Enabling In-Vitro Serverless Systems Research

Dmitrii Ustiugov*
NTU Singapore
Singapore

Dohyun Park*
UIUC
USA

Lazar Cvetković
ETH Zurich
Switzerland

Mihajlo Djokic*
IBM Research Europe
Switzerland

Hongyu Hè
ETH Zurich
Switzerland

Boris Grot
University of Edinburgh
UK

Ana Klimovic
ETH Zurich
Switzerland

ACM Reference Format:
Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. 2023. Enabling In-Vitro Serverless Systems Research. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605181.3626191

Abstract

Serverless is an increasingly popular cloud computing paradigm that has stimulated new systems research opportunities. However, developing and evaluating serverless systems in a research setting (i.e., "in-vitro", without access to a large-scale production cluster and real workloads) is challenging yet vital for innovation. Recently, several serverless providers have released production traces consisting of large sets of functions with their invocation inter-arrival time, execution time, and memory footprint distributions. However, executing the workload synthesized from these traces requires a massive cluster, making experiments expensive and time-consuming.

In this work, we show how to use the data available in production traces to construct *workload summaries* of *configurable scales* that remain highly *representative* of the original trace characteristics and can be used to evaluate serverless systems in-vitro. Compared to random sampling of functions from the original trace, our method can generate summaries of up to 10× higher representativity, measured as the average of the Wasserstein distances of the distributions of interest (e.g., function execution time and invocation inter-arrival time) from the respective distributions in the original trace. We release our toolchain that enables researchers to synthesize representative workload summaries and show how it can be used to evaluate the performance of serverless systems at diverse load scale factors.

*Work done while at ETH Zurich.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WORDS '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0250-1/23/10…$15.00
https://doi.org/10.1145/3605181.3626191

1 INTRODUCTION

Serverless has emerged as a popular cloud computing paradigm that offers a convenient high-level abstraction to the cloud for users, with cloud providers taking responsibility for managing and scaling resources based on real-time application load. In contrast to traditional Infrastructure-as-a-Service computing, in which users select virtual machines (VMs) with pre-determined ratios of CPU, memory, storage, and networking resources, serverless computing allows the underlying platform to manage and optimize resource allocations across all workloads to improve performance and cost-efficiency. Optimizing resource management for serverless workloads is a key systems research challenge.

Exploring serverless systems research directions requires access to representative workloads. Although several cloud providers have released production traces of serverless function invocations [22, 27], several key challenges exist in using these traces for systems research.

First, for anonymity reasons, these traces do not include the source code or executable binaries of the actual serverless workloads, but rather only certain statistics about function invocation and execution patterns. For example, the two-week Azure Functions trace [22] captures each function's invocations at a per-minute granularity, along with various percentiles of the function's execution time and memory

footprint across all of its invocations. This information can be used to synthesize representative workloads for experiments.

Second, replaying an entire workload trace requires significant resources, as traces may be collected at the scale of an entire datacenter for multiple days or weeks. For example, we find that replaying the full Azure Functions trace requires an experimental platform with over 10 thousand CPU cores (§2). Executing experiments at this scale for system design exploration is time-consuming and expensive, limiting the speed of innovation. Unfortunately, simply randomly sampling functions from the trace (as many prior papers have done for their evaluation) can lead to unrepresentative workloads, as functions have skewed invocation patterns and execution time characteristics. Hence, we need a robust methodology for scaling down traces while maintaining representativity.

Finally, evaluating serverless systems with a gradually increasing load factor is desirable, similar to how database benchmarks like TPC offer different scale factors [6]. However, naively sampling a monotonically increasing number of functions from a trace does not necessarily monotonically increase load due to the vast differences in invocation frequency and CPU/memory usage across functions.

Researchers have adopted ad-hoc approaches for down-scaling workload traces; however, these don't fully meet the needs of a robust research methodology. Some works use approaches that fail to preserve key characteristics of the workload trace. Others lack a complete description of the proposed sampling approach, making it difficult to understand and evaluate the technique.

