Vrije Universiteit Amsterdam


OpenDC Serverless 

Bachelor Thesis

---

# OpenDC Serverless : Design, Implementation and Evaluation of a FaaS Platform Simulator

---

**Author:** Soufiane Jounaid (2617442)

| | |
|---|---|
| *1st supervisor:* | prof.dr.ir. Alexandru Iosup |
| *daily supervisors:* | ir. Erwin van Eyk, Georgios Andreadis |
| *2nd reader:* | ir. Jesse Donkervliet |

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

September 22, 2020

# Contents

## Abstract

Function-as-a-Service (FaaS) pertains to a recent advancement in cloud computing known as serverless computing. A group of technologies provided as "services" that shift the responsibility of provisioning resources to the cloud operator and offer a fine-grained cost model. Despite the growing popularity of FaaS within the research community, evaluating the performance and cost of different resource management, scheduling, and provisioning policies remains a difficult endeavor. Conducting experiments in the cloud is costly and usually yields unpredictable results due to the underlying hardware heterogeneity of cloud infrastructures. Whereas previous work has focused on providing open-source FaaS implementations, the requirements for conducting a custom practical systems study in FaaS remain steep both intellectually and financially.

To render practical systems research in FaaS more accessible, we propose in this work OpenDC Serverless: an expandable trace-based simulator that provides a toolkit for modelling and testing custom FaaS patterns. The simulator exposes custom interfaces for the implementation of resource allocation, management, and scheduling policies. It further supports the modification of its core architectural components. Moreover, included in the toolkit is an experimentation framework that specifies an easy-to-convert-to trace format, facilitating the exchange of inputs in the community. OpenDC Serverless also provides an interface for conducting controlled and repeatable experiments. It allows checking each experiment with fine granularity, through a metric monitoring system that allows tracking of detailed metrics in the simulation.

We prototype OpenDC Serverless and conduct with it representative experiments. We showcase the simulation performance of OpenDC serverless in a reproduction experiment that involves the optimisation of a real FaaS workload through the use of a custom operational policy; The results indicate that the simulator produces valid results. We then highlight the ability of OpenDC Serverless to explore and characterize real scenarios with an experiment on simulating cold-starts in AWS Lambda.

# 1 Introduction

Serverless computing is a relatively new cloud-computing paradigm that is becoming increasingly popular. Amidst the increasing complexity of cloud offerings (briefly discussed in a Forbes article by Joe McKendrick [3]), serverless computing aims to provide a simplistic "swiss knife" for cloud developers. By offering compute, storage, and analytics services that aim to reduce a number of challenging back-end endeavors, the emerging serverless technologies aim to shift the developer's focus primarily towards the business concerns of his application. Moreover, serverless technologies also aim to increase the accessibility of the cloud for new users, by providing simpler interfaces to cloud resources than current Platform as a Service (PaaS) technology. The attractiveness and rising popularity of the serverless paradigm promises business growth and prosperity for both cloud providers and developers as outlined first by researchers from the SPEC RG Cloud group [52], [51], and also by researchers from Berkeley [33] in their respective visions for serverless computing.

New technologies are considered part of the Serverless family of cloud services when they abstract operational concerns, such as resource provisioning and load-balancing from the user, subject to additional properties. provide an event driven interface, and charge users at a much finer granularity than traditional Infrastructure-as-a-Service (IaaS), such as Amazon EC2 [7].

The serverless approach to cloud computing holds good promise [51]. It simplifies the aspects related to cloud budget management in IT companies, by reducing various costs associated with developing rigid back-end infrastructures and, by closely tracking actual resource usage. In addition, the serverless approach shortens an application's road to deployment, by cutting down the time spent on back-end development, enabling faster profit, especially for smaller companies.

The landmark technology of this family of services is Function-as-a-Service (FaaS), an easy-to-use, event-driven interface for developing and deploying cloud applications, coupled with a fine-grained monitoring and billing model. The premise is simple. Users upload their functions to the cloud, which get triggered by HTTP requests and/or other types of events (e.g. timers, messages). Moreover FaaS aims to eliminate most of the hurdles in the developer's path that are unrelated to the business logic of his application. It achieves this by handling operational concerns for the reduced granularity of functions. In practice, each function is independently provisioned, scales automatically and receives invocations through the cloud provider's own messaging infrastructure. Consequence of this operational model, the cloud provider can and does charge the developer in very small increments of function execution time (e.g., only 100ms in AWS Lambda); this is much finer grained than conventional billing, and makes the FaaS model less wasteful [51].

