In-Vitro Serverless Clusters

Hongyu Hè
honghe@inf.ethz.ch
ETH Zurich

Abstract

Serverless has been fast advancing in many fronts such as cold starts, virtualization, scheduling, workflow management, etc. Many open-source serverless systems have been built, and various benchmark suites have been developed. However, the development of serverless features is not going hand-in-hand with the experimental methodology. While being comprehensive and heterogeneous, existing frameworks and benchmarks can hardly address fundamental methodological problems.

With Azure function workloads, we show that experimental methodology is indispensable by identifying problems in existing approach that can lead to substantial distribution shifts in workload models, large variation in performance measurements, and 100× througput overestimation. To counter this challenge, we propose a methodology for end-to-end serverless experiments, including workload modeling, cluster configuration, implementation of benchmark function, and load generation. Further, we build an off-the-shelf, open-source platform with our methodology integrated, enabling in-vitro experiments with clusters of arbitrarily smaller sizes. We expect our methodology and platform to facilitate easier, more reliable and reproducible serverless experiments.

1 Introduction

Serverless computing is an increasingly popular execution model in the cloud that offers high elasticity and fine-grain billing while freeing users from the burden of resource management. Serverless cloud offerings have not only attracted a high volume of users running real-world applications; they have also stimulated systems research efforts in academia and industry. For example, researchers have optimized serverless sandbox initialization latency [7, 8, 16, 43], scheduling algorithms [40, 48], memory deduplication [46], and data communication [7, 15, 28, 35].

However, conducting system research on serverless infrastructure "in-vitro"—in a research lab setting without access to the internal system implementation of real serverless infrastructure from major cloud providers—is challenging as it requires a state-of-the-art open-source platform and a representative set of workloads. Recognizing this need, numerous open-source platforms (e.g., [1, 2, 32, 50]), benchmark suites (e.g., [12, 27, 36]), and real workload traces (e.g., [33, 47]) have been released in recent years by academia and industry. While being instrumental, none of them addresses fundamental but crucial methodological issues of "in-vitro" serverless infrastructure experiments. In this paper, we bring to light and propose solutions to the following methodology issues in serverless experiments at cluster scale in a lab setting.

Workload modeling. In-vitro serverless experiments require modeling real workloads using only a (small) subset of production traces while preserving job distributions (invocation frequency, execution time, memory footprint, etc.) [30, 34, 42, 51]. This need springs from two facts: (1) the infrastructure available to a typical lab setting can be relatively miniature (e.g., tens or hundreds of nodes) compared to a production environment (millions of nodes [49, 52]), and (2) the development and evaluation of system policies are preferably conducted with a small fraction (e.g., 0.9% [30]) of the machines used in production. However, constructing a sound workload model is particularly challenging in serverless where function workloads feature skewed distributions that are hard to approximate [24, 47, 48, 54]. Consequently, the workload models employed in prior work suffer from substantial distribution shifts (§2.1), which could have a negative impact on the results and conclusion since the job distributions used for development and evaluation are unlikely to occur in production.

Cluster setup & configuration. Having a representative configuration and stable setup is critical in systems experiments [26, 55], without which, the results could be statistically unsound and implicitly irreproducible. While incorporating many state-of-the-art features, none of existing serverless platforms can readily host real workloads without onerous manual setup and configuration. Moreover, they do not have integrated tools (e.g., load generation, monitoring/tracing) that assist with in-vitro serverless experiments. Such lack of automation and tooling hampers reproducibility and analysis. We demonstrate that cluster misconfiguration can lead to 100× throughput overestimation (§3).

To address the above challenges, we present a principled methodology for end-to-end in-vitro serverless experiments. Specifically, we make the following contributions:

- **Serverless experiment methodology:** We develop and validate the first serverless experiment methodology, including workload modeling from production function trace (§2.2), cluster configuration (§3.1), system warm-up (§3.2), and benchmark function implementation (§3.3), enabling in-vitro experiments with clusters of arbitrarily smaller sizes.

- **End-to-end experiment platform:** We build and validate an out-of-box serverless platform integrated with our methodology and experiment tools for in vitro serverless research (§4). Our work is open-source.

2 Workload Modeling

The goal of workload modelling is to create a production-like environment for realistic performance testing. Traces collected in production clusters capture the characteristics of real workloads, however they are often too large to replay in their entirety for most experiments, due to their cluster resource requirements [33, 47, 49, 52] and duration [20, 21, 47]. We use the trace from Azure Functions [47] as a running example. It contains 73K unique functions and spans two weeks of deployment time.

The need for sampling. For practical "in-vitro" experiments that can execute in smaller clusters and in less time, researchers need to sample large-scale traces. However, to ensure that meaningful insights can still be gleaned from down-scaled configurations, the sampling methodology must preserve salient characteristics of the full workload.

Workload sampling is particularly challenging and critical in the context of serverless computing, as workloads exhibit skewed distributions [24, 47, 48, 54]. For example, functions are typically short-lived (less than 1 second), however the execution time can vary widely across functions and individual invocations of the same function. Some functions are also invoked much more frequently than others (e.g., 1% of functions can account for over 80% of invocations [47]).

Representativeness. Serverless workloads can be mainly characterized in three dimensions: execution time, memory footprint, and load variation. Taking these as defining characteristics of serverless workloads, we call a sampled trace *representative* of an original large-scale trace if its distribution is *statistically similar* to the original trace's distribution *across all three dimensions*: (1) function execution time, (2) function memory footprint, and (3) function invocation frequency.

