FaaSRail: Employing Real Workloads to Generate Representative Load for Serverless Research

Christos Katsakioris
National Technical University of Athens
Athens, Greece
ckatsak@cslab.ece.ntua.gr

Chloe Alverti
University of Illinois Urbana-Champaign
Illinois, USA
xalverti@illinois.edu

Konstantinos Nikas
National Technical University of Athens
Athens, Greece
knikas@cslab.ece.ntua.gr

Dimitrios Siakavaras
National Technical University of Athens
Athens, Greece
jimsiak@cslab.ece.ntua.gr

Stratos Psomadakis
National Technical University of Athens
Athens, Greece
psomas@cslab.ece.ntua.gr

Nectarios Koziris
National Technical University of Athens
Athens, Greece
nkoziris@cslab.ece.ntua.gr

ABSTRACT

With the proliferation of Serverless Computing, the Function-as-a-Service (FaaS) paradigm is nowadays ubiquitous. As a result, the domain has attracted extensive research, both in industry and academia, identifying opportunities and addressing limitations across all aspects of this new Cloud paradigm. Recently, FaaS providers have released production workload traces of their commercial platforms. These expose important characteristics, such as the execution time of function invocations, their number and the distribution of their inter-arrival times, which must be taken into account for a concrete evaluation of innovative solutions. Nevertheless, the Serverless ecosystem still lacks a unified evaluation methodology based on such information.

In this paper we attempt to fill this gap, by developing a methodology for fitting existing, real, open-source workloads found in FaaS benchmarking suites to production FaaS workload traces, in a way that sufficiently preserves the aforementioned core statistical properties of such traces. Based on this, we build FaaSRail, an open-source load generator that receives a target maximum request rate and a target total execution duration as inputs from the user and generates representative, scaled down FaaS load.

CCS CONCEPTS

• **General and reference** → **Evaluation**; **Experimentation**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Datacenter; Cloud; Serverless; FaaS; Load Generation; Benchmarking; Open Source

ACM Reference Format:
Christos Katsakioris, Chloe Alverti, Konstantinos Nikas, Dimitrios Siakavaras, Stratos Psomadakis, and Nectarios Koziris. 2024. FaaSRail: Employing Real Workloads to Generate Representative Load for Serverless Research. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658684

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06
https://doi.org/10.1145/3625549.3658684

1 INTRODUCTION

Serverless computing is nowadays supported by all major cloud providers [1–6] and its adoption is on the rise [7] as this new programming and deployment paradigm approaches maturity. Function-as-a-Service (FaaS) allows users to upload their application code split into multiple stateless execution units (functions) that are scheduled upon certain events (e.g., HTTP requests). In turn, cloud providers (i) execute the user-defined functions inside secure sandboxes (e.g., containers and/or microVMs [8]), (ii) transparently autoscale the compute and memory resources to meet request load, and (iii) bill users in a fine-grained pay-as-you-go manner [1–6]. This model relieves users from the burden of handling infrastructure and provides cloud vendors with a unique opportunity to autonomously scale hardware resources and hence reassess resource over-provisioning [9].

However, providers face numerous challenges with respect to resource allocation [7, 9–16], scheduling, load balancing [12, 15, 17–23] and execution overheads across the stack [8, 16, 24–38] when they deploy serverless functions in their clusters. For instance, the short execution times of functions make them very sensitive to instantiation overheads (*cold starts*), hence providers keep them cached even when idling, effectively wasting memory [14, 39–41]. *A long line of recent research targets these challenges and its wide scope underlines the necessity for a unified representative methodology for FaaS experimentation.* In more detail, designing innovative solutions across the FaaS stack requires: (i) *open-source platforms* for FaaS deployment in small-scale local clusters, (ii) *representative workloads* for benchmarking, and (iii) *request generators for representative load.* To that end, open-source FaaS platforms (e.g., vHive [31], OpenWhisk [42], etc.) and benchmarking suites (e.g., FunctionBench [43, 44], ServerlessBench [45], etc.) have been released, but *the generation of representative load remains an open challenge.*

FaaS providers have released workload traces of their commercial platforms [39, 40], which outline the load on huge production clusters. They provide per-function information, such as execution runtimes, memory usage, and invocation request inter-arrival times. These traces expose several unique characteristics of FaaS, like sub-second execution durations and bursty request patterns, and effectively constitute the only available guide for representative load generation. Although the knowledge of the state of production-scale clusters is indispensable, this information cannot be easily utilized for research purposes, due to two main reasons. First, these traces are massive,

as they report statistics for tens of thousands of functions that are triggered billions of times and deployed on hundreds of nodes over several days. Second, the actual code and/or executable binaries of those functions are not provided due to anonymity reasons [39], making it essentially impossible to reproduce them.

To facilitate research on the FaaS stack, it is necessary to come up with easily reproducible and configurable workloads, which at the same time are representative of the production-level traces. *Currently, most of the common practices to generate load for FaaS platforms scale down the number of functions and requests in a way that fails to preserve the core statistical properties of the available trace.* Multiple works isolate trends present in the traces and emulate them synthetically [10, 12–15, 17, 18, 20, 26–28, 31, 32, 38, 46–48]. For instance, several works [10, 12–15, 17, 18, 20, 26, 28, 46, 47] generate requests for a subset of real workloads employing Poisson processes to emulate the burstiness of invocation request arrivals reported in production traces. However, as we show, this strategy violates other properties, such as the distribution of the invocation runtimes and the popularity of functions. Furthermore, it fails to capture both the per-function and the aggregated request load which usually varies, as reported in the trace.

On the other hand, multiple works use data directly from the industry traces [11, 15, 16, 20, 21, 23, 28, 29, 34–36, 47]. Typically, they scale down the experiment by randomly sampling the trace to collect a subset of the reported functions [11, 16, 23, 34–36, 48]. Subsequently, they map them to real workloads and generate requests using their reported inter-arrival times. However, as we show, random sampling can also violate statistical properties and in addition, it can lead to inconsistent experimentation [49].

Using partially representative load can potentially lead to incomplete conclusions or limit research. For example, a load that emulates bursty request rates but neglects the skewed popularity of FaaS functions, may mislead research on resource allocation and load balancing. Moreover, a load that follows non-representative skewed runtime distributions and, as a result, comprises mostly FaaS functions with sub-second execution durations, can overestimate the cold-start overheads of a realistic load and lead biased research on function caching.

Finally, a long line of research does not use real workloads at all and generates noop pseudo-functions (e.g., busy loops) instead [18, 19, 34, 48, 49]. This enables the fabrication of a larger number of functions, compared to the number of available workloads, with artificially varying execution durations. A combination with any request generator described above allows for a better approach of the execution time distributions reported in the traces. It is favored for cluster-level scheduling research [18, 19] and generally system simulation [34]. However, as FaaS research matures, the workload characteristics that go beyond average runtimes and memory utilisation, such as memory access patterns, I/O and CPU activity, input data etc., are leveraged across the stack, even at the cluster-level, to make decisions [10, 21, 22, 38]. Such characteristics are harder to realistically synthesize, which culminates the importance of real workloads in Serverless research, as well as the lack thereof.

In general, the Serverless ecosystem lacks a unified evaluation methodology and an established set of tools capable of generating real workload on research platforms of any scale, while preserving the characteristics unveiled by production FaaS traces. To address this gap, we first study the requirements of existing FaaS research areas and identify the following critical trace properties: (i) the distribution of execution durations of workloads and invocations, (ii) the skewed popularity of the workloads, and (iii) the varying load of requests. Moreover, we consider important the use of real workloads to generate FaaS load.