To enable meaningful and reproducible research and analysis, the community needs a methodology for scaling serverless production workload traces to various load scaling factors while preserving key workload characteristics, namely the distributions of function invocation inter-arrival times and CPU/memory resource usage.

We propose a methodology to synthesize representative and statistically meaningful workload summaries from large-scale production workload traces. A particular strength of our methodology is that it can provide workload summaries at various scale and load factors. We demonstrate how our methodology can be applied to the Azure Functions trace [22]. We release the synthesized workload summaries of the Azure trace and our toolchain at https://github.com/vhive-serverless/invitro, which consists of a *trace sampler* and a *load generator*, so researchers can use the methodology for other traces. Our In-Vitro sampler applies an iterative method of sampling functions while minimizing the Wasserstein distance between the full trace and sampled trace distributions of key workload characteristics, namely the function invocation distribution and function resource usage distribution. Compared to the random sampling of functions, In-Vitro generates workload summaries with much higher representativity, i.e., up to 10× lower Wasserstein distances for both the invocation and resource distributions. After that, the In-Vitro load generator executes the sampled trace by synthesizing function invocation requests to a serverless system under test.

2 EXISTING FRAMEWORKS AND METHODOLOGIES

Studying serverless systems without access to production deployments is challenging, due to the lack of visibility into the software system stack, access to the serverless functions that run on top of it, or knowledge of their invocation patterns. Although researchers have introduced several open-source frameworks [2, 3, 8, 13, 16, 26] to address the first issue, the lack of insight into the workload code and traffic patterns remains a problem.

Recently, several serverless providers [22, 27] have released production traces of function invocations, revealing invocation traffic patterns as well as the key serverless application characteristics, such as functions' processing duration and memory characteristics. For instance, the most comprehensive trace released by Azure Functions contains tens of thousands of anonymized functions, the number of invocations of each function in each minute of a 14-day interval, as well as the per-function distributions of processing duration (which accounts for useful execution time on the worker node) and memory usage. While these traces provide valuable insights, they are captured at the scale of thousands of nodes[1] and lack actual executable workloads. As such, these traces cannot be used directly to explore serverless systems research directions, such as scheduling and resource management policies.

To make evaluating new serverless systems or resource management policies practical in an academic lab or company staging cluster, prior works [12, 21, 23] often resort to synthesizing load based on functions randomly sampled from a production trace. Others [14] emphasize the skew in function invocation frequency (often found in real clouds [22, 27]) by choosing one function that accounts for 90% of the total load and a few functions that account for the remaining 10%. With such approaches, the resulting load may not accurately reflect the characteristics of production workloads, which risks misleading observations and system design decisions.

To reduce the computational requirements of large production workloads, researchers explore methodologies for constructing *workload summaries* for database queries [10],

[1]Assuming each function uses 1 CPU core during its execution, we estimate that one would need a 10-20 thousand core cluster (at any point in the trace) to replay the full Azure trace.

GPGPU [28], AI [19], and big data [20]. However, their approaches rely on access to a production system and/or the actual workloads. Unfortunately, currently available traces for serverless applications [22, 27] only contain workload characteristics — this is common for public traces due to anonymization challenges. Furthermore, we aim to develop a methodology that enables analyzing system performance under various loads and finding the peak system load [15]. To the best of our knowledge, existing methodologies lack support for stepping through the system load by generating workload summaries with a monotonically increasing system load.

3 THE *IN-VITRO* METHODOLOGY

Here, we define the metric of representativity for serverless workloads and introduce our approach for synthesizing representative workload summaries of various scales.

3.1 Workload Summary Representativity

We define a workload summary as a subset of the functions included in the original trace. We say a workload summary is *representative* of an original production workload trace if its distributions for the key characteristics are *statistically similar* to the original trace's distributions. In the available serverless cloud traces [22, 27], we identify two key distributions, namely 1) function invocation distribution and 2) resource usage distribution (described below). However, our summarization method can be seamlessly extended to account for an arbitrary set of distributions. To evaluate the representativity of distributions, we use Wasserstein distance (WD)[2] as a common metric of statistical similarity.

