FaaSLoad: Fine-Grained Performance and Resource Measurement for Function-As-a-Service

Mathieu Bacou
Samovar, Télécom SudParis, Institut Polytechnique de Paris, France
Inria Saclay, Palaiseau, France

Abstract

Cloud computing relies on a deep stack of system layers: virtual machine, operating system, distributed middleware and language runtime. However, those numerous, distributed, virtual layers prevent any low-level understanding of the properties of FaaS applications, considered as programs running on real hardware. As a result, most research analyses only consider coarse-grained properties such as global performance of an application, and existing datasets include only sparse data.

FaaSLoad is a tool to gather fine-grained data about performance and resource usage of the programs that run on Function-as-a-Service cloud platforms. It considers individual instances of functions to collect hardware and operating-system performance information, by monitoring them while injecting a workload. FaaSLoad helps building a dataset of function executions to train machine learning models, studying at fine grain the behavior of function runtimes, and replaying real workload traces for in situ observations.

This research software project aims at being useful to cloud system researchers with features such as guaranteeing reproducibility and correctness, and keeping up with realistic FaaS workloads. Our evaluations show that FaaSLoad helps us understanding the properties of FaaS applications, and studying the latter under real conditions.

2012 ACM Subject Classification Computer systems organization → Cloud computing; General and reference → Measurement

Keywords and phrases cloud, serverless, Function-as-a-Service, measurement, performance, resource utilization, dataset generation, workload injection

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2024.22

Supplementary Material *Software (Source code)*: `https://gitlab.com/faasload/faasload/`
archived at `swh:1:dir:575aaacc28c42d89a438e4c2c147faf9565d777f`

Acknowledgements While the paper's author is the main developer of FaaSLoad, many other people contributed to it. Their contributions are listed in a dedicated file in FaaSLoad's repository. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).
The author thanks Gaël Thomas for his edits, and Mylène Chameron-Moindrot for her help on figures. Lastly, many thanks to the anonymous reviewers of the conference for their comments.

1 Introduction

There are many ways to use the computing resources of cloud computing. The latest, and arguably most complex one, is *serverless*. Far from monolithic applications, a serverless deployment entails using cloud platform-provided services (such as database servers, message queues, object stores, etc.) to provide the service of an application that is itself cut down into atomic functions. An application is deployed and runs as the concurrent compositions of functions: this paradigm is called *Function-as-a-Service*, or FaaS. A FaaS platform serves requests to its applications by routing them to managed instances of cloud functions.

© Mathieu Bacou;
licensed under Creative Commons License CC-BY 4.0
28th International Conference on Principles of Distributed Systems (OPODIS 2024).
Editors: Silvia Bonomi, Letterio Galletta, Etienne Rivière, and Valerio Schiavoni; Article No. 22; pp. 22:1–22:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

From a systems point of view, FaaS is a wide software stack (listed below bottom-to-top):

1. a hypervisor (e.g., KVM) running virtual machines (VMs);
2. an operating system (e.g., Linux) running as a guest inside a VM;
3. a cluster manager (e.g., Kubernetes) administrating a fleet of guest OSes:
   - it relies on container runtimes (e.g., CONTAINERD) to manage containers in guest OSes;
   - and then a container engine (e.g., RUNC) actually runs containers in guest OSes;
4. a FaaS platform (e.g., OpenWhisk) to implement the FaaS paradigm over the cluster;
5. a managed language runtime (e.g., the Java VM) to run useful application code.

Stacking those numerous layers on top of the hardware can be justified, notably to manage the distributed computing of the FaaS model, and its trade-off between application performance (that benefits from instances being kept idle-warm) and resource efficiency (idle-warm instances occupy resources). However, it makes difficult to do research on the *behavior of the actual end-user application code* that is running: does the garbage collector of a Java virtual machine performs well under FaaS workloads? Does this function suffer from lack of cache locality? Etc. Indeed, most studies focus on coarse properties of the FaaS platform or applications: request latency [17], end-to-end performance of the porting of an existing application to the FaaS model [18, 11], resource management versus application performance [9]... Nonetheless, it remains paramount to understand finely *how a given program behaves on the actual hardware* that eventually executes its instructions [28, 27, 30].

Moreover, most research studies avoid the problem of scale by focusing on a single application at once, but real FaaS applications do not run in a vacuum. Researchers need to understand *how functions interfere with all other instances of themselves, and of other functions* across the platform. Towards this objective, the best reference trace of a real FaaS workload [29] is mostly incomplete, due to privacy constraints and sheer scale: it does not include application code, and it only provides statistical aggregates of properties such as execution times of functions, their memory usage, etc. Again, this is because the objective behind collecting those, was to study at high-level and coarse grain the properties of FaaS applications, instead of finely discovering the properties of programs in a FaaS platform.

To summarize, studying the properties of real programs, running on real hardware, is difficult in the deep, distributed system stack of cloud FaaS platforms. We attribute this difficulty to the lack of adequate tooling to produce relevant data, and to replay and experiment with synthetic or real workload traces.

We identify three hard use cases: (i) building a dataset of function executions, (ii) studying the behavior of FaaS functions, and (iii) replaying a real workload trace. In the first one, the purpose is to build sizeable datasets of function executions, with exact data on them: duration, CPU usage and memory usage over the execution, date of invocation, etc. The end goal could be, for example, to train machine learning models to predict memory usage. In the second use case, the target is to understand the properties of the programs running in FaaS platforms under varied context. It entails injecting a controlled workload, for example to observe the behavior of language runtimes under different circumstances. Finally, the third use case is to observe functions in situ, by reproducing the real constraints encountered in FaaS platforms. Transversal to every use case is the need for fine-grained measurements, for example to understand how architectures behave under FaaS workloads [28, 30].

Inspired by those three use cases, we developed FAASLOAD, a tool to gather fine-grained data about cloud functions. It injects a workload, as invocations of cloud functions, to a FaaS platform. The workload comes either from existing traces, for realistic experiments, or from a synthesizer, to make observations under interesting conditions. During the injection,

FaaSLoad monitors the execution of the functions at the scale of a single request: it collects fine-grained properties about their native programs throughout their execution (e.g., resource usage, architectural performance counters, software events such as context switches, etc.).

FaaSLoad is designed for the FaaS. Its implementation shows flexibility, scalability, agnosticism from the FaaS platform, and features targeted towards systems researchers.

FaaSLoad is written in Python (4177 lines of code[1]). It uses PostgreSQL[2] with the TimescaleDB[3] extension. Its FaaS-platform agnostic core is implemented for Kubernetes and Docker. It is compatible with on-premises Apache OpenWhisk[4] and Knative[5].

We exerted FaaSLoad on the three use cases presented above, including a published work [26], and to summarize, we learnt:

- FaaS, thus also FaaSLoad, is fundamentally asynchronous, with high request frequency;
- FaaS platforms hide the matching between function invocations and function instances;
- not all FaaS platforms implement the FaaS paradigm in the same ways;
- real workload traces could be made more useful with minimal additions;
- FaaSLoad accelerates experiments and observations about FaaS applications.

Section 2 presents related studies and benchmarks about Function-as-a-Service. Then, the paper gives details on the design and implementation of FaaSLoad in Section 3. Injecting real workload traces using FaaSLoad is described in Section 4. Finally, we evaluate FaaSLoad in terms of performance, and on the three use cases above, in Section 5.