2.1 Problems in Existing Workload Models

Distribution Shift. Many prior works have sampled the Azure trace to evaluate state-of-the-art features in serverless systems. As shown in Figure 1, their workload models diverge from the original trace to various degrees.

FaasCache [18] and Medes [46] randomly pick unique functions from the original trace (`Random`) [18]. Additionally, Medes also scales all function invocation rates by 5× [46] (`Random+Scaling`). Hermod [24] intentionally models the load skewness by choosing one function

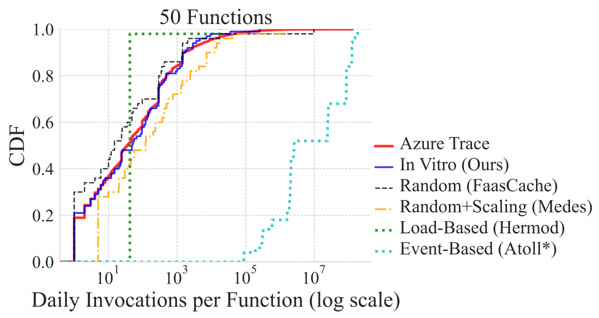


Figure 1: Distribution comparison of daily invocations per function of workload models from prior works (FaasCache [18], Medes [46], Atoll [48], and Hermod [24]) with our workload model (§2.2). For fairness, all workload models are scaled to 50 functions (see §2.1). *Atoll scales the invocations based on their testbed but does not specify the scales, so we do not do such scaling here.

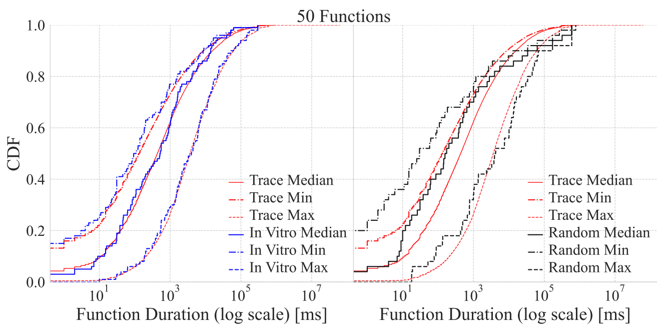


Figure 2: Distribution validation of function durations between workload samples from our method and random selection against the original Azure trace. Both samples contain 50 functions. Our method approximates well the *tail* distributions of the 78K functions of original trace with only 50 functions.

that accounts for 90% of the total load and having 49 functions that equally contribute to the remaining 10% [24] (`Load-Based`). However, since it does not specify how large the 90%-load is, we randomly select a function that has the average number of invocations (18,595) in the original trace as that 90%-load function. Atoll's workload model takes the most popular functions of every event type [48] (`Event-Based`). FaasCache originally uses 200 random functions, and Medes uses 10. Hermod selects 50 functions based on load, and Atoll does not specify the number. For fairness, we unify the number of functions to 50 as a middle ground for all workload models.

Intractable Load. Existing workload modeling methods also do not provide a way to easily control or monotonically scale the load in a trace. The ability to gradually and monotonically increase load on a serverless system is useful to find the "elbow" point, at which the system resources are saturated and queuing latency starts to manifest. The location of such an elbow point is an important

Algorithm 1 Sampling individual trace.

```
Input: $N \in \mathbb{Z}$, number of functions in resulting sample
Input: $T$, original trace of dimension $D$
Input: $\Phi$, set of statistical tests
Output: $T'$, sample trace with N functions
Parameters: $\vec{\tau}$, significance thresholds for each $\Phi$
1  $v \leftarrow$ false
2  while !$v$ do
3      $\vec{f} \leftarrow$ Draw $N$ functions uniformly at random from $T$
4      foreach $d \in D$ do
5          $\vec{r} \leftarrow$ Select $N$ records of $d$ from $T$ for $\vec{f}$
6          $v \leftarrow$ true
7          for $\phi \in \Phi$, $\tau_\phi \in \vec{\tau}$ do
8              if $\phi(\vec{r}, T^{(d)}) \leq \tau_\phi$ then
9                  $v \leftarrow$ false; break
10         if $v$ then
11             $T'^{(d)} \leftarrow \vec{r}$
12         else
13             $T' \leftarrow []$; break
14 return $T'$
```

performance indicator when evaluating resource allocation and scheduling policies [11, 13, 17, 23, 38, 44]. However, with random sampling, the load in each sample trace is uncontrolled by the sampling process, which makes it infeasible for load-testing. For example, a sample trace of 300 functions resulting from Random Selection can have 6× smaller load than a sample trace containing 40 less functions (Figure 3 left).

2.2 In Vitro Trace Sampling and Scaling

Assumptions. We use the Azure Functions trace in our work and make the following assumptions, due to limitations of the trace: (1) we discard any incomplete or inconsistent records and treat duplicated functions as different functions [1], (2) we assume exponential inter-arrival times since the traces does not contain invocation timestamps, and (3) we assume functions are independent [2].