Invocation distribution: The invocation distribution captures the distribution of per-function invocation inter-arrival time (IAT). The invocation distribution is critical to maintaining representativity as it significantly impacts the infrastructure state and dynamics (e.g., scheduling decisions), as shown by prior work [22, 24]. For example, 45% of functions in the Azure trace [22] are invoked less than once per hour on average while the most invoked 18% of functions account for >99% of invocations. However, rarely invoked functions may exhibit high cold-start delays, up to seconds [24, 26], which can be much higher than the actual invocation processing duration [9]. Hence, capturing these diverse invocation patterns is essential to drive research in efficient serverless infrastructure. We define the invocation representativity of a workload summary as a WD between the distribution of invocations of the functions in the summary and the distribution of invocations for all functions in the original trace in a given time interval. We refer to it as the *invocation WD*.

Resource usage distribution: The resource distribution captures the distribution of per-function total resource (CPU and memory) usage. Similarly to the invocation distribution, the resource usage distribution is defined per function, but the number of the function's invocations arriving in a given minute is multiplied by the average processing duration and average memory footprint of that function. The resource distribution defines CPU and memory usage in the experiment cluster by the functions in the workload summary. Thus, the resource distribution is a proxy for the overall load applied to the studied system by playing the workload summary. We define the resource representativity of a workload summary by measuring a WD between the load distribution from the workload summary and the load distribution from all functions in the original trace in a given time interval. We refer to this as the *resource WD*.

Given two workload summaries A and B constructed from the same original trace, we call A more representative than B if the arithmetic mean of the invocation and resource WDs of A is smaller that of B.

3.2 Synthesizing Workload Summaries

3.2.1 In-Vitro methodology overview. The *In-Vitro* methodology comprises two steps: sample generation using the original trace and load generation using the collected samples.

Sample generation starts with taking samples across time to capture diurnal and weekly patterns, i.e., we select all functions with invocations arriving in the sampled time period. Then, for each sampled time period, the algorithm samples subsets of functions that are the most representative of the original trace's function invocation and resource usage characteristics. To allow studies under a monotonically changing load, the algorithm generates samples recursively by taking samples with a smaller number of functions from samples with a larger number of functions. Below, we elaborate on the metrics we use for optimizing sample representativity and the details of the sampling algorithm.

After generating a set of samples with a different number of functions corresponding to the sampled time period, the *load generator* drives performance measurement experiments, where each experiment corresponds to a single load level. In each experiment, the load generator reconstructs and steers the invocation traffic from a single sample to the sample's functions. The generator captures the functions' response time, allowing analysts to study overall system performance and compare different systems under the same load. Furthermore, to evaluate the performance of a system under

[2]Wasserstein distance (WD) is a metric of the distance between two probability distributions. Informally, if the distributions are interpreted as two different ways of piling up a fixed amount of dirt, WD is the minimum amount of dirt moved from one pile to another multiplied by the moving distance to equalize the two piles.

different load levels, the load generator can step through the experiments, monotonically increasing the load by choosing samples in the increasing order of their size.

3.2.2 In-Vitro Sampler. Listing 1 summarizes our algorithm for synthesizing workload summaries from a production trace. We assume the trace consists of a set of functions and their key attributes: per-function distributions of invocation frequency, execution time on an end server node, and memory footprint.

```
samples = [] # the resulting array of samples

for excerpt in full_trace.rand_sample_by_time():
  samples[excerpt] = sample_excerpt(excerpt)

return samples

def sample_excerpt(original_trace):
  for size in sample_sizes: # in descending order
    for t in range(trials_num):
      if is_first_sample_to_draw():
        sample = original_trace.rand_sample(size)
      else:
        sample = previous_sample.rand_sample(size)

      candidates.append(sample)

    samples[size] = get_repres_sample(candidates)
    previous_sample = samples[size]

  return samples
```

