OrcBench: A Representative Serverless Benchmark

Ryan Hancock
University of Waterloo
Waterloo, Canada
krhancoc@uwaterloo.ca

Sreeharsha Udayashankar
University of Waterloo
Waterloo, Canada
s2udayas@uwaterloo.ca

Ali José Mashtizadeh
University of Waterloo
Waterloo, Canada
ali@rcs.uwaterloo.ca

Samer Al-Kiswany
University of Waterloo
Waterloo, Canada
alkiswany@uwaterloo.ca

***Abstract*—Serverless computing is rapidly growing area of research. No standardized benchmark currently exists for evaluating orchestration level decisions or executing large serverless workloads because of the limited data provided by cloud providers. Current benchmarks focus on other aspects, such as the cost of running general types of functions and their runtimes.**

We introduce OrcBench, the first orchestration benchmark based on the recently published Microsoft Azure serverless data set. OrcBench categorizes 8622 serverless functions into 17 distinct models, which represent 5.6 million invocations from the original trace.

OrcBench also incorporates a time-series analysis to identify function chains within the dataset. OrcBench can use these to create workloads that mimic complete serverless applications, which includes simulating CPU and memory usage. The modeling allows these workloads to be scaled according to the target hardware configuration.

***Index Terms*—benchmark, serverless, cloud, modeling**

I. Introduction

The serverless paradigm has seen a growth in popularity over the last few years [1, 10] with AWS seeing a 209% increase in functions executed in 2020 [11]. Serverless simplifies service deployment by shifting much of the infrastructure responsibility from developers to the cloud provider. Developers focus on the functionality of their applications while the cloud takes care of most of the deployment challenges.

Building a cost efficient serverless platform is challenging because developers are only charged for the resources used during a function's execution. Offloading the application deployment and management from developers to providers introduces complex research challenges including resource management, auto-scaling, workload consolidation, storage systems, billing, and orchestration. A benchmarking tool that simulates real workload characteristics is essential to explore these frontiers.

There are currently no benchmarking tools that capture the characteristics of real serverless workloads. Previous efforts resorted to building tools that were limited by the lack of publicly available data from serverless providers. These tools provide performance and cost estimates for running a serverless application on a given cloud platform [2, 7], or provide a synthetic set of functions that represent applications within the serverless paradigm [4].

We present OrcBench, a serverless workload generator that generates data center workloads that model the recently released Microsoft Azure dataset [14]. OrcBench generates workloads which mimic the interarrival time, resource usage, and execution times of the real world Azure traces. OrcBench supports varying cluster sizes and workload intensity, and can be used to generate complete serverless applications that chain multiple functions.

The modeling of Azure traces is challenging because of two reasons. First, the traces do not include the exact invocation time, instead, batching all invocations at minute level granularity. Second, the dataset contains 52 thousand functions and a total of 8.8 billion function invocations over a two week period. Third, there are many applications with different interarrival patterns and resource usage.

To accurately model these functions, we group them based on their interarrival times using EP-Means clustering [3]. EP-Means clustering groups the empirical cumulative distribution functions (ECDF) of each function's interarrival times. Clustering this way allows us to group similarly behaving functions together. We use the centroid of each cluster as a representative function for its group.

These representative functions are used in place of the hundreds to thousands of functions in a cluster. Each group also has a probability distribution for both CPU and memory constraints. When a user requests a simulated function this underlying distribution is sampled to give each function how much memory and CPU to consume.

Clustering treats each function independently without considering the relationships between functions. An application may consist of multiple functions that form a chain that is executed in series. We use time-series analysis to discover the relationships between functions within each application and allow us to model these chains.

OrcBench grouped 8622 functions into 17 distinct groups, and the models produced from these groups were used to generate an equivalent number of synthetic functions and traces. These 8622 simulated functions invoked 5.6 million times over 30 minutes with an average error rate of 15%. A major source of error comes from extrapolating sub-minute behavior from the one minute resolution timestamps in the Azure data set. Our two highest invoked models (representing 113 functions) accounted for 2.8 million invocations while having only an average error rate of 4.7%. The remaining functions were rarely invoked or timer based functions that were excluded from our modeling.

This research is supported by grants from NSERC and WHJIL.