2 Related work

When Function-as-a-Service emerged in commercial cloud platforms, there were many interrogations on its properties, as well as how existing and new applications behaved.

2.1 Studies and benchmarks on FaaS platforms

Authors of [33] studied the commercial platforms of Amazon Web Services Lambda, Microsoft Azure Functions, and Google Cloud Functions from the constrained point of view of customers, trying to understand the distributed architecture and finding limitations on performance and security. By nature, this work is limited by the need to reverse-engineer the inner workings of the platform (e.g., to accurately identify colocated function instances).

A lot of benchmark suites, with some companion tooling, have been published [24, 25, 35, 15, 31]. One most advanced example is ServerlessBench [35], but again, it focuses on the FaaS platforms by evaluating global function performance.

Some benchmarks are provided also as tools to lead experiments. BeFaaS [19] has an interesting focus on fine-grained evaluation of applications, although the goal is to evaluate and compare performance of FaaS platforms. vHive [32] is a experimental, representative serverless platform to experiment on the serverless stack. For both, the focus is on the platform itself rather than understanding the programs running in cloud functions.

FaaSLoad can be a support for benchmarking, but its purpose is to support *systems-level studies* by injecting a workload and collecting fine-grained data about the functions' programs.

[1] Counted with cloc: `https://github.com/AlDanial/cloc`.
[2] Home page of PostgresSQL: `https://www.postgresql.org/`.
[3] TimescaleDB home page: `https://www.timescale.com/`.
[4] Apache OpenWhisk home page: `https://openwhisk.apache.org/`.
[5] Knative home page: `https://openwhisk.apache.org/`.

2.2 Discovering the properties of FaaS programs

Other works focused more on the serverless applications. Authors of [23] have added characterization of function properties in a DevOps pipeline. While in a different context than FaaSLoad, their work also include workload generation and profiling of resource usage.

In a more general setting, there is the Serverless Application Analytics Framework [16]. It allows to collect per-function low-level and platform-level metrics, and comes with a tool to lead experiments. By comparison, FaaSLoad avoids function modifications for most data collection, and provides more extended tooling to inject workloads.

Finally, with FaaSProfiler [28], researchers looked into the architectural implications of FaaS. This is close to FaaSLoad's purpose of being able to profile at low-level the programs running in FaaS platforms. However, FaaSLoad covers more use cases, with more complex workloads, in more representative FaaS platforms.

3 Design and implementation of FaaSLoad

In this section, we describe how FaaSLoad's design and implementation respond to the three use cases identified in Section 1: building a dataset of function executions, studying the behavior of FaaS functions, and replaying a real workload trace.

3.1 Design directions

In a nutshell, the purpose of FaaSLoad is to inject a workload into a FaaS platform, and to collect fine-grained, low-level data about the functions that respond to it. The workload is represented as traces of discrete invocations of functions. This is important: FaaSLoad does not send queries to an application, following given parameters; instead, it triggers individual functions with specified arguments. The point is to gather fine-grained data about functions, instead of wholistic information about an application's performance; for example, to gather memory or CPU usage of programs, instead of end-to-end latency of an application.

We identify many properties that FaaSLoad's design must have. First at high level, the design of FaaSLoad is motivated by the need for **flexibility**. It must be useful to any researcher wanting to have a focused look on the programs in cloud functions in varied contexts. This includes their resource usage and performance profiles, at the levels of the system and the architecture. It must also be agnostic from FaaS platforms.

In addition, FaaSLoad's design must answer to inherent constraints of FaaS platforms. First is the **scalability**: invoking thousands of functions at the rate of hundreds of requests per second. Second are **semantic features** such as cold starts (see Section 3.5), etc.

Furthermore, it is also important to FaaSLoad's design to be able to **inject any shape of workload**. Our system is also science-focused, and as such strives to be useful in the context of systems research: it seeks **reproducibility** and **correctness** of its workload injection, and includes **checkpointing** for its long-running dataset generation process. Finally, the data it produces is easy to **explore and exploit**.

3.2 Overview: the injection process of FaaSLoad

The general process of FaaSLoad injecting a workload is shown in Figure 1.

Offline, **(1)** a *trace builder* writes injection traces in FaaSLoad's format. Upon starting, **(2)** the system reads those traces to extract information about the cloud functions to invoke, as well as the *injection trace points*, i.e., invocation requests to send. The main part are the *invokers*, which **(3a)** send invocation requests. Actually, they use the *platform connector*, that

invokes the requested functions by connecting to the FaaS platform. The latter eventually schedules the request to a function instance observed by FaaSLoad's *function monitor*.

An important point is that the invokers do not wait for any injection; instead, there is an *injection monitor* that watches over the whole injection process, waiting for every invocation **(3b)** sent by the invokers, to terminate. When **(4)** one invocation terminates, it fetches its related data from the FaaS platform through the platform connector, as well as **(5)** the data about the execution of the function that served the request, from the function monitor. Finally, the injection monitor **(6)** stores the data to a database for offline use.

As can be seen in this process, there are two main components to FaaSLoad: a *workload injector* (that gathers the invokers and the injection monitor, and uses the platform connector), and a *function monitor*. This modularity allows to only uses the necessary parts of FaaSLoad (e.g., using only the function monitor, with a custom client), making it **flexible**.

We detail below the designs and implementations of FaaSLoad's workload injector and function monitor. They are illustrated, with their internal components, in Figure 2.

3.3 Workload injector

We detail first the workload injector (left side of Figure 2). There are three parts to the workload injector: (i) the injection monitor; (ii) the invoker threads; and (iii) the platform connector (the public interface to the platform in Figure 2).

3.3.1 Injection monitor

The whole workload injection is *supervised* by the injection monitor, which acts as a central point. This thread periodically polls the FaaS platform through the platform connector for terminated invocations. When it happens, it fetches the execution data from the FaaS platform (to collect **semantic information**) as well as from FaaSLoad's function monitor (right side of Figure 2, see Section 3.4), and updates the database.

Notice that the injection is done in an open loop: *invocations are asynchronous*. Indeed, the invoker threads (upper part of the injector in Figure 2, see Section 3.3.2) notify the injection monitor with every invocation they send, and it is the monitor's task to keep track of them, and to handle their termination.

FaaSLoad's injection monitor centralizes collected data offline in a PostgreSQL database, with a special table for timeseries to store resource usage measurements, etc., over time. This standardized storage helps **exploring and exploiting**, by allowing cross-comparison of data, and getting non trivial views on the workload (see Section A about the database).

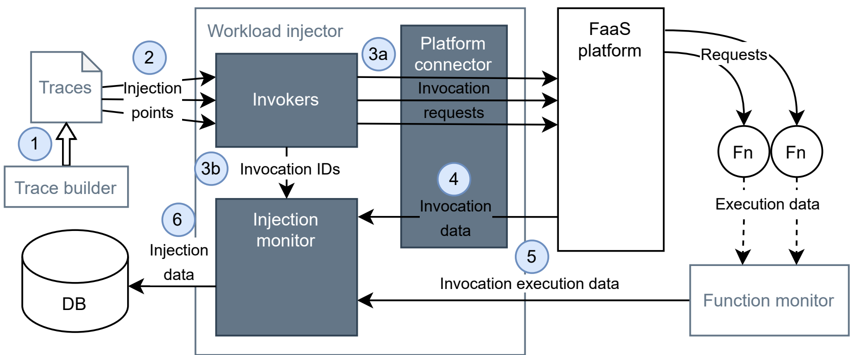


Figure 1 Overview of FaaSLoad's design and its workload injection process.