Listing 1: The In-Vitro sampling algorithm

First, the sampling algorithm randomly samples short trace excerpts, capturing diurnal and weekly patterns. Then, for each trace excerpt, we construct the first workload summary of size n by deriving a sample of n functions with all their attributes from the original trace (LoC 12). The algorithm then recursively derives smaller summaries from that sample (LoC 14) instead of the original trace until the smallest desired summary is derived. For each summary size, the algorithm samples the functions `trials_num` times and then chooses a single most representative sample (LoC 18), i.e., the one with the smallest mean of both WDs. Note that when deriving each individual sample, the algorithm is equivalent to random sampling. However, in contrast to vanilla random sampling, the algorithm chooses the most representative sample from a set of samples based on WDs of each sample in the set.

Deriving summaries recursively enables In-Vitro to generate a monotonically changing load. A large summary has a higher load (i.e., uses more CPU and memory on end server nodes) than any smaller summary derived from it because the former (large summary) includes the same functions *plus* some additional ones compared to the latter (smaller summary). In §4.2, we show that it is sufficient to sample a small number of summaries of the same size to derive a highly representative workload summary using the WD metric and filter out summaries with low representativity.

Users can configure the sample trace duration. Since cloud providers can spawn instances of functions in less than one second [24] and tear down instances after minutes of inactivity [24]. Serverless infrastructure usually tracks instance activity with a window of 60 seconds by default [1]. Hence, one can sample measurement time periods longer than several minutes from the original trace without losing representativity. In our experiments, we conservatively choose to sample 15-minute intervals, each preceded by a 20-minute warm-up interval, the length of which we empirically choose to yield performance measurements reproducible across experiments with the same sample.

3.2.3 In-Vitro Load Generator. To issue invocations from the trace, we develop a load generator that takes memory usage and runtime duration from the sampled trace as inputs to model serverless function workloads. Upon placement of each synthesized function, the load generator requests the maximum memory amount the function uses according to the data in the trace from the cluster manager. The synthesized function's code contains a busy loop that guarantees the consumption of CPU cycles for the requested runtime duration calibrated to the host node's CPU frequency a priori.

If the trace does not contain exact timestamps of function invocations (e.g., the Azure Function trace only contains the invocation count per minute for each function), the load generator reconstructs inter-arrival times conforming to a Poisson distribution at a millisecond scale. The synthesizer determines the exact timestamps when each function has to be invoked and issues a request with the execution time as each invocation's parameters. The synthesizer also measures each invocation's end-to-end latency as time from invocation to response.

4 EVALUATION

We now evaluate the representativity of workload summaries synthesized with the In-Vitro methodology compared to the summaries obtained via vanilla random sampling. We choose uniform random sampling across all functions in the trace as a baseline because this approach is commonly used in prior works on serverless computing [12, 14, 21, 23] sampling captures the high diversity of serverless function characteristics, e.g., including hot and cold functions, and short and long-running functions. We then measure serverless cluster performance under various loads with workload summaries obtained with In-Vitro and random sampling. We conclude

by studying the efficiency of serverless autoscaling policies with our methodology.

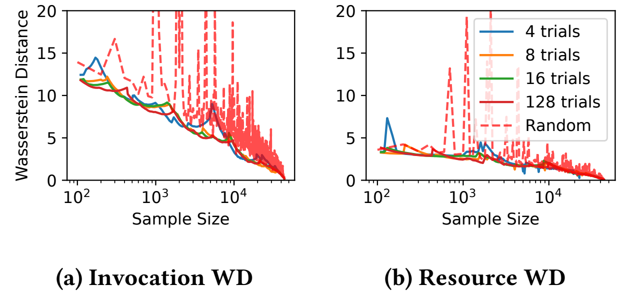
4.1 Experiment Setup

For the evaluation, we use 21 `xl170` nodes in CloudLab[11] running stock Ubuntu 20.04 on a 10-core 2.4 GHz Intel E5-2640 CPU with 64GB DRAM and a 25Gb NIC. We use vHive [26] v1.5, a Kubernetes/Knative-based serverless stack, as a platform representative of a real-world serverless deployment. One node in the cluster is dedicated exclusively to Kubernetes-specific services, Knative services, and cluster monitoring components (Prometheus [5]); another node hosts the workload synthesizer. Other ('worker') nodes run only synthesized functions. We configure Knative to use the concurrency-based autoscaling policy unless specified otherwise. We use the Azure Functions trace [22], pre-processed to exclude malformed entries, for synthesizing summaries.