II. Overview of the Microsoft Azure Traces

The Microsoft Azure dataset [8] is a collection of 52 thousand functions which were invoked 8.8 billions times over a 14 day period. The three main objects in the data are functions, applications and owners which are identified through anonymous hash IDs. Owners can own multiple applications, and applications can be composed of many functions potentially invoking each other to form function chains. This captured data is broken up into three major parts: a time-series of invocations, execution time, and memory usage.

Invocations: The invocation time-series contains the number of invocations of a function at each minute of the 14 day trace.

Execution Time: The dataset contains the average execution time and a fixed set of percentiles for each function. The execution time percentiles for the 0^{th}, 1^{st}, 25^{th}, 50^{th}, 75^{th}, 99^{th}, and 100^{th} are included. The execution times do not include the cold start of the function runtime.

Memory Usage: The data set includes the average memory usage for each *application* and is also broken into a fixed set of percentiles. Unlike the other data that is recorded per function, memory usage is recorded for the entire application because Azure packages resource allocation bounds for functions belonging to the same application together for pricing [9].

III. Methodology

Modeling the Microsoft Azure dataset requires overcoming three challenges: First, the dataset is temporal (See Figure 1). Each function fluctuates throughout the day, often seeing a steady rise over working hours (i.e., 9am to 5pm) and declining when the workday is over.

Second, the anonymization of the data reduces assumptions we can make based off additional information such as function naming. This particularly affects our ability to determine where functions exist within a chain.

Lastly, the traces of each function have low resolution timestamps of minute granularity. The execution times of most functions is far smaller (i.e., 90% of functions have an execution time of less than 10 seconds [14]) than the one minute resolution requiring us to extrapolate a model for the execution behavior for timescales less than one minute.

It is important to translate the dataset from an invocation time series to interarrival time as this decouples time from our model. However, the low resolution of the data makes this difficult. For example, functions commonly have large contiguous sections of non-zero entries in their traces, which when naively averaged to calculate the interarrival time leads to a single highly frequent data point. In a real world application we would expect a far more continuous set of data points in regards to interarrival time. Averaging would also completely flatten our invocation rate, causing us to lose important patterns that can occur during the trace.

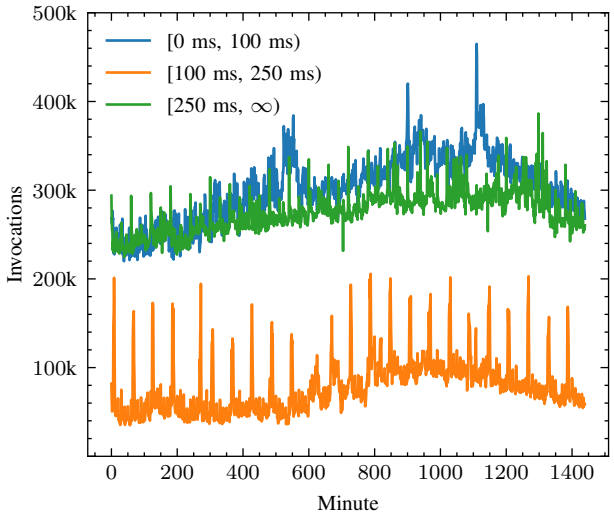


Fig. 1: Invocation counts of functions broken up by execution time in milliseconds over the course of a day

A. Modeling Functions

The modeling of the function invocation rates can be broken into three stages: trimming, data expansion and grouping.

a) Trimming: To reduce temporalness (Figure 1), we first isolate the invocation data to a single, 30 minute window of time between 12:00 PM and 12:30 PM. By sampling a smaller window of time we reduce temporal behavior within our model at the cost of accuracy. An incredibly short window would be unaffected by temporal behavior but the model could miss interesting patterns that may be expressed in the data. As this window grows, the model captures more interesting patterns.

However, if its grows too large these patterns can be difficult to detect because of temporal behavior. An outcome of choosing this time window is that our models are a representation of the workload at that specific time. For example, models created around peak hours invoke functions more frequently than models created from an earlier or later time window.

We then trimmed the data to contain only functions that were invoked at least 10 times during the selected 30 minute time period and also excluded functions triggered by timers. Timer functions can be trivially modeled and rarely invoked functions lack data for proper modeling.

