.js runtime to support interrupting functions when they make a request to the storage server. Our prototype does not yet support function chaining, nor tracing, but, when running a batch of functions interleaving compute and I/O, we observe a similar efficiency increase from 54% to 89%.

3.4 Semantic scheduling

Finally, semantic scheduling is a catch-all term for scheduling functions to resources according to some high-level property of the function such as its programming language, JIT/GC state, or accelerator (e.g., FPGA, GPU, DPU, or ASIC)

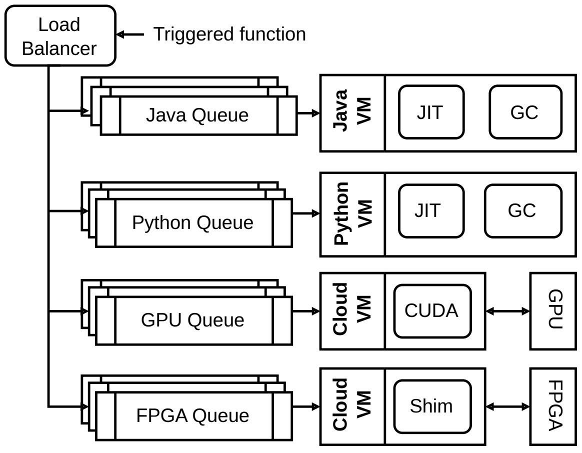


Figure 5: Overview of function queue design (Functions are routed to different queues depending on the required language runtime or accelerator.)

requirements. Recent work has shown the importance of maintaining JIT [31, 52] and GC [73] state in serverless execution, as well as the challenges of transparently supporting heterogeneous devices [17]. DFaaS builds on top of these ideas with the additional benefit of being able to delay function execution to prepare the execution environment.

In Fig. 5, we show a simplified architecture for a queue of functions from Fig. 1. In this example, DFaaS has per language-runtime and accelerator queues, and long-running VMs that maintain the JIT/GC state, as well as the accelerator runtime state. The VMs can be switched on only when enough functions wait in the queues, amortizing the cost of longer restores of the stateful JIT and GC components, or the cost of creating a cloud FPGA [32]. We expect such a design to yield benefits in terms of resource usage by de-duplicating memory and storage contents more effectively, as well as in terms of performance by sharing profiling information between co-located language runtimes and accelerators. We defer security considerations about multi-tenant access to accelerators [17, 72] or sharing JIT/GC state to future work.

4 DISCUSSION

In this section, we discuss open challenges for realizing our vision (§4.1) and further optimization opportunities (§4.2).

4.1 Challenges to our vision

In our presentation of DFaaS, we only described high level architecture considerations, together with potential orchestration mechanisms and their benefits. We anticipate the following challenges when implementing a complete DFaaS environment:

Programming model. DFaaS inherits the programming model from FaaS. Such a simplified model, however, cannot extend beyond CPU computation. Molecule [17] explored how to expose device heterogeneity to programmers, and we plan to build on that. Tied to the programming model is the execution abstraction that supports multi-tenant heterogeneous execution. How to achieve this, how to let users specify their workload requirements, and how to bill them accordingly remain open challenges.

Varying delay tolerances. Our function orchestration techniques, and particularly function delay (§3.2), rely on functions being (infinitely) delayable. In practice, we expect functions to have a finite delay tolerance or an opportunistic quota similar to XFaaS [52]. Implementing our function orchestration techniques subject to different delay tolerances requires a careful design of the scheduling, load balancing, and queuing components, and remains an open challenge.

Co-existence with traditional serverless runtime. In our DFaaS reference architecture (§3.1), we co-locate a delayable serverless runtime with a traditional, non-delayable serverless runtime. Having such an architecture could potentially negate DFaaS's benefits in term of resource usage. Therefore, we imagine that, in practice, the delayable and non-delayable infrastructure must become more integrated. Which components to share and how to make them cooperate reamin open challenges.