To compare the performance of two systems deployed in a fixed-size cluster, one needs to evaluate the responsiveness of these systems while sweeping the overall system load. Serverless traces [22, 27] contain heterogeneous functions with various execution times, inter-arrival times, etc., so the analysts need a way to reason about overall system performance in a studied scenario. Hence, we define the overall system *slowdown* as a function of the overall load, evaluating the responsiveness of the studied system under various loads. We measure slowdown as a geometric mean of all per-function slowdowns in the given time interval.[3] The per-function slowdown is defined as the geometric mean of the function's per-invocation slowdowns, each of which is the measured end-to-end response time divided by the processing duration of that invocation as specified in the trace. Finally, we define the load in the studied serverless system as the number of functions in a workload summary. By design of the recursive sampling algorithm (§3.2), workload summaries with more functions always apply a higher load on the evaluated system than the summaries with a fewer number of functions.

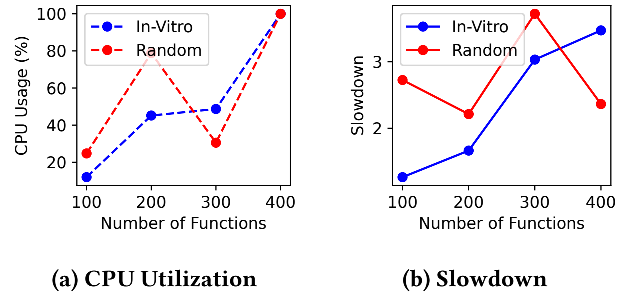
4.2 Sample Representativity Analysis

Figure 1 compares samples' representativity with the In-Vitro methodology and samples obtained with uniform random sampling. We measure representativity as the arithmetic mean of invocation WD and resource WD (§3.1). For both sampling methods, we perform five sampling runs for each sample size and plot the characteristics of the sample with the median WD. In each of the five runs of the In-Vitro sampling for a given sample size, we choose the best sample after several trials and choose the sample run with the median WD.



(a) Invocation WD **(b) Resource WD**

Figure 1: In-Vitro and random sampling methods' WD.



(a) CPU Utilization **(b) Slowdown**

Figure 2: Aggregate CPU utilization (averaged across all worker nodes during a run) and aggregate slowdown when using the In-Vitro summaries and random samples.

Figure 1 shows that the In-Vitro methodology generates samples with lower invocation and resource-usage WDs than random sampling across sample sizes. Indeed, the vanilla random samples have the invocation WD of up to 27 and the resource WD of up to 24. In contrast, the In-Vitro method generates samples with the invocation WD <12, and the resource WD <4. Overall, In-Vitro provides samples with up to 10× lower resource WD than random samples (2.3 vs 23.6). Also, In-Vitro's WD converges monotonically when decreasing the sample size (whereas random sampling exhibits a high WD variation) because In-Vitro filters out functions with an outlier invocation arrival rate, processing duration or CPU/memory usage.

4.3 Analysis with Load Sweep

Figure 2 shows how the cluster's overall slowdown and average CPU utilization vary when sweeping the load by increasing the size of In-Vitro and random samples. Note that with the increase of the size of randomly generated samples, the CPU load (Figure 2a) may increase or decrease because

[3] Slowdown in our definition can be considered a relative alternative to service level objectives, commonly defined as response time percentiles[15, 17]. Instead of the geometric mean, the slowdown can also be defined as a percentile of per-function slowdowns.