We develop FaaSRail, a tool that fits existing open-source workloads to production FaaS workload traces. FaaSRail's methodology includes: (i) reducing the number of functions in a trace in a statistically safe manner, with care not to brush off particular attributes which effectively shape the peculiarities of production FaaS deployment; (ii) mapping the functions of a real, industrial trace to realistic workloads found in open-source benchmarking suites, after augmenting them without distorting the execution characteristics of the original trace; (iii) downscaling the number of invocations per function, as reported in production traces, while retaining their function popularity; (iv) downsampling production traces over time, to accurately emulate their diurnal load variations in reasonable total duration; (v) emulating sub-minute invocation request burstiness, while preserving both the overall and the per-function invocation rate trends of production traces. FaaSRail is implemented as two open-source[1] components: (i) the offline *"shrink ray"*, which appropriately downscales and matches both the benchmarking suites and the production traces to generate experiment specifications, and (ii) the high-performant, versatile *load generator* that can further parameterize and replay such specifications against a backend FaaS system.

In summary, this paper makes the following contributions:

- The FaaSRail methodology and open-source utilities to generate replayable traces of requests for real FaaS workloads with similar core statistical properties as the industrial traces.
- An extensive evaluation of FaaSRail-generated traces against the Azure and Huawei traces.

2 MOTIVATION

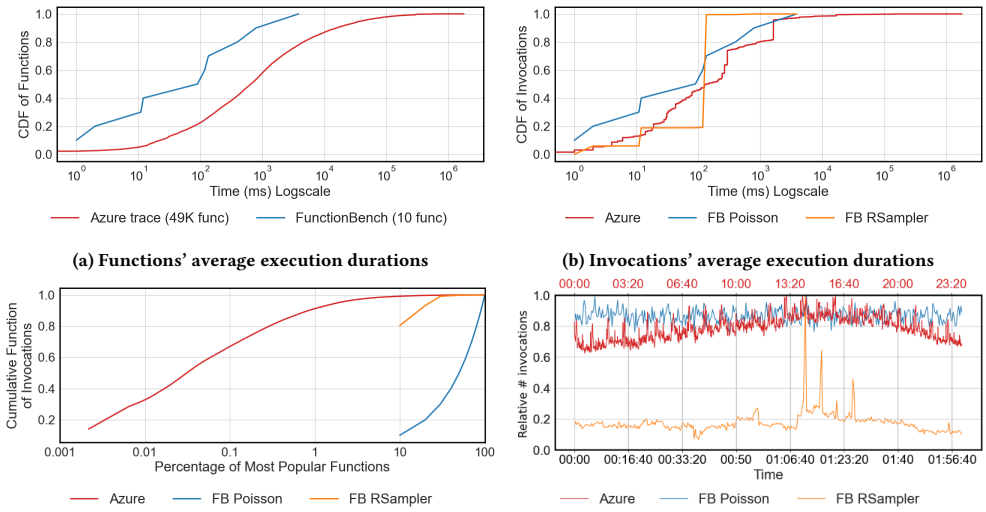
In this section we discuss the available traces of commercial serverless platforms and break down some of their key statistical properties that drive FaaS research. We then identify the challenges of leveraging these traces to generate load for FaaS experimentation and study the approaches of prior works to the problem at hand. We use our observations as motivation and guide to design FaaSRail.

2.1 Industry traces

Both Microsoft Azure [39] and Huawei [40] have released production-scale traces related to their FaaS deployments in the Cloud. Therefore, they effectively act as the main interface between industry and academia to drive related research.

The workload trace of Azure Functions reports statistics from the deployment of 80K functions recorded in a 14-days window. The statistics contain, among others, the average warm execution time for each function, as well as its total number of invocations, along with their distribution among the 1440 minutes of each day. Moreover, the trace reports the allocated and resident memory per

[1] Source code is available at https://github.com/cslab-ntua/faasrail.



(a) Functions' average execution durations

(b) Invocations' average execution durations

(c) Functions' popularity

(d) Load over time (aggregated invocation rates)

Figure 1: Load generation for serverless research. A common practice is to emulate some of the public traces' key statistical properties, such as bursty invocation request rates (Figure 1d), by generating the series of requests over time using some well known mathematical model (typically the Poisson process). Nevertheless, if not used appropriately, such approaches can regularly allow violations of other key statistical properties (Figures 1b, 1a).

application (i.e., group of functions). The trace has shed some light on several particularly interesting facts, such as:

(i) a large number of functions are very short-running; 50% of the functions execute for less than 1s;
(ii) the popularity of functions is extremely skewed; 99% of all invocations during the first day refer to merely 8% of all functions;
(iii) the most popular functions have short execution durations which leads to short runtimes dominating invocations; 80% of all invocations actually run for less than 1s;
(iv) the request rates per function are bursty, i.e., sudden spikes in requests per minute followed by idle time.

In general though, despite the aforementioned observed trends, the trace manifests huge overall variability, across all aspects of function behavior. For instance, a small portion of the functions are invoked thousands of times per minute, but 90% of the functions are invoked once per minute or less. Furthermore, the reported function execution times vary by 2 to 4 orders of magnitude. *This diversity underlines the importance of taking into consideration entire distributions to capture FaaS behavior, rather than merely average values which might be skewed.* For example, it would not be safe to overlook groups of functions based on their longer runtimes or longer idle times, as they can represent a significant portion of the total functions.

More recently, Huawei released traces for both its public-facing FaaS platform and its internal FaaS workloads. The *Huawei Public* trace has a very similar profile to Azure. The *Huawei Private* trace includes information about 200 functions monitored over 141 days running on a private cluster. This trace has more acute characteristics. Despite the smaller number of functions, the trace reports much higher invocation counts, and the functions run much faster and more frequently compared to Azure's. It also reports bursty request rates at sub-minute scale. *These differences underline the need to be capable of generating load based on various traces to model different realistic Serverless Cloud profiles.*

> **Critical statistical properties in traces**
> We consider four critical statistical properties of the production traces that capture core FaaS behavior and need to be preserved in order to have a representative load for FaaS experimentation. These are the distributions of: (i) the average execution duration of distinct functions (i.e., regardless of the number of their invocations), (ii) the popularity of functions defined as the percentage of invocations out of the total daily load, (iii) the execution durations of all invocations (similar to (i) but now functions are weighted by their number of invocations), and (iv) the arrival rates of invocations.

2.2 Relevance to research

We now discuss how the aforementioned properties drive and affect popular FaaS research areas. We seek to answer which are the most important to deliver via a load generator for serverless research.

Cold-starts. A long line of research studies cold invocations and proposes node-level [10, 16, 27, 30–32, 50, 51] and cluster-level [14, 15, 17, 18, 29, 35] techniques and policies to minimize the number and/or the overheads of cold starts. The significance of cold starts is relative to the runtime distribution of the functions [52] and their frequency is dictated by function popularity and invocation arrival rates. For instance, long running functions can amortize cold start overheads more easily [52] and less frequently, while, on the other hand, functions invoked burstily will regularly suffer from cold starts [29].

Node-level resource management. Another important area of research is node resource sharing among function invocations [10, 11,

28] and memory deduplication [16] targeting consolidation, fewer cold invocations and faster function communication [27]. Apart from the distributions related to execution times and arrival patterns, which dictate the concurrency among invocations, this line of research also requires *experimentation with real workloads* to capture CPU, memory and I/O activity.

Cluster-level policies. Multiple works [16–18, 21, 23, 28, 35] propose global schedulers to balance the load across nodes and present techniques to (pre)allocate the necessary number of instances (e.g., containers) to deliver the incoming load. Such proposals are affected equally by runtime distributions, functions popularity and arrival rates. For instance, bursty requests can lead to imbalanced scheduling and instance under-provisioning. Similarly, execution time distributions and function popularity dictate the load of nodes. While function emulation, e.g. via busy loops, has been used to simulate large clusters performance [18, 34], as FaaS research matures the characteristics of real workloads, e.g., memory and storage access, are leveraged by distributed scheduling policies [10, 21, 22, 38].

Generating load for serverless research

A load generator for FaaS experiments should provide all the critical statistical properties present in industrial FaaS traces, since most FaaS research areas are impacted by them. Moreover, it is important for the generated load to be based on real workloads.