After trimming we split the data into the top 1% of highly invoked functions and the remaining functions. We model these two groups using the same techniques but separately. Clustering highly invoked functions with lesser ones mask the interesting behavior of these infrequently invoked functions.

The data was further split between weeks, with the first week being used for modeling while the second was used for evaluation. We excluded weekends as outliers as functions exhibited very different behavior.

b) Data Expansion: The next step is to translate our invocation traces to interarrival time. Translating from invocations per minute to interarrival time makes our models independent of each minute. This translates our model from

a function telling us the number of invocations to expect at a given minute to how long we should expect until the next invocation.

As previously stated, if we naïvely average our traces to calculate our interarrival times, we are left with very few data points for modeling. This also has the disadvantage of flattening each function that causes the loss of interesting patterns in the data. We can extract more data points if we make an assumption that for any given function, the amount of invocations at any given minute is independent to any other minute within the same function.

The assumption of minute to minute independence allows us to instead view each minute in the invocation data as a Poisson process. We then can generate as many data points as invocations by using each minute as a hyper parameter to the Poisson distribution. The sampling of this distribution allows us to transform a discrete dataset into a continuous one, which better represents a realistic invocation pattern.

We believe this assumption to be sound as this stage of the modeling only focuses on each individual function rather than relationships between functions. We could not make this assumption between two minutes of two separate functions as they may belong to the same application and one function may trigger another during its execution.

c) Grouping: We then create empirical cumulative distribution functions (ECDF) for each function with this newly expanded interarrival data. The ECDF is a model of a function which is then grouped to form clusters with other similarly shaped functions. A cluster centroid is chosen that is the model which replaces all functions of the cluster.

We cluster the functions using EP-Means that clusters similar ECDFs together. However, a challenge occurs when clustering on the interarrival times. Specifically, distances between interarrival times is not representative of the behavior. For example, two functions that have a constant interarrival times of 0.01 s and 0.1 s seem close, but have a $10\times$ difference in invocation frequency.

To remedy this we use EP-Means clustering on the inverse of the interarrival times. We create ECDFs of this frequency data and use EP-Means to retrieve each grouping. From here we translate back to interarrival time and re-calculate the centroids of each group.

d) Sampling: Once models are created, we now face the challenge of properly sampling a function that represents many functions. If we were to sample our created model once for each of its represented functions, then at any given minute we would likely just see the average of the model. To overcome this, OrcBench allows for a hyper-parameter (N) which sets how many functions a model is used for, which we call the sampling group.

We then randomly spread invocations for each function being represented within the sampling group throughout the range of the sampled interarrival time (See Figure 2). Although we are modifying the interarrival time for that set of invocations, if we sample the interarrival time and randomly place these invocations, then the average interarrival time would trend towards the initial sampled time.

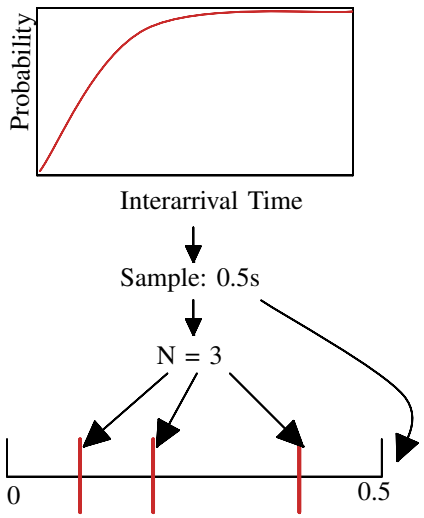


Fig. 2: An example of sampling from an ECDF of a model with a hyper-parameter of $N = 3$, and how we spread invocations within a sampled interarrival time. Each vertical line on the timeline from 0s to 0.5s represents a separate function.

B. Function Chaining

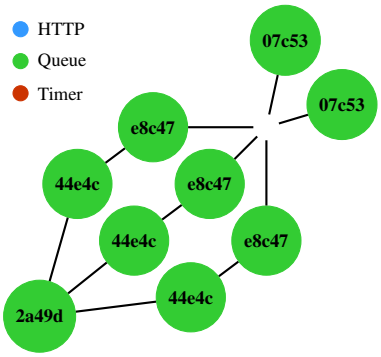
A distinguishing characteristic of serverless computing workloads is the invocation dependency between serverless functions within the same application. An application often includes multiple functions that invoke each other creating a call graph. Our goal is to generate workloads that mimic complete serverless applications and their call graphs.