Sampling algorithm. We propose a simple sampling algorithm aided by statistical tests (Alg. 1). The algorithm first draws functions ($\vec{f}$) uniformly at random, then for each function, it indexes into the original trace to obtain various dimensions (e.g., memory, invocations) one by one. During this process, it conducts a set of statistical tests before moving on to the next dimension, checking if the significance of the similarity between the sampled dimension ($\vec{r}$) and that of the original workload is above the threshold (Line 8). Specifically, we employ two statistical tests, Kolmogorov-Smirnov two-sample test [37] and

[1]github.com/Azure/AzurePublicDataset/issues/23

[2]github.com/Azure/AzurePublicDataset/issues/16

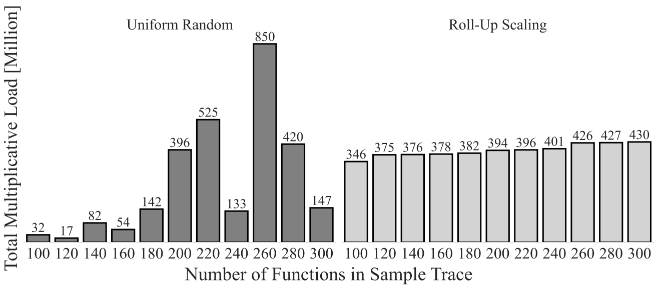


Figure 3: Comparison of the multiplicative load of sample traces with increasing numbers of functions resulting from Random Selection and our Roll-Up Sampling method. The multiplicative load is defined as the multiplication of function duration in ms and memory footprint in MiB.

Anderson-Darling k-sample test [9]. The algorithm will retry from the start unless *all* dimensions pass both statistical tests [3] (empirically, this procedure is guaranteed to terminate when sample size ≥10 functions). As a result, using only 50 functions, our sampling method approximates the (tail) distributions well across all dimensions of the original trace (Fig. 1 & 2 left).

Although random selection has the closest performance to our approach, it does not guarantee sound approximation across *all* dimensions of the original trace, such as function execution time (Fig. 2 right). Such distribution shifts make these workload models unlikely to be seen in production, which can negatively impact the applicability of resulting serverless policies. Furthermore, since the statistical similarity is guaranteed by statistical tests, trace samples from *independent* draws contain similar load and characteristics (as they all approximate the original distributions). As a result, the performance measurements of independent trace samples from our sampling approach would yield much less variation compared to straightforward random selection, providing more representative results and better convergence behaviors (Fig. 4) [26, 30, 55]. For example Fig. 4 shows the variation of system performance over 100 runs on 100 independent sample traces from resulting from Random Selection and our Roll-Up Sampling method. The performance measurements from our sample traces yield 15.5× lower standard deviation, compared to random selection.

Roll-up scaling. While drawing larger samples ($N \uparrow$) naturally improves the fidelity of sampled job distributions [30] (Fig. 7b red line) due to Law of Large Numbers, it does not provide any control over the load of resulting samples (Fig. 3 left). One way of controlling the load is by using the number of functions N as a proxy, i.e., a larger

[3]The Azure trace contains three dimensions (function invocations, execution times, and memory allocations), and here we only compare the invocation distribution because many prior works employ the other two dimensions from other traces/benchmarks.

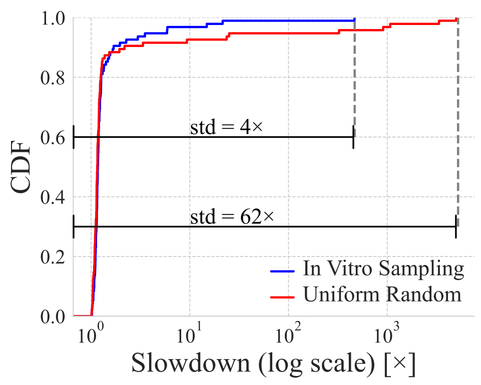


Figure 4: Standard deviation (`Std`) of slowdown resulting from 100 runs on 100 *independent* trace samples from our sampling method (`Ours`) and that of the random selection (`Uniform Random`). Slowdown is defined as the response time over the specified duration from the Azure trace.

sample trace that contains more functions is expected to bear higher load than a smaller one having less functions. There are two options to do so, (1) drilling down: drawing smaller sub-samples from a large sample trace, and (2) rolling up: combining small trace samples to form larger ones that contain more functions. Drilling down is infeasible since sampling errors cascade from larger samples to smaller ones, making them invalid. In contrast, rolling up is an analogy of bootstrapping process [41], where the sample fidelity does not degrade with the sample size, because the smallest trace samples (`Unit Sample`) at the bottom layer are independently drawn from the original trace (Fig. 7a). As a result, we use roll-up scaling to obtaining series of trace samples where the load grows monotonically with the number of functions (Fig. 3 right). Moreover, with this scaling method, the statistical similarity converges stably whereas that of Uniform Random fluctuates greatly (Fig. 7b). Roll-up scaling offers a way to control the load through the number of functions a sample trace contains, enabling load-testing on real workloads (Fig. 8).

3 In Vitro Serverless Clusters

Having a valid workload model does not guarantee obtaining desired measurements without appropriate cluster configuration (§3.1), warm-up procedure (§3.2), and function implementation (§3.3).