2.3 Challenges in trace-driven load generation

Adopting the information of industrial traces to generate request load while meeting the above criteria is not straightforward. First, the traces represent experiments of massive load and scale. Both traces report request rates of hundreds of thousands or millions of invocations per minute, which probably run in clusters of hundreds of nodes [49], while they span multiple days. Second, neither the executable binaries nor the actual code of the trace functions are typically disclosed. At the same time, the number of open-source FaaS workloads available through benchmark suites is very low (i.e., in the order of 10s) to map the large number of reported functions (e.g., 80K in the case of the Azure trace).

Therefore, there is a need for a methodology to scale down the production traces, with respect to both invocations load and time, while preserving their critical statistical properties, and to generate smaller –yet representative– experiments; i.e., for small clusters, and with total duration in the order of hours or even minutes. A richer workload pool is also necessary to map the scaled down trace functions to real workloads. Using workloads exhibiting a variety of patterns (e.g., regarding I/O, CPU and/or memory usage) is deemed essential to encapsulate realistic FaaS behavior.

2.3.1 Prior approaches. The FaaS literature has not responded in a unified manner to the problem at hand. However, we do observe some common patterns among various experimental methodologies, which we move on to discuss.

Emulation. Multiple works do not use trace data directly but rather emulate some of their trends via synthetic load generators. This enables experiments of tunable load (e.g., requests per second) and arbitrary total runtime. For instance, multiple works use Poisson processes [10, 12–15, 17, 18, 20, 26, 28, 46, 47] to generate bursty requests for the workloads of a FaaS benchmark suite. Figure 1 shows the statistical properties of such a trace for FunctionBench [43, 44], configured to run for 2 hours with 144K invocations in total. Figure 1d shows how a Poisson process models bursty arrival rates, which is the reason why several FaaS research works opt for it. However, compared to the Azure's day 1 trace (24h and 908M invocations) we find that multiple other properties are violated. The workload and invocations runtime distributions (Figures 1a, 1b) are shifted to the left (i.e., shorter execution durations), while the requests are uniformly distributed among the functions, thus violating the popularity trends (Figure 1c). Furthermore, as it becomes evident in Figure 1d, the load does not fluctuate throughout the duration of the experiment, despite it being sufficiently bursty (since it is modeled after a Poisson process). Other research works [28, 46, 48] examine other distributions as well (e.g., uniform) to generate synthetic FaaS load, leading to similar problems. Finally, some works [18, 34] isolate the skewed popularity of FaaS functions to artificially generate load, e.g., by directing 98% of the requests to a single function while uniformly distributing the rest 2% to a limited number of functions [18]. Such a strategy still fails to adhere to the actual execution time distributions and to the high variability of the load over time.

Random trace sampling. Other works [11, 16, 29, 34–36, 48] use directly the public traces to generate load. To scale down their experiments, they randomly sample the reported trace functions, and use the invocation arrival time series of the selected subset to generate requests. To further scale down the load, they may proportionally reduce the absolute number of invocations and randomly select a time window within the trace [11, 36, 48, 49]. They typically map the randomly selected functions to real workloads with similar characteristics [16, 35, 48]. However, random downsampling in all dimensions (number of functions, load and time) leads to inconsistent experimentation [49] and, as we show in Figure 1, it can also violate statistical properties. We depict a trace that we constructed by mapping FunctionBench workloads to randomly sampled functions from the Azure trace. We also downsample the number of invocations proportionally [21] to generate a trace of 144K invocations and 2h experiment (similar to the previous example). Figure 1c shows how this approach succeeds in providing skewness in functions popularity (e.g., 80% of the invocations refer to the same function), but Figure 1b shows how the runtimes distribution is far from the target. Also Figure 1d shows that, despite a single huge spike in requests, the general load of the trace is low. These problems are a direct outcome of using only a small random sample of the original trace.

Busy loops. Many of the aforementioned violations of statistical properties are derived, among others, from the limited number of real workloads used in the experiments. To address that, multiple works [18, 34, 48, 49] fabricate synthetic functions, such as busy loops, that follow more closely the trace distributions of execution times and memory usage. However, such approaches fail to provide real FaaS behavior and real data operation.

Open challenges for a representative request generator
A unified methodology to generate FaaS load is still missing. The major challenges we identify are (i) the creation of a pool of workloads that satisfies statistical characteristics of trace functions while encapsulating real FaaS behavior and (ii) the downscaling of trace load and time dimensions, to generate series of requests over time which preserve all critical statistical properties of the production-scale traces.

3 FAASRAIL

In this section we present FaaSRail, a load generator for serverless research, that tackles the challenges and meets the requirements discussed in Section 2. In summary, FaaSRail's goals are to:

(1) scale up the number of the available real-world (non-synthetic) FaaS workloads and map them to a representative subset of trace functions;
(2) scale down the trace with regard to time (i.e., duration of the experiment) and load (i.e., total number of requests);
(3) preserve the key statistical properties of the original trace, as outlined earlier;
(4) provide a utility that combines the above steps to generate a series of requests for real-world FaaS workloads, which closely tracks the original trace, and can be used for consistent evaluation of serverless research artifacts.

FaaSRail is implemented as two open-source components: (i) the offline "shrink ray", which appropriately augments the input benchmarking suite(s), downsamples the input production trace, and creates a mapping between the two to generate experiment specifications, and (ii) the online, versatile load generator that can further parameterize and replay such specifications against a backend FaaS system. FaaSRail receives a *target maximum request rate* and a *target total execution duration* as inputs from the user and generates a representative scaled-down trace of load. It also supports multiple modes, as we discuss next. An overview of the methodology and the design of FaaSRail is illustrated in Figure 2, and elaborated in the rest of this section.

3.1 Mapping trace functions to real workloads

3.1.1 Augmenting the real workloads. Despite the rich information existing in the released public traces about production FaaS workloads, a component critical for their reproducibility is missing. The binaries of the functions are undisclosed. Therefore, researchers' best chance to reproduce such a setting is to use open-source workloads instead. Nevertheless, the number of open-source FaaS workloads which can be reliably considered representative of real-world use cases is low [43–45, 53, 54], in the order of tens, even if aggregated. Azure Functions includes tens of thousands of distinct Functions in their released dataset, and this is merely a small subset of their cluster's actual load [55]. This renders the matching between Functions from public traces and representative open-source workloads a major challenge.

Therefore, the need to synthetically augment such open-source workloads becomes apparent and inevitable. In the scope of this paper, we work with 10 representative open-source workloads adopted from the popular FunctionBench suite [43, 44], shown in Table 1. We make their input more versatile and use a sufficient volume of input data, to make sure that the execution time of each function can vary significantly when possible. We consider each `(function,input)` combination as a distinct Workload, and in this way we generate a pool of Workloads with execution runtimes that span over the whole distribution found in a trace (Section 2). In this way, in the case of the 10 FunctionBench workloads at hand, we manage to collect nearly 2300 distinct Workloads that follow Azure's trace distribution (Section 4).

Table 1: Workloads used in this paper, adopted from the FunctionBench suite, along with a description of their functionality.

FunctionBench	Description
`chameleon`	HTML table rendering
`cnn_serving`	JPEG classification CNN (`tensorflow` [56])
`image_processing`	JPEG image manipulation
`json_serdes`	JSON serialization & deserialization
`matmul`	Matrix multiplication (`numpy` [57])
`lr_serving`	Logistic regression serving (`scikit` [58])
`lr_training`	Logistic regression training (`scikit` [58]))
`pyaes`	Python AES encryption
`rnn_serving`	Word generation RNN (`pytorch` [59])
`video_processing`	Gray-scale effect application (`opencv`)

To register the Workloads execution times, we deploy each in a distinct container and run it multiple times to capture its average warm execution time on a target machine.

3.1.2 Reducing Trace Functions. As discussed earlier, the number of functions reported in published production traces can be large. For instance, Azure Functions' trace [39] contains statistics for more than 80K distinct Functions. Such a number of functions is prohibitively high to replay in the context of a single experiment – even when disregarding the Functions' invocation frequency.