The Azure traces identify functions that are part of the same application, however, they do not provide information about the application call graph. Ideally, one can infer the call graph through the ordering of invocation timestamps. Unfortunately, this approach is not possible using the Azure traces because of the low resolution of the trace.

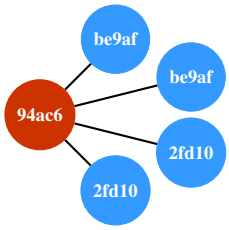
Our insight is that a function that experiences a change in the invocation count from one minute to the next should result in an proportional change in the invocation counts of any function it calls. This does not hold all the time, e.g., a function that conditionally calls one of two functions. However, in this case, the callee should see an increase in invocations proportional to the conditional branch.

We apply the Spearman correlation coefficient to each function's trace, which is a time-series of invocations. The coefficient determines how strongly two functions within an application are tied to each other.

a) Function Fan-out Ratio: A single function may call other functions many times. For example, a MapReduce style data analytics application may call multiple map functions. To identify such application structures, we examine *function call*



(a) Example of an execution graph (application id: 9cb9a). The graph includes four processing stages. Two instances of function 07c53 are correlated with three invocations of function e8c47. All functions are trigger by queue events.



(b) Example of an execution graph (application id: 88946) with three functions.

Fig. 3: Examples of function graphs. Circles represent functions and the number in the circle are the function ids in the Azure trace. The circle colors indicate the trigger type.

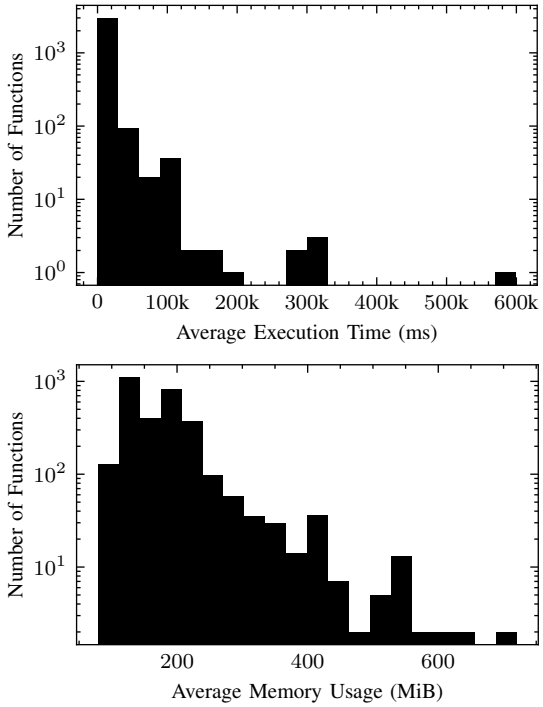


Fig. 4: Example CPU and memory histograms which back each model for generating function stubs.

ratios. Function call ratios is the amount a caller function calls a callee.

The observed call ratio may vary over time. We start by recording the ratio of invocations between all pairs of correlated functions at each minute, then use the most common ratio for each function pair across the dataset. Figure 3 shows two example graphs found using this technique. Figure 3a shows an application with a fan out of three. Figure 3b shows an application that is composed of three functions where function 88946 is correlated to functions 2fd10 and be9af.

b) Pruning the Call Graph: Finding the calling order is difficult because of the minute granularity. We can view our application as firstly starting as fully connected graph and we use several heuristics to eliminate edges from this graph. These heuristics makes the assumption that applications will consist of a single root function and the call graph contains no cycles. We use three heuristics to remove edges until a root function is identified:

First, we identify if one of the functions is a possible root through its trigger type. For instance, a function could be the only function triggered by an HTTP event while the rest are triggered by orchestrator events. Orchestrator events can only occur through other functions.

Second, we analyze the time series to find instances where an application execution is split over two minutes. The function called in the first minute is identified as the caller function, allowing us to remove the backwards edge between the two nodes in the graph.