Most major providers such as AWS Lambda [9], Microsoft Azure [17], and Google Cloud [13] already offer prominent FaaS services used by a range of different companies [10]. A review of serverless use cases and their characteristics by Eismann et al. [27] has collected and characterized a total of 89 use cases. In the report, Eismann et al. identified various real-world applications types ranging from monitoring and background tasks to scientific and business-critical applications, thus, showcasing the diversity of domains where serverless use cases are emerging. In conclusion, with its innovative outlook on simplifying cloud computing, and a projected 8 billion dollars market size by 2021 [2], serverless computing is set to put a strong foothold on the software development market.

### 1.1 Previous work

The fast growth of FaaS raises an important need for research in the field. However, because of the technology's young age, high rate of growth, and its existing production platforms revealing little information about their operation and architecture, much of the existing work in the community is divided into two prominent main directions: (1) designing custom prototypes of FaaS platforms [24], [40] and (2) benchmarking studies with the goal of learning how existing FaaS platforms operate [38], [53].

We also identify a 3rd small set of studies which have set the community's interest on establishing consensus on an open source architecture for FaaS computing systems. Most notably, the SPEC RG Cloud group's FaaS reference architecture [49] is the first systematic approach to designing FaaS systems that captures the main architectural and operational implications of FaaS systems.

Lastly, Hongseok et al. have explored the simulation of a distributed cloud organization of FaaS in their study [32]. While their work could be considered the first opening on FaaS simulation, it only explores the specific scenario of simulating FaaS in a geo-distributed computing mode and thus pertains more to the field of parallel and distributed computing. Moreover, the study is mainly focused on highlighting the correlation between cost and the execution location of a function in a distributed cloud organization of FaaS and thus does not explore the workload aspect of FaaS simulation. Therefore, Hongseok et al.'s work cannot be considered the first full-fledged FaaS simulator.

### 1.2 Problem statement

Despite the community's efforts to provide open-source implementations and architectures of FaaS (e.g. OpenLambda [29]), exploring, testing and evaluating different configurations of FaaS platforms remains challenging. This issue is due to the steep requirements of conducting a practical study on FaaS cloud systems, which include:

1. **Implementation availability.** The first requirement for such studies is an adequate implementation or representation of FaaS systems. While this was a prominent problem during the technology's very beginnings when little information was known about what lies behind the curtains of serverless, open-source initiatives by the community such as Apache OpenWhisk [8], OpenFaaS [22], Kubeless [15], Fission [18], and others have helped lift some of the obscurity surrounding the technology's functioning.

2. **Implementation complexity.** The second requirement relates to the complexity of configuring FaaS frameworks. The existing open-source implementations of FaaS are all specific to a certain level and are usually not simple nor standard to configure for researchers that are not actively involved in the serverless scene. Most of these frameworks have different container (function execution environment) orchestration schemes and different levels of architecture flexibility as described in Table 1 and Section 2 of Mohanty et al.'s evaluation of open source serverless computing frameworks [41]. Overall, this issue reduces the accessibility of implementing and evaluating of custom operational policies in open-source FaaS frameworks.

3. **High funding requirements.** The actual cost of running experiments associated with these studies can become quite high. Even if requirements 1 and 2 are satisfied, running experiments requires having a meaningful deployment environment (VM's and containers with enough amounts of CPU/RAM and a solid networking interface) for the chosen framework and workload. In addition, deploying these systems often requires domain expertise. This can be particularly costly for large-scale experiments and can restrict the choice of workload due to insufficient resources.

4. **Performance unpredictability.** The fourth requirement relates to the performance variation and predictability of cloud experimentation environments from services [31] to networking and other resources [47]. Cloud platforms typically present three issues in experimentation: performance unpredictability, lack of control over influencing factors and limited repeatability. These issues moreover relate to the multitude of uncontrollable factors that come in play in cloud platforms and are already well documented by studies such as Leitner et al.'s work on hardware heterogeneity and multi-tenancy (multiple users provisioned with the same cloud resources simultaneously) in IaaS [36]. Furthermore, hardware heterogeneity and multi-tenancy are also present in serverless as respectively shown in sections 4.4 and 6 of [53]).