Sampling. In Section 2 we discussed how randomly sampling trace functions may fail to preserve their critical statistical properties. Nevertheless, some coarse-grain sampling can still be effective. Seasonality in the total number of invocations across days shows clear weekly and diurnal patterns in the trace [39]. We further examine the reported daily (a) average execution time and (b) number of invocations for each function in Azure's trace, across all included days. We find that almost 90% of functions yield CVs (Coefficients of Variation) less than 1 (Figure 3) for both (a) and (b), suggesting low variability across days. Therefore, it should indeed be statistically safe to randomly pick a single day of a released trace (e.g., the first), thus reducing the total number of Functions we have to work on.

Aggregation. In Section 2, we discussed that a critical statistical property of public traces is the distribution of execution times of function invocations, and how the typically short execution durations have been a major design factor for FaaS infrastructure. One way to further reduce the number of Functions to work on, while guaranteeing to adhere to the aforementioned distribution, is to aggregate all trace's Functions based on their reported mean execution delays. Each of these groups can then be considered as a single "super-Function" that needs to be matched with one of the realistic distinct Workloads of our pool. We consider the number of invocations for such "super-Functions" (henceforth referred plainly

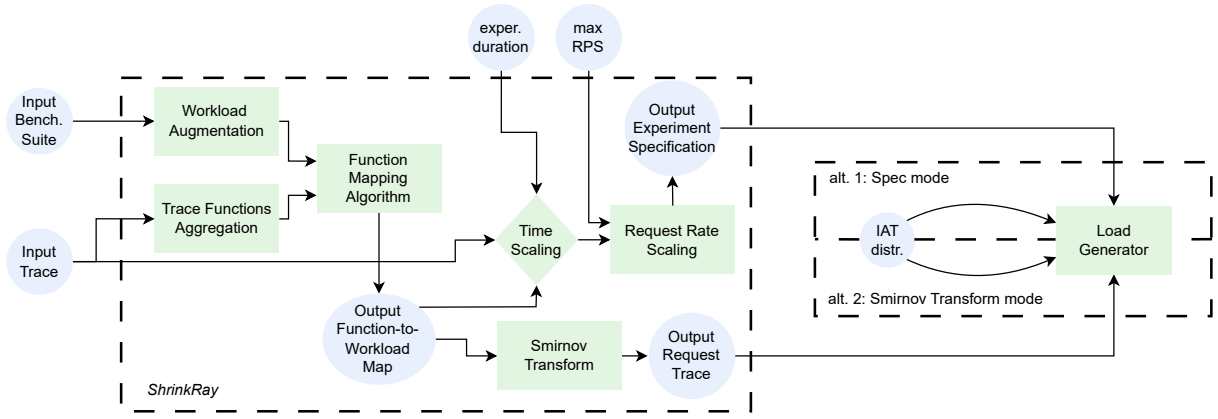


Figure 2: Overview of the methodology components, the flow of information, and the design of FaaSRail.

as Functions) to be the sum of those of the actual trace's functions it comprises.

Popularity. At first glance, it may seem as if the above aggregation of trace's functions based on their average execution time distorts functions' popularity in terms of their number of invocations (Section 2). We move on to meticulously analyze this sort of distortions for the case of the Azure trace. We define a function's *popularity* as the percentage of its number of invocations during a day over the total number of all functions invocations that day. We calculate that value for each initial function in the trace, as well as the corresponding new value for each Function produced by the aggregation step of our methodology. Subsequently, for each new Function, we compare its new popularity value with the maximum one among all initial functions in the trace with the same average execution duration. Figure 4 illustrates the CDF of these differences: the *popularity changes* among the 12757 new Functions due to the aggregation. Apart from 3 Functions whose popularity in terms of invocations is misrepresented by merely 1%, and can be considered outliers, we concur that the vast majority of Functions' popularity values remain virtually unaffected.

3.1.3 Mapping Functions to Workloads. By now, we have accumulated a (reduced) set of Functions based on the input trace, and an (augmented) pool of Workloads produced by the input benchmarking suite(s). We hereby describe FaaSRail's algorithm to map each Function to a Workload, without significantly distorting their respective distributions.

First, FaaSRail defines a (configurable) percentage error threshold; i.e., the maximum percentage of the reported average execution time of a Function that the mapping is permitted to diverge by. Subsequently, FaaSRail associates each Function with a set of Workloads in the pool while upholding this threshold (note that each Workload may be associated with more than one Function at this point). When no Workload can be associated with a Function in accordance with the threshold, FaaSRail just picks the one with average execution time closest to that of the Function. This relaxation is particularly useful for a few long-running outliers, whose rare invocations, even combined, end up never impairing the distribution of execution durations.

Finally, FaaSRail goes through the 1:N mappings of Functions to Workloads and selects one of the mapped Workloads for each Function. The workload selection aims to create a balanced distribution of different benchmarks mapped to Functions, while still converging to the execution time distribution of each function in the original trace.

3.2 Generating load

So far, we have constructed a set of Workloads mapped to representative Functions of a public trace. Each Workload inherits its

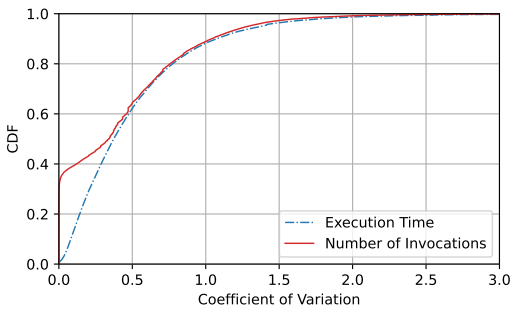


Figure 3: CDF of the Coefficients of Variation of functions' reported daily execution durations and number of invocations, across all days of Azure's trace [39], for all functions in the dataset.

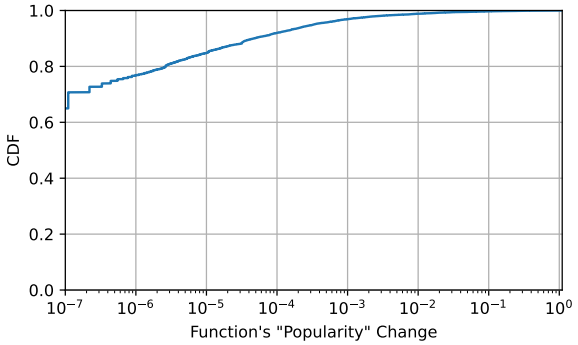


Figure 4: CDF of Functions' popularity changes due to their aggregation based on their average execution time in our methodology, for the case of Azure's trace.

corresponding Function's total number of invocations and their distribution over time (number of invocations during each of the 1440 minutes of the day reported by traces). However, as already discussed, the traces report hundred millions or even billions of invocations within a 24 hour window. In this section, we describe FaaSRail's modes to downscale the load and to generate series of invocation requests over time for the Workloads (FaaSRail output trace) that can be used in a single experiment, without violating the statistical properties of the initial traces.

FaaSRail takes as input (i) a target maximum request rate, and (ii) a target total experiment duration from the user.

3.2.1 Spec mode. This mode directly downscales the data in the traces. First, we describe how it scales the request rate, trying to minimize losses in representativity during the downsampling. Then, we describe how FaaSRail scales in time a trace's invocation rate for each Function.

3.2.1.1 Scaling the Request Rate. Given a target maximum request rate as input, FaaSRail can scale the number of requests that should be recorded on the output experiment specification for each minute. To achieve this, it normalizes the number of invocations per minute in the trace for each Function, so that the total number of invocation requests aggregated across all Functions during the busiest minute in the trace (i.e., also in the experiment) approximates this input target rate, and no other minute ever surpasses it. In this way, FaaSRail effectively downsamples the trace, without severely altering any trends of invocation request rates of the trace over the input minutes. Of course, as it would be true for any downsampling method over a highly skewed trace, misrepresentation of some functions' popularity is inevitable the more the functions' request rates get scaled down. In this case, the aggregation of the trace's functions over their average execution durations, described in §3.1, ends up aiding in minimizing distortions to the final distribution of invocations' execution time due to request rate scaling.