Third, we start at edges in the graph that have an identified direction and propagate the direction outwards to identify candidate roots. We continue doing this with other identified edges to eliminate possible candidates. Once all edges are exhausted, the root node or the list of candidates is returned.

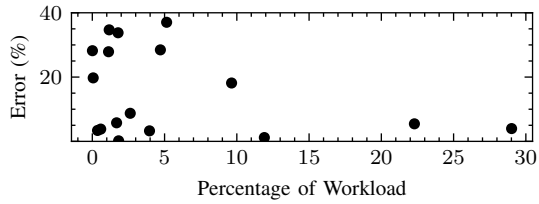
c) Application Workload Generation: We now have a groups of applications and their root functions. A user can choose one of these applications to use for their workload. If more then one candidate root is attached to the application the user must select one. We associate these root functions to their model from §III-A to generate function invocations. On each invocation, OrcBench simulates the execution of the call graph of the application and records the trace.

Users can further get OrcBench to generate function stubs for their application. OrcBench does this by sampling the underlying histogram of CPU and memory data of the root model (See Figure 4 as an example). We found no strong correlation between CPU time and memory when looking at each of our models, so we can sample these histograms to help generate our synthetic functions. These samples are then input into a template function, which when invoked, will allocate memory and execute for the specified amounts.

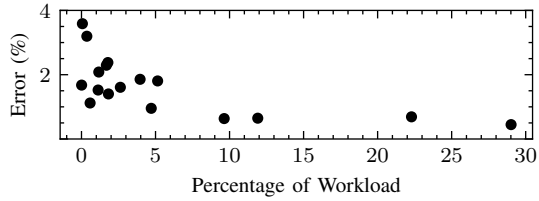
IV. Evaluation

Our evaluation examines the following questions:

- RQ1: How accurate is the individual function traces created by OrcBench?



(a) Absolute Total Error of Functions over 30 minute period.



(b) Overall Error Percentage vs. Percentage of Workload. Each point on this graph represents a mode, with the Y axis representing the percentage error this model contributes to the overall workload per percentage of work done by the workload, and the X representing how much of the overall workload the model contributes to.

Fig. 5: Error Rates within the OrcBench Models

- RQ2: How accurate are the function chains discovered by OrcBench?

In §IV-A we evaluate how close our models are to each of the functions we are modeling and how much each model contributes to the total error. Lastly, in §IV-B we evaluate an inferred application chain.

A. Models

a) Generated Model Traces: Figure 6 shows example traces generated by OrcBench which compares the invocation data to that of its model. We compare the model to the second week of the dataset. As previously stated, we use a hyper-parameter for the number of functions each model will represent during the trace. All evaluations were done with $N = 10$.

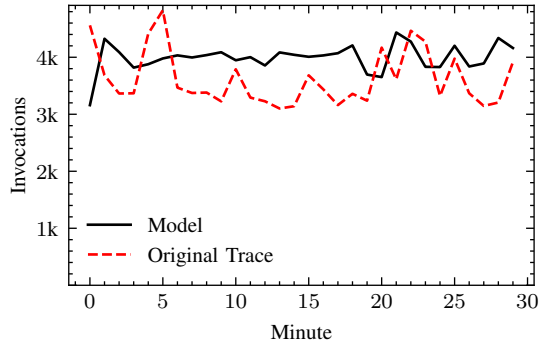
b) Model Error Rate: Our clustering found 17 distinct models which represent a total of 8622 functions, with a workload being scaled to one that matches the Azure dataset executing 5.6 million invocations over a 30 minute period.

We examined our results to determine the following:

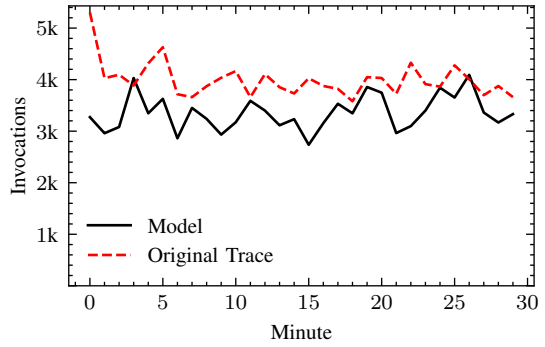
- How much error does each model have relative to the group of original functions the model is representing?
- How much error do models have relative to each other?