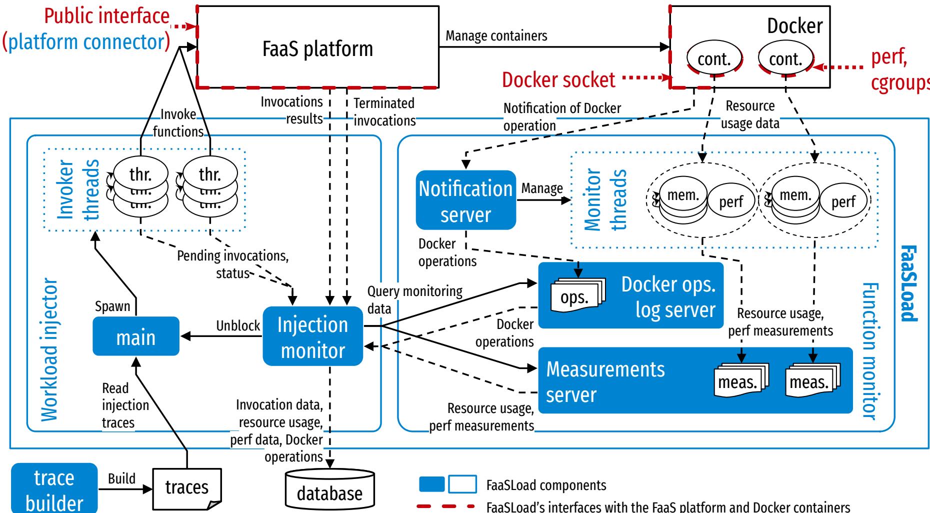


Figure 2 Implementation of FaaSLoad, with the workload injector and the function monitor.

3.3.2 Invoker threads

An injection trace may include several different functions. Furthermore, multiple different users can be simulated; if the same function is invoked under different users, then it is considered as different functions (as is the case in FaaS platforms).

Respective invocations of different functions should be independent from each other, to ensure **scalability** and **correctness** of invocation date. This is why FaaSLoad's invoking component is actually designed as distinct *invoker threads*, each dedicated to a single function (see upper part of the workload injector in Figure 2).

3.3.2.1 Working principle

While functions are of course invoked concurrently by different invoker threads, invocations of one function are *scheduled sequentially per invoker thread*. One invoker thread is fed the invocation requests of its function from the injection trace, and sends invocation queries to the FaaS platform through the platform connector (see Section 3.3.3). Reading injections points from an offline trace makes the process **reproducible**; moreover, the invoker threads' states can be **checkpointed** and restored, to generate huge datasets in multiple runs.

An invoker thread *communicates with the injection monitor* (see Section 3.3.1) by pushing in the latter's queue, the invocations it sends. It also notifies it of its termination, whether because of the end of the injection trace, or because it ended in error. In this regard, invoker threads are made robust by themselves so as not to crash the whole injection, and instead to terminate it gracefully through the injection monitor.

3.3.2.2 Invoker threads as chains of timer threads

The implementation of invoker threads is better detailed in Figure 3. One thread is actually a *daisy chain of timer threads* [7], i.e., threads that start executing after a delay. Starting an invoker thread equates to scheduling a thread to run at the time specified by the first invocation in the injection trace. This thread, before invoking the function as instructed, will similarly schedule the next invocation, and so on, thus constructing a chain of timer threads.

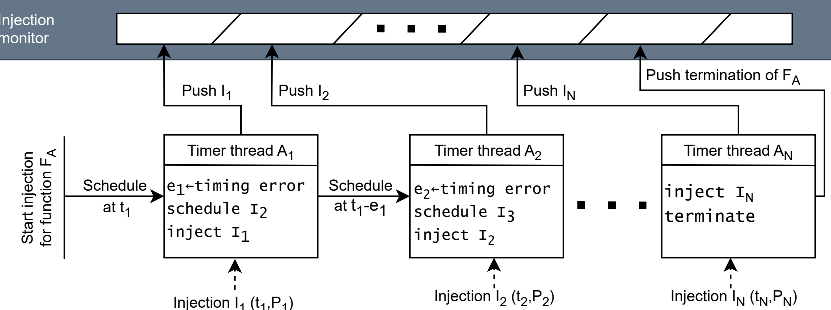


Figure 3 Invoker threads are daisy chains of timer threads, that amortize injection timing errors.

This implementation has two advantages. First, it relates to **correctness** of trace point injections, by scheduling independently each invocation; essentially, invocations are delegated to the OS's scheduler, instead of invoking in a loop with delayed iterations. Second, it helps **scalability** by alleviating the weight on the system by not creating a huge number of threads and timers at once. Indeed, any injection trace may be several hundreds of thousands of invocations long, multiplied by several hundreds of functions. FaaSLoad avoids any performance cost on this side by scheduling each invocation one by one, thus keeping the number of timer threads simultaneously in-flight equal to the number of functions to invoke.

3.3.2.3 Amplification of injection timing errors

However, daisy chaining threads may encounter a problem of *amplifying timing errors*. Indeed, a timer thread schedules the next invocation to happen after the delay given by the injection trace. In the case this timer thread is late or early (due to timing imprecisions,[6] scheduling delays, etc.), this injection timing error will propagate to the next invocation. Recursively, timing errors will accumulate through the entire invoker thread. Timing errors are compensated by deducting any earliness or lateness from the delay to the next invocation.

3.3.3 Platform connector

As seen in Figure 1, FaaSLoad is designed to neatly define the interfaces it requires from a FaaS platform, to be agnostic to it and remain **flexible**. This is embodied by the platform connector (represented as the public interface to the FaaS platform in the upper side of Figure 2). It is the *platform-specific implementation* of workload injector's interactions with the FaaS platform: (i) querying information about invocations, and (ii) invoking functions. New platforms may be supported by FaaSLoad simply by extending an abstract Python class, and implementing those two features.

In this section, we show how the platform connector is implemented for two platforms: Apache OpenWhisk and Knative; we also highlight interesting differences between them.

3.3.3.1 Invoking functions

To invoke a function, the platform connector *translates one point in the injection trace as a query* directed at the function via the FaaS platform. While both platforms expose RESTful Web APIs, the semantics to invoke a function interestingly differ.

[6] FaaSLoad is bound to the resolution of the nanosecond-scale system timers.

Knative only allows to invoke a function through a public URI, forcing the user to consider it as a web service; Knative acts as a very scalable, dynamic HTTP proxy. This is consistent with Knative's model where functions must handle themselves HTTP requests.

On the opposite, OpenWhisk provides a Web API to invoke a function as a REST object that receives processed parameters instead of the raw HTTP request. This is an interesting difference: Knative only is a set of Kubernetes resources to provide Function-as-a-Service, while OpenWhisk really implements objects and semantics specific to FaaS. In our experience, it is much easier to work with OpenWhisk, thanks to *well-defined APIs*.