3.2.1.2 Scaling in Time. One of FaaSRail's inputs is a target duration of the experiment. To meet this goal FaaSRail can be configured to use two different methodologies.

Thumbnails (default mode). With this methodology, FaaSRail attempts to downsample the per-function data about invocations over 24h in a way that somewhat preserves each request rate's variability over this time. To accomplish that, for each Function, FaaSRail aggregates adjacent minutes together by summing their reported numbers of invocations. For instance, if configured for a 2 hour experiment, FaaSRail aggregates the 1440 minutes of the day reported in a trace into 120 groups of 12 minutes each. Each such group is mapped to an actual wall-clock minute of the experiment. The number of invocations of each Function during each group (i.e., wall-clock minute of the experiment) is the sum of the invocations per minute reported for all minutes in the group.

This scaling allows for an experiment to capture the diurnal patterns identified within the specific day (Section 2, Figure 1d), at a configurable approximation (i.e., depending on the wall-clock duration of the experiment). This allows studying the behavior of the target machine (or cluster) over a variety of load volumes, all within the same experiment. On the other hand, being effectively a sampling method, this practice ends up smoothening the time series of the number of invocations per minute for each Function. This, in turn, can hide any steep peaks manifested among minutes of the original trace.

Minute Range. Alternatively, FaaSRail can be configured to replay a user-defined minute range in the trace that matches the target total execution duration. This way there is no need for further downsampling in time. This approach is favorable when prioritizing the study of extremely bursty invocation requests at a minute granularity (assuming the original trace's data allows it). However, it overlooks any daily trends manifested (unless a day-long experiment is an affordable option too).

3.2.1.3 Sub-minute behavior modeling. FaaSRail down-scales the per-minute request rates over time reported in its input trace. However, sub-minute request rate distributions need to be modeled as well. To achieve that, by default, FaaSRail uses the per-minute request rate as the intensity (λ) of a Poisson process for that minute; i.e., it inserts exponentially distributed ($\sim Exp(\lambda)$) delays between the invocations. Alternatively, FaaSRail can be instructed to interpret the specified per-minute rate as the deterministic number of requests to be emitted. In this case, inter-invocation delays can be either randomly distributed within the minute (i.e., following a uniform distribution), or equidistant (i.e., constant per-minute rate, although still possibly varying across minutes according to the specification), as implemented in relevant utilities [49]. We opted for modeling after a Poisson process as the default configuration, as this can somewhat realistically emulate burstiness in request arrivals even at the second scale (more in §3.3).

3.2.2 Smirnov Transform Mode. While FaaSRail's main target is to generate downsampled load that follows the invocation rate over time for each Function as reported in the trace, it also supports an execution mode that can emulate synthetic invocation rates per Function.

By appropriately sampling our Workload pool, we can produce a series of requests that follow the distribution of all invocations' execution durations reported in the trace. To achieve that, we apply the Smirnov Transform (also known as Inverse Transform Sampling method) [60, 61], as illustrated in Figure 5 and described below.

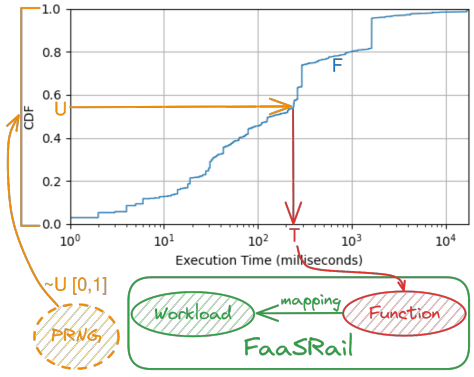


Figure 5: Invocation request generation over Azure Functions' trace using the Smirnov Transform mode in FaaSRail.

Let F_i be the empirical weighted CDF of the execution duration of all function invocations in the input trace during the examined day,

considering again their reported average values. First, we employ a PRNG (Pseudo-random Number Generator) to draw a sample of random numbers, uniformly distributed in $[0, 1]$, of size equal to the number of function invocations we want to trigger during the experiment; we denote the respective random variable as $U \sim \mathcal{U}_{[0,1]}$. Using $\widehat{F}_i^{-1}$, an approximation of the inverse CDF via linear interpolation [62], we can generate T_t from $F_i^{-1}(U)$ according to the Smirnov Transform. Subsequently, we associate the value of T_t to one of the Workloads in our pool as shown earlier. As a result, in this execution mode, the produced series of requests follows the distribution of execution durations of function invocations in the trace (evaluated in §4).

FaaSRail can then replay such a request sample by modeling requests' inter-arrival delays via a specified distribution (e.g., exponential, random, equidistant). This improves over prior work (Section 2), as the latter use fewer workloads, often invoked in equal volumes, and without considering the distribution of their execution durations at all.

3.3 Discussion

In this section we discuss the extensibility of FaaSRail and some of its current trade-offs and limitations.

Flexibility to adopt new traces. FaaSRail is not bound to the currently available FaaS public traces. It fundamentally provides a statistical method to properly downsample traces' data, while preserving some critical statistical properties. The method can be applied to any future FaaS trace if it reports similar statistics.

Flexibility to adopt new real workloads. Similarly, FaaSRail is not bound to FunctionBench [43, 44]; it can be extended to include any real workload. A larger volume of benchmarking suites would lead to even greater variety of output distinct Workloads, dissimilar to one another. In this paper we showcase how merely 10 initial benchmarks can already be sufficient to enrich workload pools through our methodology. However, in the future, we plan to augment and integrate more open-source benchmarking suites (e.g., [45, 53, 54]), aiming to significantly enrich our Workload pool even further.

Memory usage. The current version of FaaSRail focuses on statistics related to the execution time and the invocation rates of functions reported in the published traces. Another important statistic is memory usage. While FaaSRail workloads' memory footprints are similar to the ones used in literature [16, 31, 32, 52], these do not strictly follow the corresponding distributions reported in the traces. Both memory usage and access patterns are characteristics inherent to the applications, and that renders their manipulation a challenge. We believe that enhancing the variety of initial benchmarks is a crucial step towards approaching traces' memory distributions without sacrificing the representativity of workloads otherwise, and we consider this as our next step.

Sub-minute behavior. FaaSRail uses data from public traces to generate representative per-minute load and employs Poisson processes or other statistical methods to generate arrival rates (time-series) within every minute. The reason for this is that Azure's trace reports only invocations per minute. However, Huawei's trace includes also per-second rates, and we consider including this statistic in FaaSRail's method that generates time-series of requests as future work. Nevertheless, the key take-away from Huawei's trace is that burstiness is also present at seconds granularity, and FaaSRail emulates this sub-minute behavior via Poisson processes.

Fixed input per function. In its current form, FaaSRail does not vary the input of every function across invocations, thus the expected execution time does not vary across them. We consider experimenting with variable inputs as a next step.

Long idle times. It should be noted that not every kind of experimentation benefits from scaling the input trace in time. E.g., scaling in time, scales down idle times across invocations (e.g. minutes may become seconds). For that reason, experiments that focus on sandbox caching policies [34], or predictive sandbox preallocation optimizations [39, 40], might be better off using FaaSRail's Minute Range mode that does not scale load in time.

4 EVALUATION

In this section, we evaluate FaaSRail's trace down-scaling and load generation, using as input the Azure [39] and the Huawei [40] public traces introduced in Section 2. We use FunctionBench [43, 44] (§3.1, Table 1) to generate FaaSRail's Workload pool. We attempt to provide answers to the following set of questions:

(1) Does FaaSRail's workload augmentation meaningfully aid in approximating the distribution of execution runtimes, as reported in a production trace?
(2) How well do FaaSRail execution modes (*Spec* and *Smirnov*) approximate: (i) the CDF of the execution runtimes of function invocations, (ii) the distribution of function invocations over time, and (iii) the relative popularity of functions, based on a production trace?
(3) How much does our methodology distort the overall execution characteristics of the input benchmarking suite(s)?