5. **Workload availability.** Although analyzing the performance of systems when subjected to relevant workload (or workload traces) is a common approach in the community, there are very few public FaaS workloads/traces currently available. The scarcity of real-world serverless traces complicates the testing and validation of custom operational strategies in FaaS. Nevertheless, we commend Shahrad et al. for their public release of the first real-world Azure Functions trace alongside their a characterization study in [46].

As such, serverless computing currently suffers from the same "computing for the 1% problem" that IaaS technologies endured during the early days of cloud computing. The problem relates to the difficulty of simultaneously satisfying all the above requirements for a conclusive practical study in FaaS, essentially reserving FaaS systems research to the few who can satisfy all the above requirements. Simulation technologies such as OpenDC [30] and CloudSim [32] have contributed to reducing the issue in IaaS, as they allow cloud researchers and developers to evaluate the performance of their provisioning and service delivery policies in a repeatable and controllable environment free of cost in addition to being open-source.

### 1.3 Research questions

The aim of this work is to provide a scientific instrument that can help bring the exploration of different FaaS architectures and operations to the other "99%". We ask four main research questions:

**RQ1** *How can a FaaS simulator model the essential elements of FaaS architectures?* This question is important in addressing requirement 1 from the previous section in the sense that, to provide an adequate experiment framework for FaaS studies, we need to model the basic essentials of a FaaS architecture while avoiding the use of proprietary architectural patterns. Hongseok et al. laid valuable ground-work on this matter by modelling a number of these architectural components in their study on simulating a distributed cloud organization of FaaS. Moreover, The SPEC RG Cloud group's reference architecture for FaaS presents a systematic approach to designing FaaS platforms, further reducing the complexity of this challenge and thus largely contributing to a solution.

**RQ2** *How can a configurable FaaS simulator allow for exploration and testing of different or custom operational policies?* This question tackles requirement 2 from the previous section. With this question, we contemplate whether simulation is capable of modelling the complex operational concerns associated with FaaS and furthermore of offering a framework for researchers/cloud developers to test and develop their custom solutions to operational problems. Shahrad et al. have used in-house simulation to test their custom operational policy before conducting real-world tests and deploying the technology on Azure in [46].

**RQ3** *Can FaaS simulation reduce the temporal and material costs of experiments in the field?* This question tackles requirement 3 from the previous section. IaaS simulators (e.g. OpenDC, CloudSim) allow the researcher to conduct low-cost experiments on simulating VMs (Virtual machines), therefore, we ask ourselves whether the same outcome is achievable in FaaS.

**RQ4** *How can simulation of FaaS platforms provide a controllable environment for repeatable experiments, but maintain the accuracy of results provided by real-world experiments?* This question relates to the fourth requirement for FaaS experimentation. A simulator could choose to safely ignore or model performance variation in cloud platforms, therefore we ask ourselves which steps we should take to conceive a repeatable experiment environment with meaningful and controllable variations.

By combining the above collection of research questions, we simply ask ourselves whether we can design, implement and evaluate the first configurable and extendable trace-based FaaS simulator. For the purposes of this work, the fifth requirement from Section 1.2 is fulfilled by the open-access to the Microsoft Azure Functions trace, however, FaaS trace scarcity remains a large obstacle in the field of FaaS system research. With this work, we also aim to provide incentive for the release of real-world production FaaS workload traces.

## 1.4 Research approach

To address the research questions, we develop in this work, OpenDC Serverless, the first trace-based, configurable open-source FaaS simulator. Through this work, we aim to provide an early framework for education and research in FaaS simulation that satisfies the four first requirements of Section 1.2, accordingly with OpenDC's vision [30].

Based on the SPEC RG Cloud reference architecture for FaaS [49], the simulator models the essential architectural components required for a basic FaaS system in an extendable and customizable fashion. We use clear separation of concerns between all the different architectural layers to allow users to modify whichever component they desire with no design overhead. Moreover, we design several configurable and expandable interfaces to allow the users to implement custom operational policies

Furthermore, the simulator provides an experiment framework designed to be used and configured by students, cloud researchers, cloud providers and curious third-parties alike. This framework allows for input of operational parameters (e.g. policies), pricing models, delay models, seeds and more in a centralized input manner using configuration files. Furthermore we designed a custom trace format with simplicity and adaptability as our priorities, thus allowing future real-world traces to be easily converted to it.