Interestingly, while OpenWhisk supports asynchronous requests via its REST API, Knative requires a convoluted setup. Thanks to FaaSLoad's implementation of invoker threads using timer chains (see Paragraph 3.3.2.2), this is not a problem: one invocation of a function may block while the next invocations are scheduled anyway.

3.3.3.2 Monitoring terminated invocations

Monitoring terminated invocations is achieved in the injection monitor by *polling*. This is either through a dedicated REST URI for OpenWhisk, or by parsing logs of internal components for Knative. Nonetheless, in both cases, matching an invocation ID with the function instance that served it (in order to query execution data from the function monitor, see Section 3.4) requires awkward parsing of internal logs of the FaaS platform. While it is understandable that normal users of FaaS platforms cannot request such low-level, internal information, it would be helpful to provide it for administration and monitoring purposes.

3.4 Function monitor

After covering the design and implementation of the workload injector, we focus on the function monitor that can supplement FaaSLoad's workload injection (right side of Figure 2).

The role of the function monitor is to *collect fine-grained data about the programs* running in the cloud functions: CPU and memory usage during the execution, CPU cache misses, etc.; additionally, it watches events related to the management of functions instances (i.e., Docker containers[7]): creation, deletion, etc.

3.4.1 Architecture of the function monitor

In FaaSLoad's global view, the monitor is a *server*, independent from the workload injector for **flexibility**: it is the job of the latter to request the former, to fetch the fine-grained data about the execution of the function that served an invocation. However, to ensure **scalability**, the monitor is actually *distributed*: one instance of it exists on each Kubernetes node of the platform, to monitor local cloud functions. It is the workload injector that requests the right monitor instance to fetch function execution data.

This design limits its use to FaaS platforms deployed on-premises; i.e., it cannot be used on commercial platforms such as Amazon Web Services, Microsoft Azure, Google Cloud, etc. This is aligned to FaaSLoad's goal of collecting any low-level performance information at fine grain, which would not be permitted on them.

[7] The function monitor requires Docker to monitor its socket, but could easily be made to work on other container backends that also use a UNIX socket where it could listen for function container events.

In addition, the monitor scales one-to-one with the function instances. Indeed, the monitor is a manager of threads: for every function instance, it spawns a *monitor thread* (upper part of the monitor in Figure 2) to collect the related data, which is stored in a shared data structure. The monitor responds to a request from the injector via its *measurements server* (lower part of the monitor in Figure 2), that looks into this structure.

3.4.2 Monitoring functions

Here, we detail how the function monitor collects function execution data at fine grain.

3.4.2.1 Data collection

By itself, the function monitor only monitors memory usage of functions, via *one thread per function*, that periodically measures the sum of its resident-set size (RSS) and the total size of its mapped files [6]. Most monitoring is done by *delegating the production of measurements to external programs*, so the monitoring capabilities can be extended. The function monitor runs, in a dedicated thread per function, the multipurpose monitoring tool PERF [5]: it can be configured from FAASLOAD's configuration files to monitor many hardware and software performance counters, including CPU usage, CPU cache misses, system calls, etc.[8]

In the end, the function monitor gathers all measurements in a map shared between monitoring threads and a measurement server. An element of this map represents all measurements about a single function instance. Thus, measurements for a given function instance are only shared between its monitor thread and the measurement server.

3.4.2.2 Monitoring interfaces

The monitoring programs such as PERF work at the process level. Thus, the function monitor matches function instances with processes for them, and collects their output. It is designed *to work at a lower level than the FaaS platform.* For **flexibility**, it even works standalone.

Monitoring occurs through the low-level interface of *control groups*, a feature of the Linux kernel required to implement containers. Indeed, a function container is built, among other things, from a memory control group; while it is firstly used by the FaaS platform to limit the memory usage of the function, the function monitor reads its accounting files to measure the memory usage (see Paragraph 3.4.2.1). PERF also accepts monitoring processes identified by a control group. In case another performance monitoring program would require a process ID instead, it is easy to retrieve the list of processes inside a control group.

By monitoring functions from the outside, the monitor does not incur any *overhead* on the functions nor on the FaaS platform themselves. At worst, it incurs a CPU usage penalty on the node of the FaaS platform, which can be ignored by properly separating its resources from resources allocated to the FaaS platform and its function instances. This method also does not require *any modification to the functions* under study. Nonetheless, because FAASLOAD's injection monitor also collects the data returned by an invocation (see Section 3.3.1), functions can be augmented to return meaningful information, adding to **flexibility**. For example, a function that processes an image could return a more precise view of the execution time by detailing the time to extract the image from data storage, the time to transform the image, and the time to load the result back to data storage.

[8] Using PERF is currently hardcoded in FAASLOAD, but integrating another such tool would be quick, as collecting measurements is abstracted from parsing PERF's output.

3.4.2.3 Data servicing: matching invocations with function instances

A client (e.g., the injection monitor, see Section 3.3.1) requests the function monitor for execution data of an invocation. Recall that the monitor does not interact with the FaaS platform, thus it does not know anything about invocations. Its client must cross the gap between invocations and function instances, by providing in its request to the measurements server, the *container ID matching an invocation.* This information relates to a FaaS platform-specific internal scheduling decision, obtained via the platform connector (see Section 3.3.3).

Moreover, the function monitor's monitoring threads always run as long as their target function instance exists. It means that the monitor collects measurements that may not match an invocation. Indeed, an instance may be kept warm by the FaaS platform to quickly serve future requests. Thus, a client must provide the measurements server with the *time window* of measurements: the start and end time of the invocation. To this request, the measurements server responds with all the measurements that match these criteria.

This implementation allows the function monitor to be completely independent from any FaaS platform, but also to be useful at lower levels than the FaaS platforms.

3.5 FaaS semantics

FaaSLoad is designed for the FaaS context. It includes notions of functions, as well as different users to invoke functions separately on the FaaS platform. Its workload injector invokes functions with parameters through the public interface of FaaS platforms (i.e., it does not bypass the normal path), and it gathers their results, including their potential failure.

Most importantly, FaaSLoad takes into account cold starts, by storing the added initialization time, and marking invocations served by a cold-started instance. Indeed, when a FaaS platform serves a request to a function, it has two choices: spawn a new instance, or reuse a previous idle instance. The former case is called a *cold start*, while the latter is conversely called a warm start. A cold start increases the response latency of the function, because the FaaS platform must first initialize a new instance [10, 17]. Information about cold starts is provided by the FaaS platform, via the platform connector (see Section 3.3.3).

Finally, FaaSLoad gathers a global view of function instance events (see Section 3.4). In addition, mapping invocations to instances is easily achieved by customizing cloud functions to keep track of their instances, and return this piece of information as a result to be stored by FaaSLoad (see Section 5.3 for another example of function customization).

4 Workload traces

FaaSLoad injects workloads provided under its own format. This format is designed to allow **injecting any shape of workload**, by including only the minimum but complete (for **reproducibility**) information required: the function to invoke, and a list of invocations. FaaSLoad relies on off-line trace producers (see Sections C and 4.2 for examples) to make the traces. Here, we describe the trace format, and how to use real workloads.