We measure Workloads execution runtimes, calibrated to follow the runtime distributions of each trace (Section 3.1), on a 2-socket Intel Xeon 4314 (IceLake), with 128GiB memory and 16 cores on each socket. To minimize jitter, we use a single NUMA node and run each Workload alone, pinned on a single core with fixed frequency and SMT disabled.

4.1 Function Mapping

Workloads runtime distribution (Q1). Figure 6 shows four CDF plots, all referring to execution runtimes of distinct workloads: (i) Azure Functions' production trace [39], (ii) Huawei private functions' production trace [40], (iii) vanilla FunctionBench suite [43, 44] with representative input found in the literature [31, 52], and (iv) FaaSRail's Workload pool.

It is evident that using as few as 10 vanilla FunctionBench workloads is too inflexible to accurately emulate production workloads. On the other hand, we observe that the CDF of FaaSRail's Workload pool is significantly smoother and approximates Azure's. This showcases that FaaSRail's simple way to augment existing benchmark suites improves the representativity of the generated Workloads.

The Huawei trace refers to internal workloads, and reports execution times for merely 104 distinct ones during its first day. Compared to Azure's, this is significantly smaller, and reports a quite

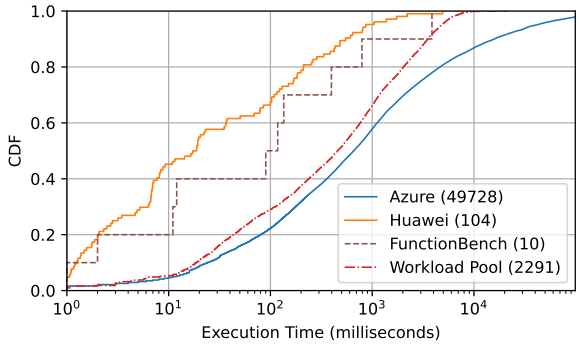


Figure 6: CDFs of execution runtimes (cardinality shown in parentheses in the legend) for: (i) Azure Functions' production trace, (ii) Huawei private functions' production trace, (iii) vanilla FunctionBench suite with commonly used input, and (iv) FaaSRail's Workload pool.

different distribution of execution durations; functions run significantly faster. Judging from Figure 6, and due to the small number of reported functions, even plain FunctionBench workloads might be suitable to represent its runtime distribution to some extent. Nevertheless, the variety in FaaSRail's Workload Pool allows better approximating the distributions of both traces. For the case of Huawei, a smaller subset from the Pool consisting mostly of Workloads with shorter runtimes ends up mapped for generating load from. This is better shown in the next subsection, where we study the distribution of invocations execution durations in a FaaSRail-generated trace.

Workloads memory distribution. While FaaSRail's current version does not include a methodology to approximate the memory distribution of trace functions, we study how far it is from that goal. Figure 7 shows the CDF of the memory usage distributions of Workloads in FaaSRail's pool, along with the CDF reported in Azure's trace for deployed Applications. We observe that the distribution of memory usage of FaaSRail's Workloads is actually not that dissimilar compared to Azure's, by simply setting Workloads memory sizes close to what can be found in literature [31, 52] Nevertheless, it is clearly shifted to its left (i.e., less memory usage in general). This is expected, and already discussed in §3.3. Moreover, note that an

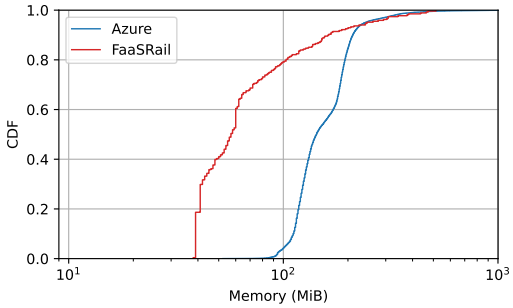


Figure 7: CDFs of memory usage of (i) ~17K Applications (i.e., aggregated ~45K functions) in the first day of Azure Functions trace, and (ii) all distinct Workloads that correspond to ~117K invocation requests produced by FaaSRail in *Spec* mode.

Application in Azure Functions often comprises multiple functions; in this case, their aggregated memory has been reported.

4.2 Spec Mode

FaaSRail's *Spec* execution mode aims to generate load by down-sampling traces while preserving their characteristics with respect to: (i) the distribution of execution runtimes, (ii) the request rate over time, and (iii) the relative popularity of functions. We use FaaSRail's Spec mode to scale the Azure trace down to 2 hours and to 20 requests per second at most. This scales the ~908M invocations of Azure down to ~118K invocations. We evaluate the fidelity of the down-scaled trace to the original, with regard to the aforementioned characteristics.

Invocations runtime distribution (Q2). Figure 9 shows the CDFs of the execution duration of all the invocations in the Azure trace and the down-scaled FaaSRail-generated load. FaaSRail-Spec manages to accurately model the distribution of the original trace.

Invocations request rates over time (Q2). Figure 8 visualizes the relative number of invocations during the 2 hour experiment, along with a similar experiment using a plain Poisson process (Section 2) and the actual Azure's trace (first day only), each normalized to its peak. FaaSRail-Spec uses its default *thumbnail* configuration to scale time (Section 3), meaning that it aims to encapsulate the load's trends of the entire day within the 2h duration of the experiment. It is apparent that FaaSRail successfully models the varying number of invocations of its input trace; it closely follows local minima and maxima of the trace, in contrast to the commonly used plain Poisson process. Furthermore, due to itself employing the model of Poisson arrivals as well to model per-second request rates, FaaSRail achieves to successfully emulate sub-minute burstiness equally well.

Function popularity (Q2). We hereby evaluate function popularity in the same way as in [39], defining popularity as the distribution of invocations across functions. Figure 10 shows the cumulative fraction of total function invocations attributed to the most popular functions in the trace, for both FaaSRail and the Azure trace's first day. FaaSRail's curve is shifted to the right of Azure's due to the lower number of its distinct Workloads compared to the number of functions present in Azure's first day. Apart from that, the two curves have similar characteristics. First, their topmost popular functions are responsible for a disproportionately large fraction of all invocations. Furthermore, their slope is similar, and they both have a tail of functions with low popularity. Judging from these, we concur that FaaSRail manages to sufficiently model the skewed popularity of the input trace functions.

4.3 Smirnov Transform Mode

As described in §3.2.2, in this execution mode, FaaSRail focuses only on following the distribution of execution runtimes of a trace, and issues the invocation requests over time inserting delays based on the specified independent distribution (i.e., exponential, uniform or constant). The purpose of this mode is to enable studies of arbitrary load over workloads representative of the trace. We generate two distinct traces of 120K invocations, each having as input the Azure and the Huawei trace, respectively.