3.1 Cluster Configuration

Systems experiments often require multiple runs and frequent restarts to reset system states for obtaining stable and consistent results [26,55]. However, correctly setting up and configuring serverless clusters is a non-trivial task. Existing open-source serverless platforms require onerous manual setup and tuning to execute representative workload traces. This complexity impedes results reproduction since the cluster boosting procedure is prune to misconfigurations and may vary by practitioner. Therefore, fully automated cluster setup is required to mitigate such pitfalls. Furthermore, required resources (CPU, memory) from the trace should be reserved for every deployed function as what is done in most public clouds [3,4]. Guaranteed resources are reserved exclusively for respective jobs even if they are not fully utilized [14,26,31,49]. Without reserving resources for each function (e.g., CPU and memory quotas), system performance can have much larger variation due to more dynamics in sharing of resources between functions that should have been designated to the corresponding tenants. Similarly, it can also result in overestimation of the maximum load the system can sustain because of unrealistically high resource utilization. To provide such resource guarantees, during deployment serverless systems should map the maximum memory requirement of each function to its CPU quota based on common standards from public clouds (e.g., [3]) By doing so, cluster scheduler can be informed to reserve these resources exclusively for corresponding function instances, providing the same resource guarantees for each function as that of the public clouds.

3.2 Two-Phase Warm-up

Before the start of experiments, a *warm-up procedure* needs to be carried out. For performance measurements to be representative, the underlying system should be in a steady, long-running state [25,29] as if it has been running the workload in a production environment for a sufficient period of time.

Specifically, if the experiment starts at time t, each function should have approximately a certain number of instances as if the system has been running since $(t-w)$, where w is the eviction window of the system. In other words, any decisions made by the control plane before $(t-w)$ should either have taken effect or have disappeared due to eviction by time t. To achieve this, the warm-up procedure should have at least two phases. In Phase 1, it profiles the first w minutes of the trace to obtain an estimation of the expected concurrency L for each function. Next, using the profiled concurrences, it can employ Little's Law ($\mathbb{E}L = \frac{\Lambda/60}{1000/\bar{T}}$, where Λ is the total invocations, $\bar{T}$ is the mean execution time in ms of a function, and 60 and 1000 are for transforming the time unit in the trace from minute and ms to second respectively) to compute the number of warm-up instances for each function. In Phase 2, the procedure should directly bring up the numbers of instances estimated in Phase 1 for every function. Then, it starts invoking functions for the next $\geq w$ min-

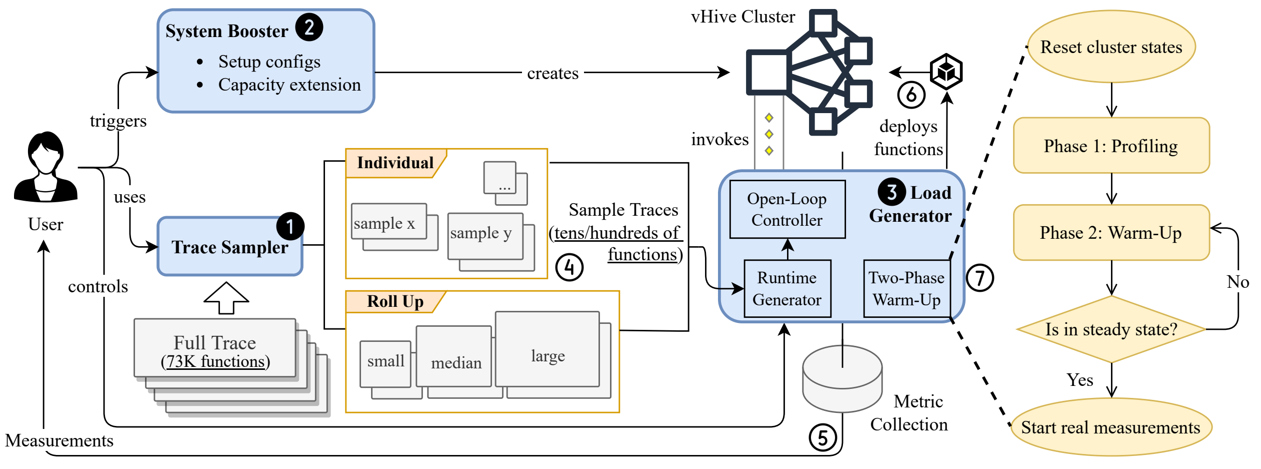


Figure 5: Overview of the end-to-end platform for in-vitro serverless experiments.

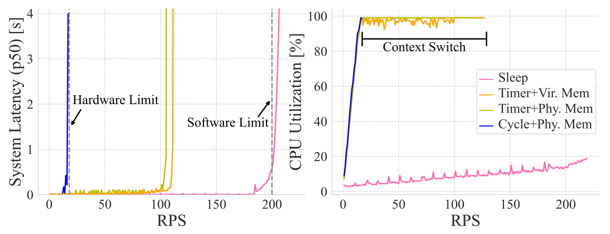


Figure 6: Validation of our function implementation by invoking one function with 1s of duration and 170 MiB of memory footprint (both are p50 of Azure trace) on one node, gradually increasing requests per second (`RPS`). `Sleep`: Idle wait. `Timer+Vir/Phy.Mem`: Timer-based timing and virtual/physical memory allocation. `Cycle+Phy.Mem`: Cycle-based timing and physical memory allocation.

utes, letting the cluster scale function instances naturally. Once the system has reached a steady state [25], the actual experiments and measurements can then begin.