To show that we can extract valuable data from experimentation using the simulator, we present two experiments centered around simulating and comparing custom cold start mitigation policies.

The first experiment we conduct is a reproduction of the Azure Hybrid Histogram cold start mitigation policy experiments from Shahrad et al.'s recent study on characterizing and optimizing real Azure Functions workloads [46]. With this first experiment, we aim to show that we can obtain data comparable to real-world experiment results through exploring and testing custom operational policies in the simulator.

The second experiment we conduct, is a 24 hour long experiment where we execute real workloads with various invocation patterns both in the simulator, and in AWS Lambda. With this second experiment, we aim to show that we can, to an extent, simulate cold starts in AWS Lambda and possibly evaluate the platform's cold start mitigation policy.

We then subsequently show that the simulator can greatly reduce experimentation costs through a comparison of the resources it took us to simulate the reproduction experiment and the resources used for the real-world run in terms of both time and money.

Lastly, we highlight the control and repeatability aspect of the simulator by showing that we can obtain repeatable results with controlled variations using proper seeding of randomization methods on the multiple iterations of our reproduction experiment.

In conclusion, we propose in this work the first FaaS simulation study to extract valuable data from real-world Faas traces. But the work does not end here; we present a number of open limitations in FaaS simulation, which spotlight the importance of pursuing research on FaaS simulation.

## 1.5 Structure of this thesis

The first half of this thesis encompasses the conceptual design contributions. We begin by explaining the backgrounds and concepts needed to understand the design in Section 2. We present the design of the simulator in Section 3, in which we first provide an overview of the system and then explore the main features in depth. In the second half of this thesis, we present the practical contributions. We begin by describing our experiment setups for both the Azure Hybrid Histogram experiment reproduction and the AWS Lambda cold start simulation in Section 4. Subsequently, we present the results obtained through both experiments in Section 5 and moreover provide small summaries of the key findings at the end of each of the experiment's results. Lastly, we end this work with an analysis of open limitations in Section 6 and a conclusion containing a summary of this work's main contributions and future research directions in Section 7.

# 2 Background Concepts and Models

In this subsection, we present the key concepts needed to understand this work. First, we define the Function-as-a-Service (FaaS) terminology used throughout this work. Then, we highlight and briefly define some of the the components of the SPEC RG Cloud FaaS reference architecture that are used for the simulator's design.

## 2.1 Function-as-a-Service terminology

In FaaS, the user configures his function by first uploading his code, choosing a certain amount of provisioned memory and then setting an invocation trigger (e.g. HTTP request). Then, every time the function is triggered, the provider spins up an execution container, allocates memory to it and chooses the duration of its lifetime. In this subsection we define the terminology used in the latter sentences.

**Function triggers.** FaaS platforms offer various ways of triggering function invocations. These events can vary from simple HTTP requests and timers to more complex events such as event orchestrations (e.g. AWS Step Function) or storage orchestrations (e.g. Amazon DynamoDB).

**Execution container.** An execution container embodies the run time environment of a cloud function. When started, the execution container loads the function code, adequate compiler or interpreter and required runtime libraries, then, it prepares to receive invocations for whichever function it loaded.

**Function provisioning.** In popular FaaS platforms, when a user creates a function, he chooses the maximum amount of provisioned memory for it. If during execution, the function exceeds the provisioned memory, the function can be terminated by the FaaS platform.

**Cold start, warm start.** Cold starts occur when an invocation is received for a function that has no available execution container. The time spent deploying the container, retrieving the function's code and preparing the runtime environment is referred to as the cold start delay. Warm starts represent the opposite scenario. If an invocation is received for a function that has an available execution container, the invocation is simply routed to the said container, thus skipping all the delay caused by initialization procedures.

**Cold start mitigation strategy.** A cold start mitigation strategy tweaks execution container lifetime and tries to optimize the trade-off between reducing cold starts and the resources consumed by idle execution containers. A typical strategy operates by setting a fixed execution container lifetime. Other strategies adjust execution container lifetime dynamically, and some strategies keep a pool of warm execution containers.