Invocations runtime distribution (Q2). Figures 11a and 11b show the CDF of invocations' expected execution durations, along with

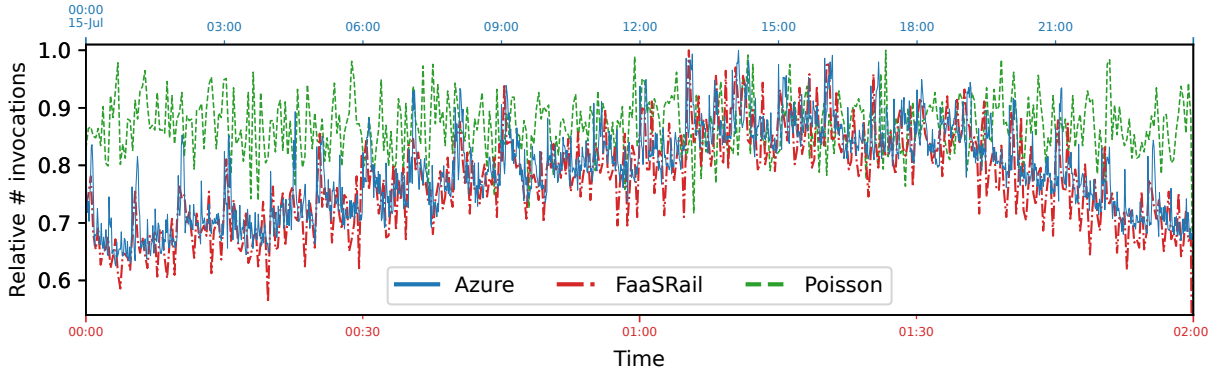


Figure 8: Relative number of invocations (normalized to peak) for: (i) Azure Functions' trace (first day), (ii) scaled down 2 hour, 20 maximum requests per second FaaSRail experiment in *Spec* mode with multiple variable Poisson processes per minute, (iii) plain Poisson process with rate of 20 requests per second, commonly employed in literature. Note that the primary x-axis refers to time in (ii) and (iii), while the secondary x-axis refers to time in (i).

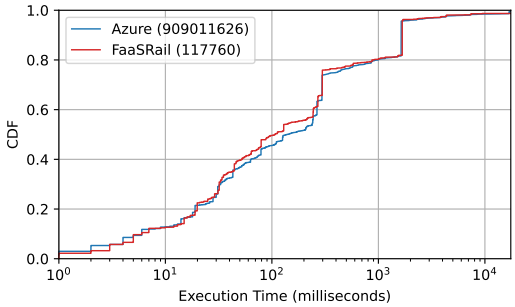


Figure 9: CDFs of the execution runtimes for the Azure trace and the FaaSRail-Spec down-scaled trace.

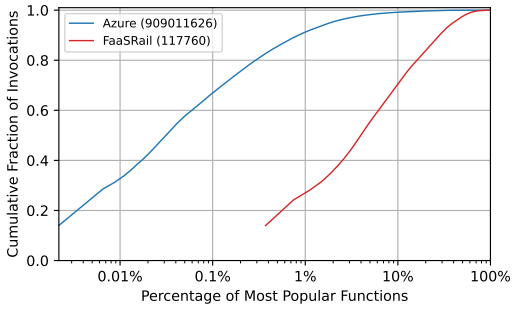


Figure 10: Cumulative fraction of total function invocations due to the most popular functions, for the first day of Azure Functions' trace and a series of 117760 requests produced by FaaSRail in *Spec* mode (configured for a 2 hour experiment, with a maximum rate of 20 requests per second).

that of the corresponding production traces. It is evident that FaaSRail manages to closely model the distribution invocations' execution durations for both cases, thanks to its methodology's Function mapping. For the same reason, the produced invocations adhere to the popularity characteristics that we showed earlier.

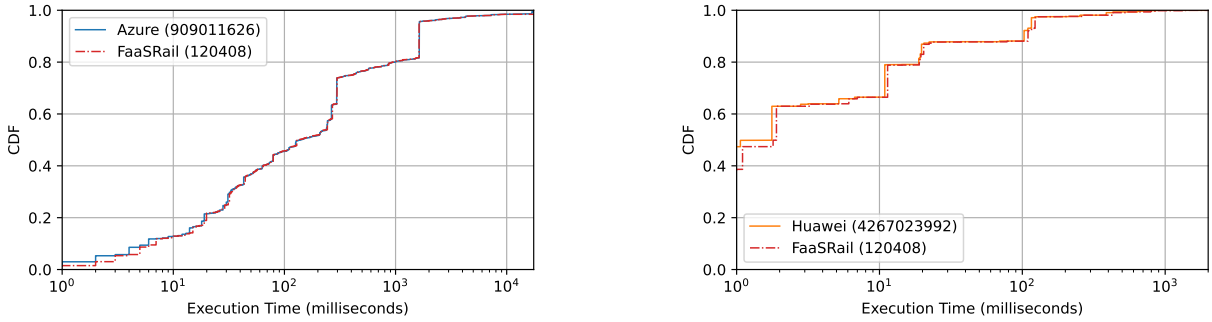
Replaying the produced series of requests against a FaaS system using exponentially distributed request inter-arrival delays –as commonly done to model them as a Poisson process– using a constant rate of requests per second leads to generated load which is significantly more representative of a production trace compared to using a plain benchmarking suite.

Comparison with *Spec* mode. Juxtaposing Figures 9 and 11a reveals a slight divergence of FaaSRail's CDF curve from Azure's in the case of *Spec* mode compared to *Smirnov Transform*. We attribute this to the additionally occurring downsampling steps of FaaSRail's methodology (§3.2.1), which only occur in the case of the former execution mode, since it is the only one taking into account the per-minute invocation request rates of each Function.

4.4 Workload characteristics preservation

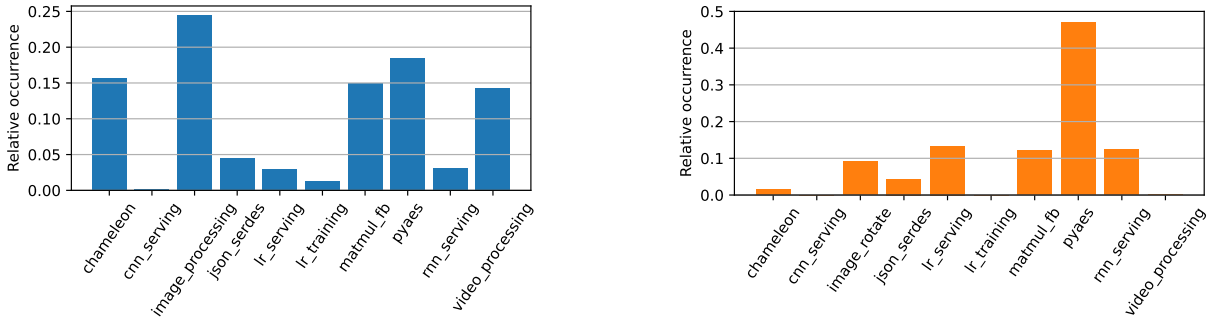
A benchmarking suite, like FunctionBench, includes a set of functions that have certain execution characteristics; e.g., CPU intensity, memory access patterns, I/O activity, etc. Augmenting such a suite unequally into a Workload pool can potentially distort the overall execution characteristics of the suite, especially when the popularity of the trace's mapped Functions does not effectively make up for such imbalance losses. To answer Q3 and determine how does FaaSRail's methodology affect the representativity of the initial benchmarks, we produce request traces using FaaSRail, and plot the number of invocations per initial benchmark (i.e., disregarding its varying input) for both traces in Figure 12.

Figure 12a shows how 118K produced invocation requests are distributed among the initial FunctionBench functions, after mapping the Workload pool to Azure's trace. We observe that most of them are sufficiently represented, despite the discrepancies both among them and compared to a hypothetical, vanilla, equidistributed case (i.e., in which each benchmark would correspond to 10% of the total load). This is due to deviations between the distributions of Azure's trace and FaaSRail's Workload pool, but even more so due to the popularity of certain Functions, which highly affects this balance, essentially dictating the occurrences of the requests through the experiment's specification. The reason for `lr_training`'s low representation is its extremely long execution durations compared to typical FaaS functions according to the production traces. For instance, its quickest variation requires more than 3s to run. However, only about 3% of all invocations in Azure's trace reportedly run for that long or longer (as shown in Figures 11a and 9), which



(a) Azure (b) Huawei (private)

Figure 11: Plotted CDFs of all invocations' expected execution durations (their number is shown in parentheses in the legends) for each production trace examined, against FaaSRail with the respective trace's Function mapping.