3.3 Function Implementation

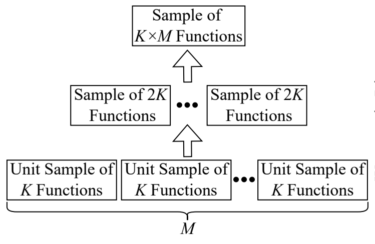
Resource utilization is the bottom line of cloud providers and a key performance metric of serverless policies. Given a workload with valid job distributions and system configurations, the actual amount of load imposed on the system relies on the runtime implementation of the function server; therein lies the question: *How to implement the functions such that the runtime specifications are reliably fulfilled?*

Idle waiting (e.g., `sleep`) and busy spinning stopping on timer-events are two commonly used ways to fulfil function durations from the trace [4]. However, we show that these methods may yield misleading. To demonstrate this implication, we invoke one function with 1s of duration and 170 MiB of memory footprint (both are p50 of Azure trace) on one node, gradually increasing requests per second results (Fig. 6). Since the node has 16 physical cores with both turboboost and hyperthreading disabled, the max. theoretical throughput is ~16 RPS (Hardware Limit), i.e., the system can serve ~16 concurrent requests. We also impose the same software limit as Azure does [39]: max. 200 instances/function (Software Limit). The max. throughput of our implementation (cycle-based timing + physical memory allocation) is limited by the number of cores, but throughput of the idle implementation (`Sleep`) is only capped by the instance limit. The timer-based implementations with physical/virtual memory allocation saturate the CPU at the same point as ours but end up measuring the efficiency of context switching afterwards (since the time spent context switching is also part of the timing), *instead of the serverless policies under evaluation*. Consequently, wrong implementation can lead to 100× throughput overestimation and 5× underestimation of resource utilization.

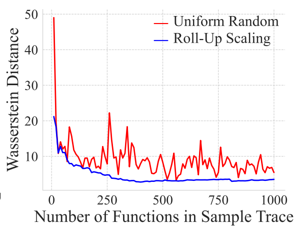
Firstly, letting a function idle wait barely imposes any load on the underlying serverless system as the job is put on runqueue and woken up by interrupts. For timer-based implications, function executions are amenable to kernel traps (e.g., context switches, I/O blocking). For example, under high CPU utilization (e.g., $N+1$ kernel threads running on an N-core machine), a 1s-duration function instance can spend 300ms on context switching and only 700ms on execution. In other words, although the function was stopped by the timer after exactly 1s, it only imposed 700ms-load on the system (rather than the intended 1s of load).

Therefore, timer-based implementation can give mis-

[4] Here, we are talking about loaded functions not unloaded functions implemented with `noop` that return immediately upon any requests


Sample of $K{\times}M$ Functions
Sample of $2K$ Functions
Sample of $2K$ Functions
Unit Sample of K Functions
Unit Sample of K Functions
Unit Sample of K Functions
M

(a)



(b)

Figure 7: (a) Roll-up scaling method. (b) Divergence of memory distribution of trace samples from the original Azure workload measured by Wasserstein distance. The divergence of the roll-up samples converges steadily as the number of functions in the sample trace increases, while that of the random samples fluctuates greatly.

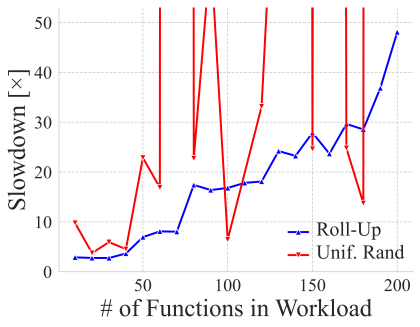


Figure 8: System performance with sample traces on a three-node cluster (one master, two workers). Roll-up scaling offers a way to control the amount of load in trace samples in the workload modelling process (§2.2). Specifically, it makes the load increase monotonically with the number of functions in sample traces, enabling load testing with real workloads. In contrast, since the load in random selection is intractable (§2.1), system performance fluctuates greatly in an uncontrolled manner.

leading results and even lead to evaluating the wrong target (e.g., the capability of context switching), which makes policy evaluation and comparison hard (Fig. 6). Moreover, it also bears high randomness (e.g., a 1s-duration function can impose varying amounts of load in the different runs) and, in turn, implies *irreproducibility*.

Similarly, replaying function memory footprints requires physical memory allocation. For example, invoking a system call (e.g., `mmap`) to replay a function's memory footprint from the trace without touching any allocated pages is problematic on Linux, because the allocated memory pages remain virtual if not used, imposing less stress on the underlying system than intended.

4 End-to-End Experiment Platform

To realize our methodology and to make serverless experiments easier and more reproducible, we build an end-to-end (E2E) platform atop one of the state-of-the-art serverless framework, vHive [50]. We choose vHive for its generality, ease of integration of various sandboxes (e.g., gVisor [53], Firecracker [7]), and other advanced features. It offers a convenient interface to the serverless infrastructure but does not provide tools to assist our end-to-end methodology such as workload modeling, load-testing. Thus, apart from a tuned vHive framework, we build three main components for our experiment platform: Trace Sampler (❶), System Booster (❷), and Load Generator (❸) [5].

Obtaining sample traces. Trace Sampler implements our sampling and scaling methods described in §2.2 with <1K LoC. Its generality makes it easy to be adapted to other traces of different data models. Users can run Trace Sampler by with simple commands to obtain a set of sample traces (④), where *every sample trace* is a statistically sound approximation of the entire Azure workload (Fig. 1 & 2 left), and the load increases gradually with the number of functions (Fig. 3 right).

Automated setup & configuration. With a single command, users can use System Booster to create an in vitro cluster of arbitrary sizes where related parameters (e.g., networking, resource quotas) are automatically configured for hosting heavy workloads , making our open-source platform the *first* to support 500+ function instances per node, on par with leading public clouds. In turn, System Booster makes conducting, repeating, and reproducing experiments much easier and more efficient, mitigating complexity and pitfalls described in §3.1.