4.2 Further opportunities

Cold-starts have influenced the design of many components in the serverless stack from the cluster manager [16] to the programming model [34], including storage [29, 50], communication [41], execution environments [59], and image provisioning [10]. Ignoring cold-starts thus challenges the previous design choices and opens up the design space for a class of serverless frameworks. We will explore optimizations in adjacent areas such as ML inference [20] or RPC scheduling [30, 49] to leverage their findings in DFaaS.

5 CONCLUSIONS

We have presented DFaaS, our vision for a serverless environment in which cold starts do not matter. Based on an analysis of cloud workload traces, academic surveys, and research publications, we observe that a lot of serverless usage today is, contrary to popular wisdom, for non-interactive, latency insensitive, batch workloads. We make the case that batch workloads are a good fit for serverless and conclude that, instead of focusing on cold-starts, serverless cloud researchers should focus on supporting batch workloads.

Using these insights, we present DFaaS, a first step towards a FaaS platform for batch workloads. DFaaS bridges the gap between FaaS and batch by allowing functions to be delayed arbitrarily. We describe a DFaaS reference architecture, as well as function orchestration techniques, and describe the possible challenges when implementing them.

We believe that focusing on batch workloads such as ML inference and leaving cold-starts behind opens a novel and exciting research direction in serverless, and we hope that researchers in the field will join us in exploring the opportunities ahead.

Acknowledgments. This work has been partially funded by the European Union through the Horizon Europe projects CloudStars (101086248) and CloudSkin (101092646). We thank our shepherd, Prashant Shenoy, the anonymous reviewers, and Marios Kogias for their helpful feedback.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*.

[2] Amazon. 2014. FINRA Case Study. https://aws.amazon.com/solutions/case-studies/finra/.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *ACM Symposium on Cloud Computing (SoCC '18)*.

[4] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs. In *17th European Conference on Computer Systems (EuroSys '22)*.

[5] AWS. 2014. Functions Calling Functions. https://docs.aws.amazon.com/lambda/latest/operatorguide/functions-calling-functions.html.

[6] Azure. 2014. Azure Functions Billing. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-billing.

[7] Azure. 2014. Azure Functions Performance Reliability - Use Async Code. https://learn.microsoft.com/en-us/azure/azure-functions/performance-reliability.

[8] Timon Back and Vasilios Andrikopoulos. 2018. Using a Microbenchmark to Compare Function as a Service Solutions. In *7th IFIP WG 2.14 European Conference (ESOCC '18)*.

[9] 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 *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '17)*.

[10] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *USENIX Annual Technical Conference (ATC '23)*.

[11] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: efficient execution of serverless workflows. *Proc. VLDB Endow.* (2022).

[12] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *15th European Conference on Computer Systems (EuroSys '20)*.

[13] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *ACM Symposium on Cloud Computing (SoCC '19)*.

[14] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: leveraging runtimes for the serverless era. In *Workshop on Hot Topics in Operating Systems (HotOS '21)*.

[15] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: a serverless benchmark suite for function-as-a-service computing. In *22nd International Middleware Conference (Middleware '21)*.

[16] Lazar Cvetković, Rodrigo Fonseca, and Ana Klimovic. 2023. Understanding the Neglected Cost of Serverless Cluster Management. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23)*.

[17] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[18] 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 *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[19] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. 2020. A Review of Serverless Use Cases and their Characteristics. https://arxiv.org/abs/2008.11110.

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*.

[21] Hassan Hassan, Saman Barakat, and Qusay Sarhan. 2021. Survey on serverless computing. *Journal of Cloud Computing* (2021).

[22] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. https://arxiv.org/abs/1812.03651. (2018).

[23] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *28th Symposium on Operating Systems Principles (SOSP '21)*.

[24] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *ACM Symposium on Cloud Computing (SoCC '17)*.

[25] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. https://arxiv.org/abs/1902.03383.

[26] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function? Characterizing Long-Term Trends in Production Serverless Workloads. In *ACM Symposium on Cloud Computing (SoCC '23)*.