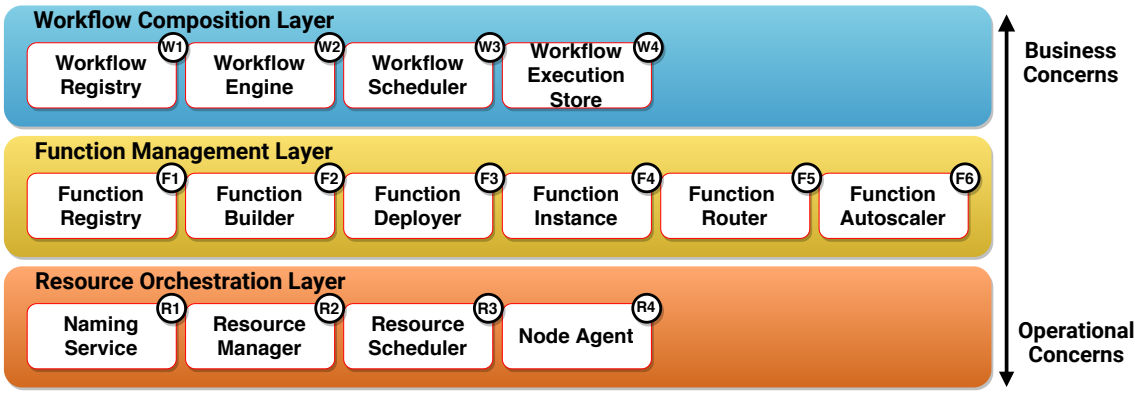



Figure 1: The SPEC-RG reference architecture for FaaS platforms reproduced from Figure 1 of [49].

## 2.2 SPEC RG reference architecture for FaaS

The SPEC RG reference architecture for Function-as-a-Service (FaaS) [49] provides one of the first systematic approaches to designing FaaS platforms. Figure 1 shows the three hierarchical layers of the reference architecture. For the simulator's design, we chose to model the core components required for a generic FaaS platform. In this subsection, we present and briefly describe the architectural components and operational patterns used in the simulator's design in a manner that highlights their purpose for this work.

### 2.2.1 Architectural components

For this work, we primarily set the spotlight on the Function Management Layer of which we model the following components:

F4 Function Instance: the Function Instance is an execution container for functions. Each instance is specific to a function and can be spawned multiple times concurrently.

F3 Function Deployer: the Function Deployer takes care of creating new Function Instances. This process involves loading the function's code and preparing the function's runtime environment. Once it has retrieved all the required parameters for the deployment, the deployer delegates the actual deployment to the Resource Manager/Scheduler.

F5 Function Router: the Function Router takes care of routing a function's invocations to one of its Function Instances. this component can request deployments from the Function Deployer if no instance is available .

Moreover, since we want to sketch together a lightweight FaaS system, we chose to only model some inter-dependent components from the other layers.

R2 Resource Manager: the Resource Manager monitors and manages the state of virtual machines and execution containers (Function Instances). This component is

responsible for making sure the resources are in an adequate state, for example if a Function Instance has finished a job, the manager should de-allocate its memory.

R3 Resource Scheduler: the Resource Scheduler is responsible for choosing on which resources Function Instances are deployed. The scheduler is moreover responsible for taking appropriate actions (e.g. start or terminate resources...) to reach an optimal state of resources.

W3 Workflow Scheduler: the Workflow Scheduler is responsible for feeding invocation requests to the Function Router, it decides which functions run when.

We chose not to model both the Function Registry and Builder due to the small amount of logic these components add to the simulation. Instead we can incorporate their effects in the Function Deployer.

As for the Function Autoscaler, we choose to model it in future extensions of this work. Moreover, since we are not planning to model the physical aspect of resources, we chose to omit the use of Node Agents in the resource orchestration layer.

Lastly, when it comes to the workflow composition layer, we choose to only model the Workflow Scheduler for the reason that we only model sequential workflows, meaning we compose functions in the order they were executed. Therefore, the scheduler can encompass both the Workflow Engine's and Registry's logic.