We simulated a trace using each model and calculated the mean squared error (MSE) between the simulated trace and each original function trace within its group. We added this error to a sum then normalized it with every other model.

We then normalized the total invocations the model contributes to the error. This is the ratio of the normalized error to the normalized model invocations, i.e., how much error a model contributes per percentage of the total workload. Figure 5b shows this value for every model. Note that many higher



(a) Model of 3142 Functions



(b) Model of 272 Functions

Fig. 6: Example traces of our models which have been scaled to the number of functions to the functions they originally represent. We compared these modeled functions to the sum of the traces of the original functions.

error models represent less than 1% of the total workload. Figure 5a shows how much total error accumulated during the course of a 30 minute run. We aggregate the total MSE between the model trace and each function trace.

We see a trend with the error that shows models which represent a a larger weight of the overall workload have lower error. This occurs as these models have far more data points to create a better fitting function leading to a more refined centroid being created during clustering. Further, as more function invocations occur within an individual minute the extrapolated Poisson model used for each minute becomes more accurate. The average absolute error for all of our models is 15%.

B. Function Chaining

We evaluated a discovered application call graph using a simulated call chain. We first selected an application with an identified root and attach its model. We provide the structure of the call chain and execution times for each function within the application. The simulator works by executing callee functions after the caller function completes. We consider the fan-out ratios when invoking the following functions.

The trace is recorded and then compared to the original function invocation traces of functions within the application.

Our discovered application was found to have a 6% average error rate when compared to the original application trace.

V. Related Work

Work within serverless computing that focuses on orchestration infrastructure [5, 6, 12, 13] often uses open source applications to evaluate and compare their designs. The solutions focus on individual applications and their latency/throughput. Due to the lack of any global workload benchmark, readers are left to assume how these design decisions affect the infrastructure as a whole.

Benchmarks like FaaSdom [7] and SeBs [2] focus on benchmarking cloud providers themselves to give insights into each cloud provider's expected function runtime and cost. These benchmarks provide a strong basis for micro benchmarks over a larger workload based one like OrcBench.

VI. Conclusion

In this paper we introduce OrcBench, a serverless workload and synthetic function generator. We provided the first orchestration level benchmark, which allows researchers to better study serverless environments.

OrcBench's modeling approach allows for workloads to be scaled as needed to meet hardware goals. When compared to the original Microsoft Azure data, OrcBench was able to produce models which represent 8622 functions and invoke 5.6 million times over a 30 minute period with an average error of 15%. OrcBench uses various techniques to infer sub-minute behavior of its functions to overcome the original data sets low resolution traces. OrcBench can be found at https://github.com/rcslab/orcbench.

VII. Acknowledgments

The authors thank Tavian Barnes for his advice during the creation of OrcBench. This research was supported by grants from Canada Natural Sciences and Engineering Research Council (NSERC) and Waterloo-Huawei Joint Innovation Lab.

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The rise of serverless computing. *Commun. ACM*, 62 (12):44–54, nov 2019. ISSN 0001-0782.

[2] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing, 2020.

[3] K. Henderson, B. Gallagher, and T. Eliassi-Rad. Ep-means: An efficient nonparametric clustering of empirical probability distributions. New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450331968.

[4] J. Kim and K. Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[5] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/kotni.

[6] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 782–796, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507717. URL https://doi.org/10.1145/3503222.3507717.

[7] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380287. doi: 10.1145/3401025.3401738. URL https://doi.org/10.1145/3401025.3401738.

[8] Microsoft. Azure functions dataset 2019. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md, 2019.

[9] Microsoft. Azure functions developer guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference?tabs=blob#function-app, 2022.

[10] A. Passwater. 2018 serverless community survey: huge growth in serverless usage. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage, 2018.

[11] N. Relic. Serverless benchmark report. https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020, 2020.

[12] F. Romero, G. I. Chaudhry, Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. arXiv, 2021. URL https://arxiv.org/abs/2104.13869.

[13] R. B. Roy, T. Patel, and D. Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051.

[14] M. Shahrad, R. Fonseca, Íñigo Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, 2020.