(a) Azure (b) Huawei (private)

Figure 12: Balance among benchmark types (i.e., initial benchmarks of the FunctionBench suite, disregarding their varying input) in terms of their percentage of invocations within a request sample produced by FaaSRail, for each production trace.

explains the low representation of `lr_training` in FaaSRail's series of requests as well. The special case of `cnn_serving` being so rare among the produced requests is attributed to the lack of augmentation for the specific workload, which renders it a less probable candidate to be matched to a Function during FaaSRail's mapping stage.

On the other hand, Figure 12b visualizes the same distribution for the case of 35000 invocations produced by FaaSRail's Smirnov Transform execution mode, having its Workload pool mapped to Huawei private functions' trace. In this case, we observe severe imbalances; namely, nearly 48% of all invocation requests refer to some variation of the `pyaes` function. Moreover, three benchmarks (`cnn_serving`, `lr_training` and `video_processing`) never appear in the output request series. The reason for this is twofold. First, as we showed in Figure 6, Huawei's trace includes many short-running functions. Second, not all FunctionBench benchmarks are augmented equally, nor are their execution durations equally versatile (i.e., sensitive to their input). In general, under our current FunctionBench augmentation, `pyaes` dominates in the Workload pool, and even more so among short-running functions, leading to the imbalanced benchmark representation observed in this case. The rest of the invocations are otherwise sufficiently distributed among the six remaining benchmarks.

5 RELATED WORK

In Section 2, we extensively analyzed the common practices found in literature for generating FaaS load, namely: (i) synthetic loads via common distributions (e.g., Poisson process), and (ii) loads derived by randomly sampling a production trace, like Azure's. Here, we briefly discuss some open-source generators that mostly follow such practices.

FaaSProfiler [46] is a tool that generates synthetic load for testing and profiling FaaS platforms. The user defines the target request rate and picks the invocation pattern. It can be generated from a pool of known distributions (e.g., a uniform distribution, or a Poisson process). FaasProfiler uses ad-hoc workloads to generate load, while we focus on augmenting workloads from known benchmarking suites, taking care not to distort their overall performance characteristics.

Illuvatar [48] is a Serverless platform designed primarily for research purposes. Part of the open-source framework is a FaaS load generator. It can be used to generate synthetic loads (e.g., based on a Poisson process, as above), but can also generate loads from production FaaS traces, like Azure's. It randomly samples a subset of functions from the original trace, and to further scale down the load (e.g., concurrent requests), it tweaks their inter-arrival times. It also maps each sampled function to the FunctionBench [43, 44] workload with the closest execution time. In Section 2, we showed

how the random sampling of production traces can generate non-representative load over time, and how blindly mapping trace functions to FunctionBench workloads can generate non-representative runtime distributions.

Very recently, Ustiugov et al. presented **In-Vitro** [49], a methodology and set of tools to generate workload summaries from production traces, instead of randomly sampling them. In-Vitro recursively downsamples the trace while catering to always pick the most representative candidate sample in terms of invocation rate, execution times and memory usage, and models invocation arrivals as Poisson processes. While this work is a more accurate solution for generating representative FaaS workloads from public traces than earlier ones, it still is fundamentally different from FaaSRail. First and most importantly, In-Vitro relies on synthetic workloads (busy loops) to generate load. As discussed in Section 2, such workloads fail to capture complex behavior that is hard to synthesize in a realistic fashion (e.g., memory access patterns, I/O intensity, etc.) and is crucial for certain lines of FaaS research (e.g., intra-node resource allocation, sandbox snapshotting, etc.). Second, In-Vitro operates only upon a user-defined window of the input trace, the size of which dictates the duration of the experiment. It is therefore incapable of encapsulating the entire trace's information (e.g., trends in time).

6 CONCLUSION

We present FaaSRail, a load generator for serverless research. FaaSRail combines open-source, real-world, non-synthetic FaaS workloads from popular benchmarking suites, like FunctionBench, and public request traces of commercial FaaS platforms, to generate representative series of requests suitable for evaluating Serverless research prototypes. We identify critical statistical properties found in traces and explain why generated FaaS load should adhere to them. We describe FaaSRail's approach to scale the generated load down, both in volume and in time, without violating such properties. Finally, we present FaaSRail's execution modes, extensively evaluate their generated load's representativity against the two production-scale traces released by Azure Functions and Huawei, and compare them with common practices employed by today's Serverless literature.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We also thank Georgios Goumas, Filippos Tofalos, Georgia Lytra, and the members of the Computing Systems Laboratory at National Technical University of Athens, for the fruitful discussions and useful comments on the work. This work was funded by the European Union under the Horizon Europe grant 101092850 (project AERO).

REFERENCES

[1] Microsoft Azure. Microsoft Azure Functions. https://azure.microsoft.com/en-gb/services/functions/.
[2] Amazon AWS. AWS Lambda. https://aws.amazon.com/lambda/.
[3] Google. Google Cloud Functions. https://cloud.google.com/functions.
[4] IBM. IBM Cloud Functions. https://cloud.ibm.com/functions/.
[5] Huawei. Cloud functions. https://developer.huawei.com/consumer/en/agconnect/cloud-function/.
[6] Alibaba. Function Compute. https://www.alibabacloud.com/product/function-compute.
[7] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, 2023.
[8] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.
[9] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021.
[10] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, 2023.
[11] Z. Wu, Y. Deng, Y. Zhou, J. Li, S. Pang, and X. Qin. Faasbatch: Boosting serverless efficiency with in-container parallelism and resource multiplexing. *IEEE Transactions on Computers*, 5555.
[12] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, 2020.
[13] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *SoCC 2021 - Proceedings of the 2021 ACM Symposium on Cloud Computing*, 2021.
[14] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, 2020.
[15] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, 2022.
[16] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, 2022.
[17] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, 2021.
[18] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, 2022.
[19] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, 2019.
[20] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. UnFaaSener: Latency and cost aware offloading of functions from serverless platforms. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.
[21] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, 2023.
[22] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, 2023.
[23] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, 2022.
[24] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.
[25] Lazar Cvetković, Rodrigo Fonseca, and Ana Klimovic. Understanding the neglected cost of serverless cluster management. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, 2023.
[26] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[27] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava 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.

[28] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[29] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[30] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[31] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 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.

[32] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, 2022.

[33] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: a fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, 2022.

[34] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, 2021.

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

[36] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. Sfs: smart os scheduling for serverless functions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, 2022.

[37] Al Amjad Tawfiq Isstaif and Richard Mortier. Towards latency-aware linux scheduling for serverless workloads. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '23, 2023.

[38] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[39] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[40] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, 2023.

[41] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The gap between serverless research and real-world systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, 2023.

[42] Apache Software Foundation (ASF). Open Source Serverless Cloud Platform, 2023. URL https://openwhisk.apache.org/.

[43] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019.

[44] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, 2019.

[45] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, 2020.

[46] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, 2019.

[47] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[48] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. Ilúvatar: A fast control plane for serverless computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, 2023.

[49] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. Enabling in-vitro serverless systems research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, 2023.

[50] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, 2022.

[51] David Schall, Andreas Sandberg, and Boris Grot. Warming up a cold front-end with ignite. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[52] Christos Katsakioris, Chloe Alverti, Vasileios Karakostas, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Faas in the age of (sub-)μs i/o: a performance analysis of snapshotting. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, 2022.

[53] vHive Open-Source Community. vswarm - serverless benchmarking suite. https://github.com/vhive-serverless/vSwarm, 2024.

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

[55] Azure. Comparing azure function workloads – released trace vs atc paper, June 2020. URL https://github.com/Azure/AzurePublicDataset/blob/ef8b2517b27357df0b418b6e6ca4efcdeb5117b0/analysis/AzureFunctionsDataset2019-Trace_Analysis.md.

[56] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015.

[57] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 2020. doi: 10.1038/s41586-020-2649-2.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[60] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, USA, 1986.

[61] Wikipedia contributors. Inverse transform sampling — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Inverse_transform_sampling&oldid=1186202367, 2023. [Online; accessed 22-January-2024].

[62] Skipper Seabold and Josef Perktold. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.