4.1 Trace format

Injection traces are simple text files containing lines of tab-separated values. An example is given in Section B. One invocation is simply one line listing the inter-arrival time from the previous invocation, and the arguments to pass to the function.

FaaSLoad's traces declare injection points using delays between each other, instead of relative time since the beginning of the trace. This has two advantages: it simplifies implementing the invoker threads explained in Section 3.3.2; and it simplifies implementing trace generators based on statistical distributions of the inter-arrival time (IAT) (see Section C).

4.2 Real workload injection

FaaSLoad allows to inject real workloads into a FaaS platform. This relies on two pieces:

1. a dummy cloud function which execution time and memory usage are set by its parameters;
2. a converter from the real workload traces to FaaSLoad's injection traces.

While the first one is straightforward, the second one, converting real workload traces, is much more difficult. Indeed, the canonical data of real workload traces published by Microsoft Azure [29] does not include essential information such as the actual functions that are executed, or even exact invocation dates. First, this comes from the obvious confidentiality requirement of the workloads; and second, the sheer amount of invocations in any sizeable dataset makes collecting complete data prohibitive, so Azure's data only include *statistical properties* that summarize features: quantiles of the inter-arrival times for a given time slice, of the memory usage of whole applications instead of individual functions...

4.2.1 Converting Azure workload traces for FaaSLoad

We worked on processing the Azure dataset of cloud function traces into injection traces. Based on previous work [20], we implemented the following process:

1. **sampling down**: Azure traces must be reduced for the experimental testbed: selecting a time window, keeping "interesting" functions, randomly selecting a subsample of users;
2. **building the call graph**: refine the traces by finding dependencies between functions, and allow replaying causality;
3. **computing concrete values from statistics**: Azure traces include averages and quantiles for function invocation rates and execution times, and application memory usage that must be "realized" into workable function IATs, durations and memory usages.

4.2.2 Limits of the conversion process

The conversion process of Azure workload traces has hard limits: sampling criteria are arbitrary; we compute concrete IATs by assuming they follow a Poisson arrival process, which is only correct for a small fraction of functions [29]; and we compute concrete memory usage of functions by distributing an application's memory usage to its functions depending on their relative execution duration, despite no correlation.

To inject more correct traces, and beside exhaustive traces, we suggest including the following features that would still preserve the confidentiality of the workload, and increase weakly to moderately, the size of the dataset:

- scale of the hosting hardware (number of machines, CPU specifications...) to reduce arbitrary sampling, by doing linear scaling instead;
- triggered side-effects: functions can be triggered by many sources, so including the sources triggered by functions allows deducing the call graph of functions inside an application;
- IAT statistics instead of invocation rates;
- per-function memory usage statistics.

Newest datasets [22, 13, 21] show improvements. The latest from Huawei data centers [21] includes exact invocation timestamps and per-function resource usage. Information about the call graph is still missing, as well as the scale of the hosting data centers.

5 Evaluation

In this section, we first evaluate FaaSLoad's performance to inject a workload in Section 5.1. Then, we showcase its features through three use cases: building a data of function executions for machine learning in Section 5.2, studying the Java Virtual Machine in a FaaS setting in Section 5.3, and replaying a real workload trace in Section 5.4.

5.1 Scalability of FaaSLoad

First, we evaluate the scalability of FaaSLoad's workload injector through two experiments. In the first one, we check the scalability of one invoker thread in the face of a bursting injection trace. In the second one, we check the scalability of the whole injector over the number of functions it has to invoke. In both experiments, the metric is the injection timing error, i.e., the time difference between the actual invocation date and the expected invocation date from the trace. Positive errors mean it was late; negative errors denote early invocations.

The setup is as follows: FaaSLoad runs on a dedicated machine, connected to a Kubernetes cluster of 8 nodes. In the cluster, Apache OpenWhisk is deployed, and configured for scalability (2 invokers per node, all memory available). All machines are identical: every machine is a Dell PowerEdge R640 from 2019, with 96 GiB of memory and an Intel Xeon Gold 5220 at 2.20 GHz with 18 cores, running Debian 11 (or Debian 12 for the machine running FaaSLoad), from the distributed testbed Grid'5000 [12]. We provision an OpenWhisk deployment that can serve without overhead the requests sent by FaaSLoad while it is under test, so that the results are not affected by OpenWhisk's performance.

5.1.1 Scalability of invoker threads

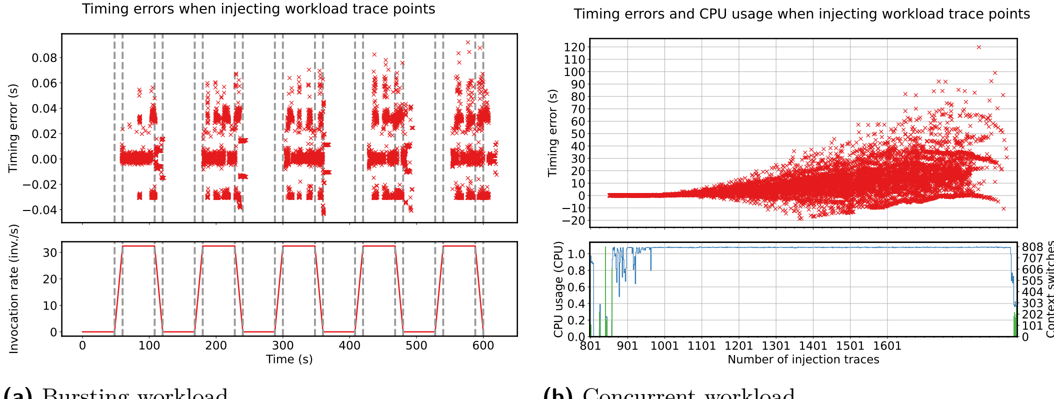
In this experiment, a single function is injected by FaaSLoad. The function is the dummy function used to inject real workload traces, see Section 5.4. The function's invocation profile is based on the function from the Azure dataset [29] with the 99% highest invocation rate. The injected function starts at its low invocation rate of 0.017 inv./s, i.e., once per minute; then ramps up to its high invocation rate of 32.4 inv./s, and finally ramps down back to the low invocation rate. Its synthetic trace (see Section C) repeats this bursting phase 5 times.

Results are displayed in Figure 4a. The bottom plot illustrates the invocation profile of the function described above. The top plot shows the timing errors of the invocations.

Timing errors range in absolute value from 1 µs to 92 ms, with a mean of absolute timing errors of 12 ms. For reference, in Azure traces [29], the median execution time is a bit under 900 ms. Note that in practice, FaaSLoad does not compensate timing errors below a configurable threshold (usually set at 10ms) to avoid overcompensating insignificant errors.

5.1.2 Scalability of the workload injector

In this experiment, the dummy function is injected under the same rate (0.033 inv./s, i.e., twice per minute, the median maximum rate in the Azure dataset), but under different users, i.e., as concurrent, different functions. The number of concurrent functions being injected increases over time. The rate of an individual function is low enough to avoid timing errors in a single invoker thread, so this experiment evaluates the scalability of the whole injector.



(a) Bursting workload.

(b) Concurrent workload.

Figure 4 Scalability of FAASLOAD's invoker threads: timing errors when sending invocations of functions. **(a)** A single invoker thread under a bursting workload; **(b)** concurrent invoker threads.