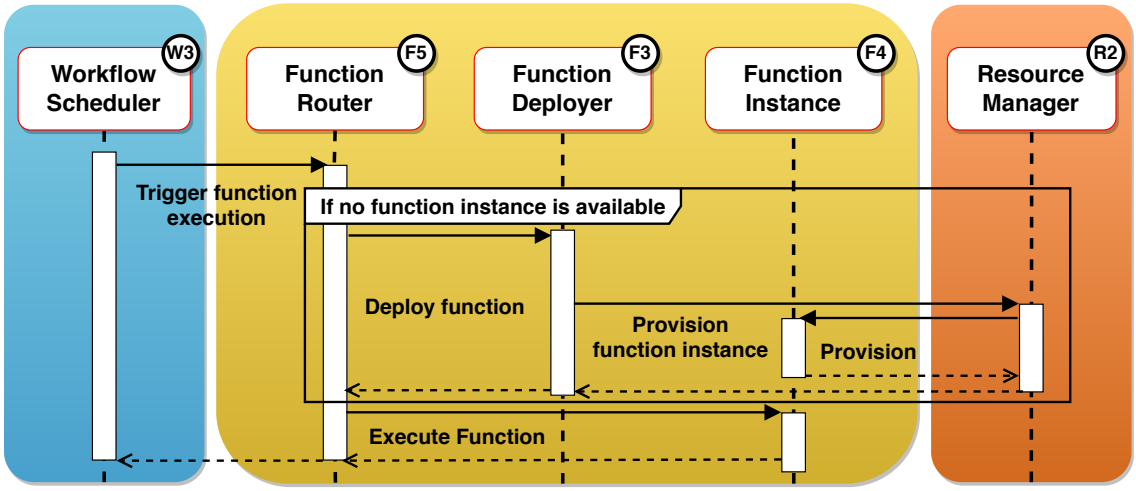



Figure 2: Function execution pattern reproduced from Figure 3 of [49] (altered to only model the components used for this work and not include the workflow execution pattern).

### 2.2.2 Operational patterns

Since we are not using the entirety of the SPEC RG reference architecture for FaaS, we adapted the function execution operational pattern to our use case for this work. Figure 2 shows our altered version of the function execution pattern. in this subsection, we explain some of the changes we made to the original pattern and explain why the original parts are not needed.

When explaining our choices of architectural components, we mentioned that we do not model the Function Builder and Registry but instead chose to incorporate their impact in the Function Deployer. In this sense, the simulated Function Deployer models the delay impact caused by both the Function Builder and Registry. Moreover, we omit the workflow execution pattern presented in the original figure since our simulation use case is entirely sequential and trace-based.

| Timestamp (ms) | Invocations | Average execution time per invocation (ms) | Provisioned CPU per invocation (Mhz) | Provisioned memory per invocation (mb) | Allocated CPU per invocation (Mhz) | Allocated memory per invocation (mb) |
|---|---|---|---|---|---|---|

Table 1: OpenDC Serverless function history file format.

# 3 Design of the Simulator

We start this design section by presenting the conception process of the simulator's trace format in Section 3.1 followed by the functional and non-functional requirements of the simulator's design in Section 3.2. After defining all its requirements, we present a high level overview of the simulator's architecture in Section 3.3, which we then explore by walking through a typical execution of the simulation in Section 3.4. Lastly, we offer a deeper look into request routing in Section 3.5, function instances in Section 3.6, resource management and scheduling in Section 3.7 and finally, monitoring in Section 3.8.

## 3.1 Conception of the OpenDC Serverless trace format

At the start of this project's design-cycle, there were no publicly available Function-as-a-Service (FaaS) traces. We therefore had to first design a trace format to be a precursor for the simulator's design. Our requirements for designing a trace format were centered around modelling the essential metrics recorded in a FaaS workload. A complicated trace with a multitude of different fields would probably end up containing more information than needed and would complicate the conversion process of real-world traces to the OpenDC serverless trace format. To avoid such an issue, we took inspiration from a pre-release description of the Azure Functions trace format [12].

In the process of designing the trace, we sketched a small collection of requirements presented in the list below

1. The full trace should be a directory of CSV files, with each file representing the history of a single function.

2. Each history file should only contain the essential metrics produced by FaaS workloads (timestamp, invocations, execution time, allocated resources). Moreover, the timestamps should be consistent across all functions.

3. For scalability reasons execution time and allocated resources should be averaged. This change in granularity should allow the history file's size to scale reasonably.

4. The trace format should be able to be mapped to the Azure Functions trace. (The only information originally available about the Azure Functions trace was a small bullet point list of the what the traces will contain on the Azure dataset 2019 Github page [12]).

After going through a few revisions of the trace format, we ended up with the small 7 column function history file format presented in Table 1.