Open-loop load generation. Load Generator is the spinal cord of our end-to-end platform and embodies the bulk of our methodology. It employs an open-loop controller, which is more intricate but also more useful than close-loop controller in evaluating system stability and latency [29,55]. Given statistics provided by the Azure trace, it generates function execution times and memory footprints using Smirnov Transform [19]. Specifically, Load Generator treats a function runtime (e.g., CPU, memory) as a random variable, say Y with a cumulative density distribution (CDF) F_Y. Then, Load Generator approximates the CDF simply through probability integral transform: it first generates random variable $U \sim \mathcal{U}(0,1)$ and draws the runtime by $F_Y^{-1}(u) = y$ for $\mathbb{P}(0 < Y < y) = \mathbb{P}(0 < U < u)$. As for generating inter-arrival times (IATs), since this information is missing from the published Azure trace, Load Generator provides users with several common IAT distributions, e.g., Exponential, Uniform, periodic.

[5] Code will be available upon publication.

Load Generator replays function traces systematically, generating load against vHive. Before starting any measurements, it executes a two-phase warm-up process (§3.1, ⑦). The eviction period (w) in vHive is 5 minutes, which is used as the profiling duration. In Phase 2, Load Generator generates warm-up invocations for the next 10 minutes ($2w$). To guard against insufficient warm-up, we take inspiration from Lancet [29] and adopt augmented Dickey-Fuller tests (ADF) for testing the stationarity of the latency measurements (⑤). If the system has not yet reached in a steady state [25] after 10 minutes, users will receive a warning to increase this duration. However, unlike Lancet that checks the stationarity of E2E response time, we use pure system latency, (E2E response time with execution times subtracted) as the stationary measure (see Appendix A.1 for detailed explanation and proof).

Cycle-based timing & physical memory allocation. To address the pitfalls discussed in 3.3, we need to circumvent timer-based implementation, directly gauging the amount of work, instead of the time of existence, for each function. To achieve this objective, we resort to cycle-based timing (⑥)—before warmup, Load Generator benchmarks the number of cycles a *timing unit* can complete in a function sandbox (e.g., container, μVM) for a given testbed. As a result, each function knows how much actual work it has done by counting cycles instead of relying on timers. This timing mechanism is immune to kernel traps (§3.3) as the internal counter of each function only measures the work done. To make the timing unit more dependent on the hardware rather than the software (e.g., the compiler, the version of vHive), we use an x86 instruction `sqrtsd` commonly available in modern processors [5]. Also, this instruction is commonly backed by one port and a single execution unit (e.g., `DIVIDE`), which intrinsically lessens the timing variability caused by deep pipelines [6]. As a result, our cycle-based implementation is precisely bounded by the hardware capacity and obviates measuring other factors, e.g., context switching (Fig. 6). In terms of memory allocation, Load Generator evenly distributes the memory footprint of an Azure application [47] to its functions (akin to [18]. It requests memory also via `mmap` but *touches 50% of allocated memory pages* (not all published memory footprints correspond to physical memory [10]).

5 Characterization & Future Work

Our methodology enables reliable, analytical reasoning of performance of serverless policies. Being able to sweep real workloads is one of such use cases. Another is the quantitative characterization of serverless-specific features. For instance, cycle-based function implementation enables researchers to analytically reason about system performance as functions get shorter. With vHive, as we

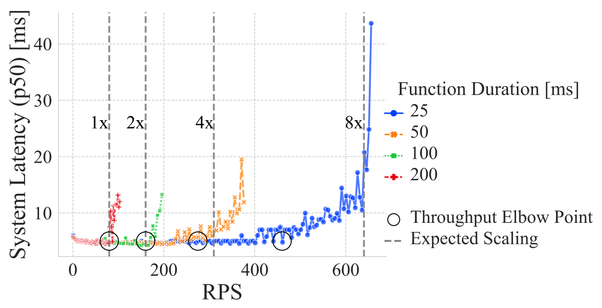


Figure 9: Latency measurements (p50) of functions with various durations. The elbow points gradually lag behind the expected throughput scaling, indicating larger fractions of system overheads starting to dominate when functions get shorter.

reduce the function duration from 200 ms to 100 ms and then to 50 ms, we obtain approximately 2× and ∼4× the throughput respectively (Fig. 9). However, such expected scaling stops when function duration is reduced further, e.g., from 50 ms to 25 ms, due to the increasing fraction of system overheads, such as network virtualization, remote procedure call. This system tax induces about 5 ms of base latency, which is instrumental to be further broken down.

6 Conclusion

This paper proposes an end-to-end methodology for serverless experiments. It addresses fundamental, methodological problems, including workload modeling, cluster configuration, system warm-up, implementation of benchmark function, and load generation. We realize our methodology by building an out-of-box serverless experiment platform that produces representative workloads, enables in vitro experiments with clusters of arbitrarily smaller sizes, and makes performance testing more reliable and reproducible.

References

[1] Openwhisk `https://openwhisk.apache.org/`, 2016.

[2] Fn `https://fnproject.io/`, 2022.

[3] Google cloud function quota `https://cloud.google.com/functions/pricing`, 2022.

[4] Lambda quotas `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`, 2022.

[5] ABEL, A., AND REINEKE, J. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS* (New York, NY, USA, 2019), ASPLOS ’19, ACM, pp. 673–686.