The results are presented in Figure 4b. The bottom plot shows the CPU usage, and context switches lives by FAASLOAD's workload injector. The top plot is as in Section 5.1.1, showing the timing errors for all invoker threads alike.

Results start at 801 concurrent traces, because the invoker shows no significant timing errors before this point. Timing errors start to explode after 1001 concurrent functions. FAASLOAD's workload injector, using only Python's threads, is not parallelized because of Python's Global Interpreter Lock, as shown by the CPU usage capping at 1 full CPU.

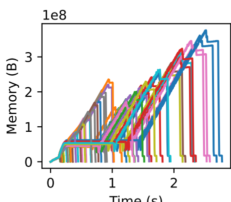
5.1.3 Conclusion on the scalability of FaaSLoad

FAASLOAD is capable of handling bursting workloads at the scale of real workload traces. However, the runtime limitations of the threading implementation of Python interpreters severely inhibits its capability to scale to the breadth of real workload traces in terms of concurrent workloads, capping at around 1000 concurrent traces of function invocations. Nonetheless, Table 1 in Section 5.4 shows this is enough for a representative workload trace. Possible improvements include rewriting the invoker threads in different ways:

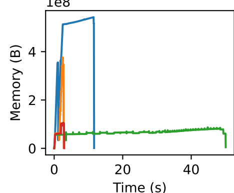
- in Python, using multiprocessing [3]: spawn new interpreters in subprocesses, with separate GILs, but without implicitly shared memory, making the implementation harder;
- in Python, with the coming feature of subinterpreters [4]: a lighter form of multiprocessing with less overhead and implicitly shared memory;
- in a systems language: best potential performance, but this would require the most work.

5.2 Use case 1: building a dataset for machine learning

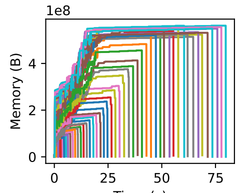
FAASLOAD was used for building a dataset for machine learning, in a published paper [26]. In this work, we studied the impact of features of functions inputs on their memory usage, and then developed ML models to infer the memory usage. For instance, image processing functions may use different amounts of memory depending on the image size, format, etc. FAASLOAD collected a dataset of memory usage of 12 functions.



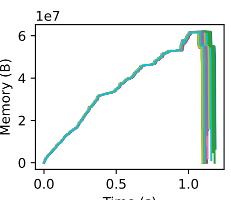
(a) Blurring an image with SHARP (NodeJS): image dimensions.



(b) Blurring an image with SHARP (NodeJS): image format.



(c) Blurring an image with SHARP (NodeJS): blur coefficient.



(d) Summarizing a text with SUMY (Python): length of text.

Figure 5 Examples of data collected with FAASLOAD: variability of memory usage during function execution, depending on input features. In each figure, one color represents one value of the feature.

To build a dataset with FAASLOAD, the setup is a follows: declare functions under study in a manifest file,[9] then use FAASLOAD's helper tool to load the functions into OpenWhisk; finally to build the dataset, let FAASLOAD run automatically under a reproducibly random injection trace, and with the function monitor configured to collect needed data.

Figure 5 shows a few example results of the data obtained by FAASLOAD. In this work, we successfully trained J48 (C4.5) models [34] with up to 91% of accuracy, thanks to FAASLOAD providing data to identify significant features, and to train models. In addition, data showed how memory usage can vary during a function execution, with different patterns.

5.3 Use case 2: studying the Java Virtual Machine for FaaS

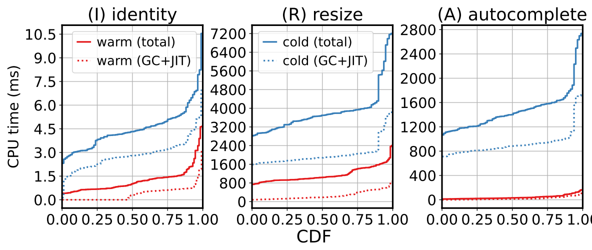
Using FAASLOAD, we conduct an experiment on a Java virtual machine (IBM Semeru OpenJ9 VM[10], used as the default Java VM by OpenWhisk): we examine the Garbage Collector (GC) and the Just-In-Time (JIT) compiler. In the Java VM, the GC runs concurrently to the application, to free memory from the heap by clearing unused objects. As for the JIT, it profiles the application on its first run to find code pieces that would benefit from being executed natively, i.e., compiled to the host architecture, instead of being interpreted.

Specifically, we measure the effects of cold starts on the Java VM itself, beyond the latency they add to function invocations. In this experiment, we check if, and how, cold starts affect the function's language runtime, by focusing on the Java VM's GC and JIT.

We ran FAASLOAD to generate a dataset of executions for 3 Java applications: (`I`) an **`identity`** function returning its parameters without any processing, (`R`) an image **`resizing`** function, and (`A`) an `autocompletion` algorithm (taking a radix as parameter, it returns the most probable matches from a dictionary). We had FAASLOAD collect the detailed CPU usage of the Java VM. This uses the VM's specific interface `JvmCpuMonitorMXBean` [2], that provides the breakdown of `total` CPU usage between `GC+JIT` (i.e., the Garbage Collector and the Just-In-Time compiler) and `application`. Functions have been modified to return this breakdown, and FAASLOAD collects it for data processing.

Figure 6 shows transposed CDFs of CPU usage for the three functions. There is a visible increase in CPU usage and variability between cold and warm starts, both for `GC+JIT` and for `application`. About warm executions, this is easily explained: the Java code has been JIT compiled, it is now optimized and simply runs more efficiently. About cold executions, the

[9] This is inspired by OpenWhisk's manifest files used by its tool WSKDEPLOY [1].
[10] Home page of Semeru: `https://developer.ibm.com/languages/java/semeru-runtimes/`.



(a) Distribution of CPU usage.

Func.	`total`	`GC+JIT`	%
(`I`)/w	1.2	0.4	33
(`I`)/c	4.5	2.9	64
(`R`)/w	1150	260.9	23
(`R`)/c	3840.0	2022.0	53
(`A`)/w	42.8	22.2	52
(`A`)/c	1470.0	922.1	63

(b) CPU usage (mean, ms). /w = warm start, /c = cold start.

Figure 6 Impact of cold starts on CPU usage by Java functions and their VM's GC and JIT.

results show that both the GC and the JIT compiler use more CPU time (both in absolute, and relatively to the `total` CPU time) in cold executions. By definition, the cold execution of a function instance is the very first one, and this is the one that suffers the JIT overhead the most, as well as some GC overhead of cleaning initialization objects.

As a takeaway, during cold start invocations, `GC+JIT` CPU usage can represent more than 60% of `total` CPU time: the cold function instance spends more time optimizing the program, than running it. This trade-off may be beneficial for long-running applications, but in FaaS applications, a function instance may never be reused: then the optimizations are lost, because they are not shared between instances (although some works started looking into that [14]). We conclude that current language runtimes are not FaaS-ready.

5.4 Use case 3: replaying a real workload trace

In Section 4.2, we showed our conversion algorithm for the Microsoft Azure Functions Dataset from 2019 [29]. In this section, we evaluate their correctness when converted for replaying by FaaSLoad. Note that in this evaluation, we skip building the call graph (see Section 4.2.1), because it was not reliable due to the lack of data from the original traces.