[27] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *28th Symposium on Operating Systems Principles (SOSP '21)*.

[28] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *ACM Symposium on Cloud Computing (SoCC '19)*.

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*.

[30] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In

USENIX Annual Technical Conference (ATC '19).

[31] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *19th European Conference on Computer Systems (EuroSys '24)*.

[32] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[33] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *USENIX Annual Technical Conference (ATC '21)*.

[34] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *ACM Symposium on Cloud Computing (SoCC '23)*.

[35] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '23)*.

[36] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *USENIX Annual Technical Conference (ATC '22)*.

[37] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2024. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23)*.

[38] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. 2021. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *11th Workshop on Programming Languages and Operating Systems (PLOS '21)*.

[39] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*.

[40] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. 2023. The Gap Between Serverless Research and Real-world Systems. In *ACM Symposium on Cloud Computing (SoCC '23)*.

[41] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *19th European Conference on Computer Systems (EuroSys '24)*.

[42] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION: Optimized Execution Latency for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*.

[43] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: a benchmark suite for serverless computing. In *14th ACM International Conference on Distributed and Event-Based Systems (DEBS '20)*.

[44] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2022. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Comput. Surv.* (2022).

[45] Netflix. 2021. The Making of VES: the Cosmos Microservice for Netflix Video Encoding. https://netflixtechblog.com/the-making-of-ves-the-cosmos-microservice-for-netflix-video-encoding-946b9b3cd300.

[46] Netflix. 2021. The Netflix Cosmos Platform. https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad.

[47] 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 *USENIX Annual Technical Conference (ATC '18)*.

[48] Apache OpenWhisk. 2024. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[49] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *26th Symposium on Operating Systems Principles (SOSP '17)*.

[50] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *ACM Symposium on Cloud Computing (SoCC '21)*.

[51] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[52] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, et al. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *29th Symposium on Operating Systems Principles (SOSP '23)*.

[53] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *17th European Conference on Computer Systems (EuroSys '22)*.

[54] Trever Schirmer, Valentin Carl, Tobias Pfandzelter, and David Bermbach. 2023. ProFaaStinate: Delaying Serverless Function Calls to Optimize Platform Performance. In *9th International Workshop on Serverless Computing (WoSC '23)*.

[55] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2022. Fusionize: Improving Serverless Application Performance through Feedback-Driven Function Fusion. In *IEEE International Conference on Cloud Engineering (IC2E '22)*.

[56] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2023. Fusionize++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization. https://arxiv.org/abs/2311.04875.

[57] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 *USENIX Annual Technical Conference (ATC '20)*.

[58] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *27th ACM Symposium on Operating Systems Principles (SOSP '19)*.

[59] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *USENIX Annual Technical Conference (ATC '20)*.

[60] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *21st International Middleware Conference (Middleware '20)*.

[61] Josef Spillner. 2019. Quantitative Analysis of Cloud Function Evolution in the AWS Serverless Application Repository. https://api.semanticscholar.org/CorpusID:152282338.

[62] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *50th Annual International Symposium on Computer Architecture (ISCA '23)*.

[63] 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 *IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS '20)*.

[64] 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 *ACM Symposium on Cloud Computing (SoCC '22)*.

[65] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*.

[66] 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 *USENIX Annual Technical Conference (ATC '21)*.

[67] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud. In *16th European Conference on Computer Systems (EuroSys '21)*.

[68] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[69] 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 *ACM Symposium on Cloud Computing (SoCC '20)*.

[70] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*.

[71] 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 *28th Symposium on Operating Systems Principles (SOSP '21)*.

[72] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: Shielded Enclaves for Cloud FPGAs. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[73] Ziming Zhao, Mingyu Wu, Haibo Chen, and Binyu Zang. 2024. Characterization and Reclamation of Frozen Garbage in Managed FaaS Workloads. In *19th European Conference on Computer Systems (EuroSys '24)*.