With this format, we satisfy our requirement of having all the essential metrics in addition to adding function provisioning fields (provisioned cpu and memory) which represent the provisioning values specified by the user when creating the function (later on used to determine cold start duration in Section 3.6). Furthermore, the trace format averages execution time, provisioning and allocation metrics per invocation to satisfy the scalability requirement. Finally, we present in the Section 4.2.2 how our trace format satisfies the requirement of mapping to the real-world Azure Functions trace.

## 3.2 Requirements

The simulator's design requirements are grouped in 2 main categories. **Architectural requirements** (where requirement x under this category is labeled A.x) and **Simulation requirements** (labeled S.x). Most of the architectural requirements are directly extracted from the reference architecture [49]. Together they form the minimum core specification of components in a Function-as-a-Service (FaaS) infrastructure. Simulation requirements on the other hand, are mostly composed of customizable platform specifics such as cost and metric monitoring, as well as system management. In this section we present below a list of these requirements followed by a small analysis of stakeholders and use cases

A.1 The system should model Virtual machines (VMs), and allow for input custom VM layouts.

A.2 The system should model execution containers and their deployment.

A.3 The system should support customizable delay modelling (custom cold starts, metatdata lookup delay, etc...).

A.4 The execution container should support invoking and termination.

A.5 The system should support customizable routing of invocations from the trace to available execution containers. Moreover, the system should deploy a new execution container if none are available.

A.6 The system should support customizable resource management and allocation.

S.1 The system should support centralized configuration input, i.e. a configuration file with all customizable parameters should be passed to the simulator.

S.2 The system should sequentially schedule invocation requests from the trace and keep track of trace time.

S.3 The system should support both trace and simulation configuration input and parsing.

S.4 The system should support tracking customizable metrics from the simulation and moreover organize them in a report.

S.5 The system should support tracking the cost of the simulated computations using different customizable pricing models.

S.6 The system should compile a report of the recorded metrics at the end of the simulation.

S.7 The system should support repeatable executions and controlled randomization through the use of deterministic seeding methods.

**Use cases.** We manage to capture in the list below a number of research, education and business related use cases for our system

1. Users can use this tool to reproduce and evaluate research studies conducted in real-world FaaS environments. We show a small reproduction study using our tool in Sections 4.1 and 4.2.

2. Users can use this tool to evaluate and scope out any faults in operational policies before their deployment on production FaaS platforms. The reproduced study experiment in Section 4.2 provides a small glimpse into this use-case.

3. Users can compare and evaluate different allocation, request routing and cold start mitigation strategies by implementing them through the simulator's interfaces.

4. Users can evaluate how different workloads run on different amounts of provisioned resources in terms of delays, cold starts, resource utilization, etc...

5. Users can explore and characterize FaaS performance in different data center environments. This would help characterize what a good environment for FaaS workloads could be.

6. Users can extract workload characteristics by analyzing customizable metrics produced from multiple simulations. These characteristic could range from frequency of invocations to percentage of cold starts over time and more.

**Stakeholders.** We refrain from citing indirect stakeholders such as End Users, but instead mention their relation to the direct stakeholders of this technology

1. FaaS researcher: researchers often do not have full access to the platforms they are researching on. Considering that most of the current existing work is either benchmarking related or reverse-engineering of existing FaaS technologies [35] [24], the need for simulation becomes paramount in research focused on improving FaaS such as workload characterization studies and studies for delay mitigation solutions.

2. Platform provider: the FaaS provider has to ensure quality of service to its customers. Research and development is generally done in-house with largely commercial technologies such as FaaS. However, despite having full access to the actual FaaS platforms, there is often a strong need for pre-deployment simulation of the technology being researched as we see in [46]. This need arises from corporate matters such as securing funding, early demonstration of effectiveness and quality assurance.

3. FaaS user: the FaaS user develops applications for end users. While the interest of such an actor in the simulator might not be apparent, serverless application developers typically have to deal with the operational quirks of FaaS, and thus would most likely benefit from the insight a simple-to-use simulator provides about their workloads. End users would then naturally benefit from an improved quality of service (e.g. lower latency).