[6] ABEL, A., AND REINEKE, J. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *ICS ’22: 2022 International Conference on Supercomputing, Virtual Event, USA, June 27-30, 2022* (June 2022), L. Rauchwerger,

K. Cameron, D. S. Nikolopoulos, and D. Pnevmatikatos, Eds., ICS '22, ACM, pp. 1–12.

[7] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)* (2020), pp. 419–434.

[8] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)* (2018), pp. 923–935.

[9] Anderson, T., and Darling, D. A test of goodness of fit. *Journal of the American Statistical Association 49* (1954), 765–769.

[10] Azure. Azure functions dataset 2019 trace analysis `https://github.com/Azure/AzurePublicDataset/blob/master/analysis/AzureFunctionsDataset2019-Trace_Analysis.md`, Jun 2020.

[11] Cai, X., Wu, X., Zhou, X., et al. *Optimal stochastic scheduling*, vol. 5. Springer, 2014.

[12] Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., and Hoefler, T. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference* (2021), pp. 64–78.

[13] Delimitrou, C., and Kozyrakis, C. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)* (2013), IEEE, pp. 23–33.

[14] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices 49*, 4 (2014), 127–144.

[15] Demoulin, H. M., Fried, J., Pedisich, I., Kogias, M., Loo, B. T., Phan, L. T. X., and Zhang, I. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 621–637.

[16] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).

[17] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 281–297.

[18] Fuerst, A., and Sharma, P. Faascache: keeping serverless computing alive with greedy-dual caching. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021).

[19] Gentle, J. E. *Random number generation and Monte Carlo methods*, vol. 381. Springer, 2003.

[20] Google cluster-usage traces. `https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md`.

[21] Google cluster workload traces. `https://research.google/tools/datasets/google-cluster-workload-traces-2019/`.

[22] Granger, C. W. J., and Joyeux, R. An introduction to long memory time series models and fractional differencing. *Journal of Time Series Analysis 1* (1980), 15–29.

[23] Harchol-Balter, M. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[24] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Practical scheduling for real-world serverless computing. *ArXiv abs/2111.07226* (2021).

[25] Kasan, H., Kim, G., Yi, Y., and Kim, J. Dynamic global adaptive routing in high-radix networks. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (2022), pp. 771–783.

[26] Kasture, H., and Sanchez, D. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)* (2016), IEEE, pp. 1–10.

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

[28] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 427–444.

[29] Kogias, M., Mallon, S., and Bugnion, E. Lancet: A self-correcting latency measuring tool. In *USENIX Annual Technical Conference* (2019).

[30] Lee, J., Kim, C., Lin, K., Cheng, L., Govindaraju, R., and Kim, J. Wsmeter: A performance evaluation methodology for google's production warehouse-scale computers. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), pp. 549–563.

[31] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), pp. 450–462.

[32] Ltd, O. Openfaas `https://www.openfaas.com/`, 2022.

[33] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., and Xu, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 412–426.

[34] Magalhães, D., Calheiros, R. N., Buyya, R., and Gomes, D. G. Workload modeling for resource usage analysis and simulation in cloud computing. *Computers & Electrical Engineering 47* (2015), 69–81.

[35] Mahgoub, A., Wang, L., Shankar, K., Zhang, Y., Tian, H., Mitra, S., Peng, Y., Wang, H., Klimovic, A., Yang, H., et al. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 285–301.

[36] Maissen, P., Felber, P., Kropf, P., and Schiavoni, V. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems* (2020), pp. 73–84.

[37] Massey, F. J. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association 46* (1951), 68–78.

[38] McClure, S., Ousterhout, A., Shenker, S., and Ratnasamy, S. Efficient scheduling policies for {Microsecond-Scale} tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 1–18.

[39] MICROSOFT. Event-driven scaling in azure functions `https://docs.microsoft.com/en-us/azure/azure-functions/event-driven-scaling`, Jun 2022.

[40] MITTAL, V., QI, S., BHATTACHARYA, R., LYU, X., LI, J., KULKARNI, S. G., LI, D., HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. *Proceedings of the ACM Symposium on Cloud Computing* (2021).

[41] MOONEY, C. Z., MOONEY, C. F., MOONEY, C. L., DUVAL, R. D., AND DUVALL, R. *Bootstrapping: A nonparametric approach to statistical inference*. No. 95. sage, 1993.

[42] MORENO, I. S., GARRAGHAN, P., TOWNEND, P., AND XU, J. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing 2*, 2 (2014), 208–221.

[43] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 57–70.

[44] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 361–378.

[45] ROY, R. B., PATEL, T., AND TIWARI, D. Icebreaker: warming serverless functions better with heterogeneity. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022).

[46] SAXENA, D., JI, T., SINGHVI, A., KHALID, J., AND AKELLA, A. Memory deduplication for serverless computing with medes. *Proceedings of the Seventeenth European Conference on Computer Systems* (2022).

[47] SHAHRAD, M., FONSECA, R., GOIRI, Í., CHAUDHRY, G. I., 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 *USENIX Annual Technical Conference* (2020).

[48] 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 ACM Symposium on Cloud Computing* (2021), pp. 138–152.

[49] TIRMAZI, M., BARKER, A., DENG, N., HAQUE, M. E., QIN, Z. G., HAND, S., HARCHOL-BALTER, M., AND WILKES, J. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems* (2020), pp. 1–14.