Major challenges are to compute concrete values for Inter-Arrival Times (IATs), and function execution times and memory usage, while keeping the same statistical properties as the original, incomplete and at times inconsistent traces. We evaluate the fidelity of the conversion, by calculating the same parameters as in the companion paper to the traces.[11]

They are shown in Table 1. For execution times, the parameters are the log-mean and standard deviation of the log-normal fit to their distribution; as for memory, they are the parameters of a fitted Burr XII distribution of the memory usage of applications.

As expected, the shorter the sampling window, and the smallest the sampling subset, the less representative the resulting traces. Nonetheless, results would vary on the exact sampled time windows (here, they all start at the beginning of the traces). As shown in Section 5.1.2, FaaSLoad can handle up to around 1000 concurrent functions, so Table 1 indicates that the best trade-offs would be to sample over 1 h and 1% (closest log-mean execution time), or over 10 min at 2% (closest memory statistics), although it of course depends on the objectives.

[11] Note that the values of the parameters we calculated over the original traces sampled at 100%, are different from those given in the companion paper, because the authors published different traces.

Table 1 Comparison of FaaSLoad traces sampled from a real workload, with the original workload. In **bold**: values of fitting parameters closest to 100% of original. IATs distrib. gives fractions of applications invoked less than once per hour, and less than once per min.

Sampling			IATs distrib. (%)		Exec. time		Memory		
Window	Sub%	# funcs.	$\leq 1/h$	$\leq 1/min$	μ	σ	c	k	λ
original	100.00	49816	67	91	-0.52	2.68	7.023	0.681	143.671
original	1.00	11874	42	77	-0.45	2.33	14.446	0.217	112.216
1 day	100.00	26151	31	72	-0.81	3.58	14.212	0.217	116.108
1 day	1.00	4913	19	63	-0.74	2.89	14.794	0.233	110.707
6 h	100.00	13248	16	63	-1.04	4.10	17.147	0.177	114.513
6 h	1.00	2198	17	52	-0.29	**2.83**	17.077	0.196	111.673
1 h	100.00	8389	0	55	-1.38	4.75	17.901	0.164	113.775
1 h	1.00	1086	0	49	**-0.46**	3.80	26.227	0.125	111.074
30 min	100.00	7443	0	51	-1.54	4.94	17.913	0.161	113.163
30 min	1.00	919	0	42	-0.64	3.45	42.380	0.083	112.989
10 min	100.00	6034	0	44	-1.85	5.23	268.939	0.005	73.303
10 min	2.00	980	0	64	-1.34	2.43	**8.268**	**0.413**	**132.611**
10 min	1.00	449	0	37	-2.20	4.39	453.219	0.004	95.477

6 Conclusion

In this paper, we introduced FaaSLoad, a workload injector for FaaS platforms to gather fine-grained data about functions. FaaSLoad's aim is to help in systems research on FaaS platforms, by collecting fine-grained performance and resource data about the actual programs running serverless, and by injecting synthetic and real workloads into FaaS platforms for experiments. We showed that FaaSLoad is efficient in conducting experiment in FaaS environments, and allows to replay real workloads.

In the long term, we hope that FaaSLoad will encourage systems research in directions that cross-cut the deep software layers of cloud computing.

For now, we will work on improving the performance of the workload injector, by rewriting critical components in a system language instead of Python. Additionally, FaaSLoad could support the concept of applications as seen in Serverless platforms (most often, graphs of functions), to handle more realistic workloads based on causality. Another interesting feature would be to control the cold starts, given how impactful they are on application performance. Finally, a community effort could expand FaaSLoad's utility to more use cases.

References

1 Apache OpenWhisk utility for deploying and managing OpenWhisk projects and packages. URL: https://github.com/apache/openwhisk-wskdeploy/tree/master.

2 Interface JvmCpuMonitorMXBean. URL: https://www.ibm.com/docs/api/v1/content/SSYKE2_8.0.0/openj9/api/jdk8/jre/management/extension/com/ibm/lang/management/JvmCpuMonitorMXBean.html.

3 multiprocessing – process-based parallelism. URL: https://docs.python.org/3/library/multiprocessing.html.

4 PEP 684 – A Per-Interpreter GIL. URL: https://peps.python.org/pep-0684/.

5 perf: Linux profiling with performance counters. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

6 The Linux kernel documentation: Memory Resource Controller: Misc. interfaces: stat file. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/memory.html.

7 threading – Thread-based parallelism: Timer Objects. URL: https://docs.python.org/3/library/threading.html#threading.Timer.

8 What AWS Lambda's Performance Stats Reveal: Memory Usage, April 2019. URL: https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/.

9 Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana- Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434. USENIX Association, 2020. URL: `https://www.usenix.org/conference/nsdi20/presentation/agache`.

10 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 Haryadi S. Gunawi and Benjamin C. Reed, editors, *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 923–935. USENIX Association, 2018. URL: `https://www.usenix.org/conference/atc18/presentation/akkus`.

11 Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 263–274. ACM, 2018. `doi:10.1145/3267809.3267815`.

12 Daniel Balouek, Alexandra Carpen-Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding Virtualization Capabilities to the Grid'5000 Testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science - Second International Conference, CLOSER 2012, Porto, Portugal, April 18-21, 2012. Revised Selected Papers*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer, 2012. `doi:10.1007/978-3-319-04519-1_1`.

13 André Bauer, Haochen Pan, Ryan Chard, Yadu N. Babuji, Josh Bryan, Devesh Tiwari, Ian T. Foster, and Kyle Chard. The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Gener. Comput. Syst.*, 153:558–574, 2024. `doi:10.1016/J.FUTURE.2023.12.007`.

14 João Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: leveraging runtimes for the serverless era. In Sebastian Angel, Baris Kasikci, and Eddie Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor , Michigan, USA, June, 1-3, 2021*, pages 58–64. ACM, 2021. `doi:10.1145/3458336.3465305`.

15 Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: a serverless benchmark suite for function-as-a-service computing. In Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga, editors, *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, pages 64–78. ACM, 2021. `doi:10.1145/3464298.3476133`.

16 Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, and Wes Lloyd. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software. In *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*, pages 67–72. ACM, 2020. `doi:10.1145/3429880.3430103`.

17 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 James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020. `doi:10.1145/3373376.3378512`.

18 Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 363–376. USENIX Association, 2017. URL: `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi`.

19 Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Xiaohang Zhao, and David Bermbach. BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms. In *IEEE International Conference on Cloud Engineering, IC2E 2021 , San Francisco, CA, USA, October 4-8, 2021*, pages 1–8. IEEE, 2021. doi:10.1109/IC2E52221.2021.00014.

20 Ryan Hancock, Sreeharsha Udayashankar, Ali José Mashtizadeh, and Samer Al-Kiswany. OrcBench: A Representative Serverless Benchmark. In Claudio Agostino Ardagna, Nimanthi L. Atukorala, Rajkumar Buyya, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Gargi Banerjee Dasgupta, Fabrizio Gagliardi, Christoph Hagleitner, Dejan S. Milojicic, Tuan M. Hoang Trong, Robert Ward, Fatos Xhafa, and Jia Zhang, editors, *IEEE 15th International Conference on Cloud Computing, CLOUD 2022, Barcelona, Spain, July 10-16, 2022*, pages 103–108. IEEE, 2022. doi:10.1109/CLOUD55607.2022.00028.