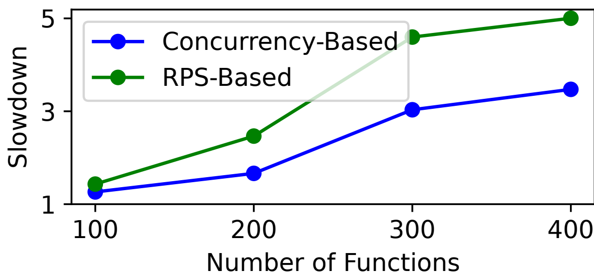


Figure 3: Slowdown with each autoscaling policy.

each sample is drawn independently from the original trace. For example, the 300-function sample comprises many short-running and rarely-invoked functions whose collective CPU load is lower than that of the 200-function sample. In contrast, for In-Vitro, summaries display a monotonically increasing CPU load with an increasing number of functions. Figure 2b shows that load variability across samples affects the overall slowdown, making it difficult to reason about system behavior under low and high loads.

4.4 Comparing Autoscaling Policies

We show an example use case for applying the In-Vitro methodology for serverless system research and design space exploration. We use In-Vitro workload summaries to compare the performance of two autoscaling policies available in the Knative [4] serverless system: concurrency-based (used by default) and request-per-second-based (RPS-based) autoscaling. The former makes scaling decisions based on the sum of the number of invocations waiting and currently processed in the system, whereas the latter policy relies on each function's estimated invocation arrival rate and execution time. As before, we gradually increase the function count to sweep the load on the cluster and measure slowdown.

Figure 3 shows that the concurrency-based scaling policy delivers consistently higher performance (i.e., up to 1.7× lower slowdown with the same workload summaries) than the RPS-based policy. However, we observe that the concurrency-based policy uses 2.2× more CPU resources than the RPS policy (not shown in the figure), indicating the performance-cost trade-off. The intuition for these results is that the RPS-based scaling policy only reacts to the incoming rate of requests, thus it cannot react to the state of the system (e.g., build-up of queues) as fast as the concurrency-based policy, hence causing higher slowdowns of function invocations. In contrast, the concurrency-based policy drives scaling based on queue occupancy, naturally depleting the queues faster than its counterpart.

5 EARLY EXPERIENCE WITH IN-VITRO AND FUTURE WORK

In-Vitro aims to introduce a standard method for comparing different systems under load generated from the same standard trace[4] and open many opportunities for evaluating serverless systems of different kinds.

In our experience, In-Vitro has shown its high workload compression efficiency: in our experience, 10-20 cluster nodes suffice to observe the effects of queuing on various control and data plane components in a Knative/Kubernetes-based distributed system. Also, In-Vitro has a large application scope. For instance, apart from black-box serverless system evaluation, we have been actively using In-Vitro to study the implications of control-plane traffic on the cluster manager's internal services in Knative and Kubernetes. We also use In-Vitro to study system support for new serverless applications, such as AI-as-a-Service including elastic machine-learning inference and LLM prompt-tuning systems, the application traffic in which is similar to the Azure Functions trace [18].

In future, we plan to compare our observations in a small-scale research setting to the phenomena observed in large-scale commercial platforms, which we plan to study by playing the In-Vitro traces on functions deployed in real providers, such as AWS Lambda, Azure Functions, Google CloudRun, Alibaba Function Compute. Thus, In-Vitro can complement STeLLAR, our previous work on black-box benchmarking of various subsystems of commercial clouds [25].

The scope of serverless computing has been expanding dramatically, so we anticipate that In-Vitro would help researchers in other emerging areas, such as cloud-edge systems [29] and Industry 4.0 [7].

6 CONCLUSION

Exploring the design of serverless systems requires access to representative workloads. While recently released production traces contain useful statistics of real workloads, to foster system innovation, they require a methodology for generating representative workload summaries to run experiments in a small-scale research setting. We introduce a methodology to derive representative workload summaries at configurable scales from production traces and show the utility of the toolchain we release by analyzing two autoscaling policies' performance.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their insightful feedback, Tom Kuchler for conducting early studies of the invocation traces, Marios Kogias and the EASL lab for their valuable comments and advice.

[4]We release the standard sampled traces on GitHub: https://github.com/vhive-serverless/invitro.