[50] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)* (2021), ACM.

[51] VERMA, A., KORUPOLU, M. R., AND WILKES, J. Evaluating job packing in warehouse-scale computing. *2014 IEEE International Conference on Cluster Computing (CLUSTER)* (2014), 48–56.

[52] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. *Proceedings of the Tenth European Conference on Computer Systems* (2015).

[53] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[54] ZHANG, Y., GOIRI, Í., CHAUDHRY, G. I., FONSECA, R., ELNIKETY, S., DELIMITROU, C., AND BIANCHINI, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 724–739.

[55] ZHANG, Y., MEISNER, D., MARS, J., AND TANG, L. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), IEEE, pp. 456–468.

A Appendix

A.1 Stationarity Test on System Overheads

To develop and evaluate scheduling policies, we take special interests in the long-running behaviors of serverless systems. To this end, the system needs to be first warmed up to a stable state like the systems in production, before starting any actual measurements. This requirement gives rise to one question: *How to systematically determine when the system has been sufficiently warmed up?* Too long a warm-up would waste time at each experiment run whereas if it is too short, insufficient warm-up would result in a host of problems (e.g., fluctuating measurements with increasing/decreasing trends).

Serverless systems are queuing machines. When the arrival rate is roughly the same as the service rate, the system can achieve a relative equilibrium, i.e., a stable, long-running state [25, 29]. To tackle this challenge, we take inspiration from Lancet [29] where observed end-to-end (E2E) latencies (response times) are used for determining the system state. This approach is intuitive since latency directly reflects the queuing state of the system, i.e., a decreasing trend in latencies imply that the system is still warming up, and when the system is overloading, the latencies would exhibit clear increasing trends. To capture such trending, Lancet employs augmented Dickey-Fuller tests (ADF) for testing the stationarity of time series data by checking whether it has a unit root. Stationarity guarantees that the mean of time series measurements is stable, and the variance does not have clear trends either. It is the premise of most predictive time series models such as ARIMA used in [47].

However, our approach is different from Lancet—we do not directly use E2E latencies for checking stationarity. This is because, unlike Lancet that generates completely synthetic loads, Load Generator generates real-world workloads that commonly exhibit periodicity [45]. Periodic data with fixed recurrent patterns is not stationary, which can *mislead* the ADF test. Instead, we check the pure system overheads obtained by subtracting the execution time from E2E latencies. Pure system overheads are induced by various kinds of system taxes such as, network latency, queuing delay, container operation time, etc. These overheads of long-running services in a stable production environment are stationary (i.e., without drastically increasing or decreasing trends). To reason about it formally, without lose of generality, we model E2E latencies using a common autoregressive–moving-average (ARMA) model with p lags and q error-lags:

$$l^{(t)} = \alpha_0 + \sum_{\tau=1}^{p} \alpha_\tau \cdot l^{(t-\tau)} + \epsilon^{(t)} + \sum_{k=1}^{q} \beta_k \cdot \epsilon^{(t-k)} + z^{(t)}, \quad (1)$$

where $l^{(t)}$ is the E2E latency measure and $\epsilon^{(t)} \sim \mathcal{N}(0, \sigma^2)$ is the error term, and $l^{(t)}$ is the corresponding execution time at time t. α_i and β_i denote the constant coefficients. Now, let $y^{(t)}$ be the pure system overhead where $y^{(t)} \leftarrow l^{(t)} - z^{(t)}$ and $\mathbb{E}y = c$. Then, Eq. 1 can be rewritten as:

$$\begin{aligned} l^{(t)} &= \alpha_0 + \sum_{\tau=1}^{p} \alpha_\tau \cdot \left(y^{(t-\tau)} + z^{(t-\tau)}\right) + \epsilon^{(t)} \\ &\quad + \sum_{k=1}^{q} \beta_k \cdot \epsilon^{(t-k)} + z^{(t)} \\ &= \alpha_0 + \vec{\alpha}_p \cdot \vec{y}_p + \vec{\alpha}_p \cdot \vec{z}_p + \epsilon^{(t)} + \vec{\beta}_q \cdot \vec{\epsilon}_q + z^{(t)} . \quad (2) \end{aligned}$$

Then, we derive Eq. 3 by taking the expectation of Eq. 2:

$$\begin{aligned} \mathbb{E}[l^{(t)}] &= \mathbb{E}\alpha_0 + \mathbb{E}\left[\vec{\alpha}_p \cdot \vec{y}_p\right] + \mathbb{E}\left[\vec{\alpha}_p \cdot \vec{z}_p\right] + \mathbb{E}\epsilon^{(t)} \\ &\quad + \mathbb{E}\left[\vec{\beta}_q \cdot \vec{\epsilon}_q\right] + \mathbb{E}z^{(t)} \\ &= \alpha_0 + \vec{\alpha}_p \cdot \vec{c} + \vec{\alpha}_p \cdot \mathbb{E}\vec{z}_p + \mathbb{E}z^{(t)} , \quad (3) \end{aligned}$$

where only the last two terms that depend on the execution time z are not constant. In turn, we thereby proved that, when hosting real-world traces, it is important to subtract the execution time from the E2E latencies in order to prevent the ADF test from being biased by the measurements. For completeness, another way of testing stationarity with E2E latencies is via differencing [22] (e.g., ARIMA). However, this method requires fitting the periodicity to obtain the differencing lag value for every type of workloads, which is not as feasible since the sample traces vary in function composition.