21 Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless Cold Starts and Where to Find Them, 2024. arXiv:2410.06145.

22 Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Nicholas 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 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*, pages 443–458. ACM, 2023. doi:10.1145/3620678.3624783.

23 Vasileios Katevas, Georgios Fatouros, Dimosthenis Kyriazis, and George Kousiouris. Embedding automated function performance benchmarking, profiling and resource usage categorization in function as a service DevOps pipelines. *Future Gener. Comput. Syst.*, 160:223–237, 2024. doi:10.1016/J.FUTURE.2024.05.051.

24 Jeongchul Kim and Kyungyong Lee. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama, editors, *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, pages 502–504. IEEE, 2019. doi:10.1109/CLOUD.2019.00091.

25 Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. FaaSdom: a benchmark suite for serverless computing. In Julien Gascon-Samson, Kaiwen Zhang, Khuzaima Daudjee, and Bettina Kemme, editors, *14th ACM International Conference on Distributed and Event-based Systems, DEBS 2020, Montreal, Quebec, Canada, July 13-17, 2020*, pages 73–84. ACM, 2020. doi:10.1145/3401025.3401738.

26 Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: an opportunistic caching system for FaaS platforms. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 228–244. ACM, 2021. doi:10.1145/3447786.3456239.

27 David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: characterization and optimization. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 757–770. ACM, 2022. doi:10.1145/3470496.3527390.

28 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 2019, Columbus, OH, USA, October 12-16, 2019*, pages 1063–1075. ACM, 2019. doi:10.1145/3352460.3358296.

29 Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan 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 Ada Gavrilovska and Erez Zadok, editors, *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

30 Roberto Starc, Tom Kuchler, Michael Giardino, and Ana Klimovic. Serverless? RISC more! In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies, SESAME 2024, Athens, Greece, 22 April 2024*, pages 15–24. ACM, 2024. doi:10.1145/3642977.3652095.

31 Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*, pages 51–62. IEEE, 2021. doi:10.1109/IISWC53511.2021.00016.

32 Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 559–572. ACM, 2021. doi:10.1145/3445814.3446714.

33 Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking behind the curtains of serverless platforms. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

34 Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011. URL: https://www.worldcat.org/oclc/262433473.

35 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 Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 30–44. ACM, 2020. doi:10.1145/3419111.3421280.

A Database format of FaaSLoad

FaaSLoad stores all its data to a PostgreSQL database, for standardized data management.

There is a central `runs` table that stores information directly related to an invocation at the FaaS platform-level: its ID in the platform, its start and end time, etc. The second most important table is `resources`, a TimescaleDB timeseries-optimized table for data collected by the function monitor. This table references the `runs` table, and stores all the values of all collected performance counters, resource usage, etc. for every run.

The table `runs` references a `functions` table that stores identifying data about the functions invoked when injecting the workload. In addition, a `parameters` table stores parameters of function invocations (specifically needed when generating a dataset, where parameters are set randomly by FaaSLoad). Then, a table named `results` stored plain results of function invocations, i.e., the values returned by the executions of the functions. This can be used by customized functions to return internal information (see Section 5.3 for an example of this). Both `parameters` and `results` reference the table `runs`.

Additionally, the table `dockeroperations` stores events related to function instance management by the FaaS platform; e.g., creating a new instance, etc.

B Workload trace format of FaaSLoad

FaaSLoad's trace format is very simple, yet comprehensive. An example is given in Listing 1.

Listing 1 Example of FaaSLoad injection trace. A header gives meta information, then injection points are listed.

```
user001 dataset_gen/sharp_blur  1024
0.326   3.jpg   sigma:56.256303360401
2.75    5 .jpg  sigma:39.14279952944534
0.061   1.jpg   sigma:72.80070853800676
```

The first line is special, and gives about the function its owner, its name, and its memory allocation. The user name is important, since the same program could be executed under the same function name but under two different users, to execute it under different memory allocations (thus the memory amount in the header). It is also used to simulate different user profiles with regards to memory, as described in Section C.

After this simple header comes workload injection points, i.e., instructions for function invocations. Each line is a point, listing:

- the time to wait before the invocation, relative to the previous one;
- an optional input filename;
- an optional list of other named parameters;

Then, an injection point can optionally specify an input filename and named parameters that will be passed to the function. The special handling of an optional input filename comes from FaaSLoad's original use case: generating a dataset of invocations of functions over varied inputs. While in practice this parameter is just passed to the invoker function like other, named parameters,[12] it receives special handling because FaaSLoad, when executed to generate a dataset of invocations, must cover all provided inputs of the kind expected by the function. This is interesting for FaaSLoad to consider as first class because many FaaS functions follow a process of transforming input data.

As can be seen, the injection trace format is very simple. This is a feature intended to allow writing any trace generator without added complexity on the trace format, and to allow easily auditing such resulting traces.

C Synthetic workload generator

The easiest way to obtain injection traces for FaaSLoad is of course to synthesize them. While the open trace format lets anyone write their own generator, FaaSLoad comes with a trace builder that synthesizes traces from user specifications, written as YAML files. An example of a user specification is given in Listing 2.

First, a user specification defines a memory profile. This memory profiles expresses how well the user sets the memory limit to its functions:

smart sets the memory allocation of its functions to exactly the required amount;

average sets the memory allocation to the **smart** value, multiplied by the added average increase observed in FaaS platform, which is 1.6 times [8];

maximum sets the memory allocation to a constant maximum allowed by the FaaS platform.

[12] The filename is passed to the invocation under the fixed named `object`.

Listing 2 Example of user specification used by FaaSLoad's synthetic workload generator.

```
users:
  user001:
    memory-profile: average
    nb-inputs:
      image: 10
      audio: 10
    actions:
      number: 3
      matches:
        - number: 2
          match:
            runtime: ``nodejs''
            annotations:
              input_kind: image
          distribution: poisson
          rate: 0.3
        - number: 1
          activity-window:
            from: 120
            to: 180
          match:
            docker: ``python''
            annotations:
              input_kind: audio
          distribution: uniform
          rate: 0.5
  user002:
    memory-profile: maximum
    nb-inputs:
      image: 5
    actions:
      - name: ``faasload/sharp_resize''
        times: [2, 12, 22, 32, 42, 52]
```

Using this feature, one can easily generate traces with the same characteristics, but observing the impact of memory overallocation on the FaaS platform.

Then, the user specification gives the number of available inputs for each kind that could be requested by functions, to generate input filenames, as described in Section B.

Finally, the specification expresses which functions are invoked, at which times. This expression is very flexible:

- fixed selection of functions with exact IATs;
- fixed selection of functions with random IATs following statistical distributions;
- varying in selection of functions based on criteria (name, runtime, annotations, etc.) with random inter-arrival times following statistical distributions.

With such specifications, one can execute FaaSLoad multiple times with different traces generated from the same user specification, to check reproducibility of experiments.

To cover another use case, user specifications also allow to give activity windows of functions. Functions will only be invoked during their respective activity window. This is useful to experiment with the behavior of given functions when other functions suddenly appear, and may contend for resources and interfere in performance levels.