REFERENCES

[1] Additional Autoscaling Configuration for Knative Pod Autoscaler. Available at https://knative.dev/docs/serving/autoscaling/kpa-specific/#stable-window.

[2] Fission: Open Source, Kubernetes-Native Serverless Framework. Available at https://fission.io.

[3] Fn project. Available at https://fnproject.io.

[4] Knative. Available at https://knative.dev.

[5] Prometheus. Available at https://prometheus.io.

[6] Transaction Processing Performance Council. Available at https://www.tpc.org.

[7] What are Industry 4.0, the Fourth Industrial Revolution, and 4IR? Available at https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-are-industry-4-0-the-fourth-industrial-revolution-and-4ir.

[8] Apache. OpenWhisk. Available at https://openwhisk.apache.org/.

[9] Datadog. The State of Serverless 2021. Available at https://www.datadoghq.com/state-of-serverless-2021.

[10] Deep, S., Gruenheid, A., Koutris, P., Naughton, J. F., and Viglas, S. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow. 14*, 3 (2020), 418–430.

[11] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K.-C., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)* (2019), pp. 1–14.

[12] Fuerst, A., and Sharma, P. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)* (2021), pp. 386–400.

[13] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2016).

[14] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Hermod: Principled and Practical Scheduling for Serverless Functions. In *Proceedings of the 2022 ACM Symposium on Cloud Computing (SOCC)* (2022), pp. 289–305.

[15] Kogias, M., Mallon, S., and Bugnion, E. Lancet: A Self-Correcting Latency Measuring Tool. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)* (2019), pp. 881–896.

[16] Kubeless. Kubeless: The Kubernetes Native Serverless Framework. Available at https://kubeless.io.

[17] Leverich, J., and Kozyrakis, C. Reconciling High Server Utilization and Sub-Millisecond Quality-of-service. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 4:1–4:14.

[18] Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., and Stoica, I. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2023), USENIX Association.

[19] Liang, M., Fu, W., Feng, L., Lin, Z., Panakanti, P., Zheng, S., Sridharan, S., and Delimitrou, C. Mystique: Enabling Accurate and Scalable Generation of Production AI Benchmarks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023* (2023), Y. Solihin and M. A. Heinrich, Eds., ACM, pp. 37:1–37:13.

[20] Panda, R., Zheng, X., Gerstlauer, A., and John, L. K. CAMP: Accurate Modeling of Core and Memory Locality for Proxy Generation of Big-Data Applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018* (2018), J. Madsen and A. K. Coskun, Eds., IEEE, pp. 337–342.

[21] Saxena, D., Ji, T., Singhvi, A., Khalid, J., and Akella, A. Memory Deduplication for Serverless Computing with Medes. In *Proceedings of the 2022 EuroSys Conference* (2022), pp. 714–729.

[22] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)* (2020), pp. 205–218.

[23] Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S., and Akella, A. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the 2021 ACM Symposium on Cloud Computing (SOCC)* (2021), p. 138–152.

[24] Ustiugov, D., Amariucai, T., and Grot, B. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)* (2021), pp. 51–62.

[25] Ustiugov, D., Amariucai, T., and Grot, B. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *IEEE International Symposium on Workload Characterization (IISWC)* (2021).

[26] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)* (2021), pp. 559–572.

[27] Wang, A., Chang, S., Tian, H., Wang, H., Yang, H., Li, H., Du, R., and Cheng, Y. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)* (2021), pp. 443–457.

[28] Yu, Z., Eeckhout, L., Goswami, N., Li, T., John, L. K., Jin, H., Xu, C., and Wu, J. GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation. *IEEE Trans. Computers 64*, 11 (2015), 3153–3166.

[29] Zhang, J., Jin, C., Huang, Y., Yi, L., Ding, Y., and Guo, F. KOLE: Breaking the Scalability Barrier for Managing Far Edge Nodes in Cloud. In *Proceedings of the 2022 ACM Symposium on Cloud Computing (SOCC)* (2022).