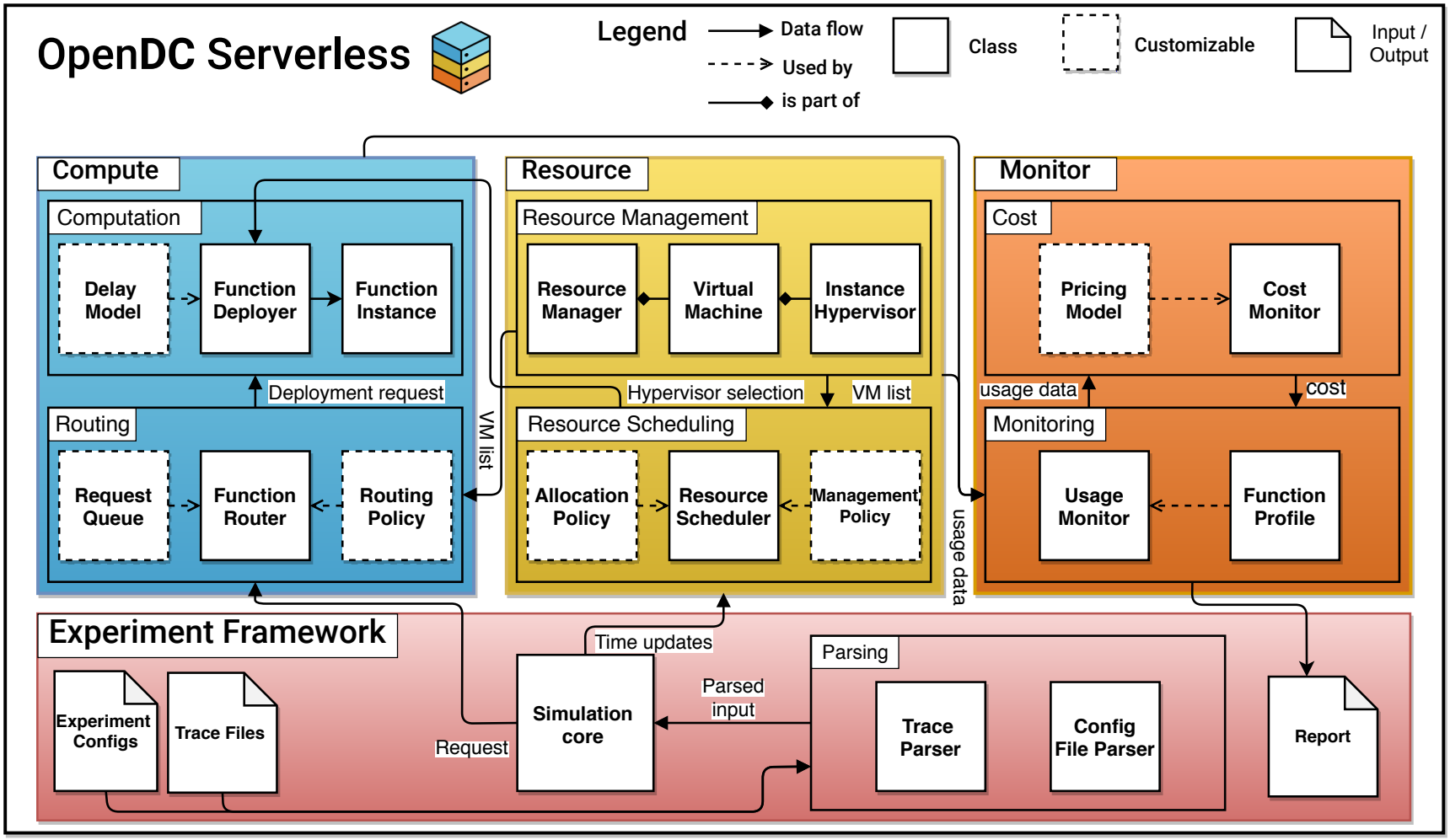



Figure 3: High-level class overview of the simulator.

### 3.3 High-level overview of the system

Figure 3 shows a high-level overview of the simulator. In this section, we will go over every component in the system, citing its purpose briefly, highlighting which requirement it satisfies and which section presents more specific details.

**Experiment framework: inputting and parsing experiment configurations and trace files.** The simulator's trace comes as a directory of CSV files, each representing the history of a single function. Moreover, each file contains 7 columns which we explore in more detail in Section 3.1. The simulator also takes as input a directory of experiment configuration JSON files, each containing all the necessary parameters to for one distinct simulation. These parameters include but are not limited to: pricing model, delay model, allocation policy, resource management policy and virtual machines. Moreover, both forms of input are parsed accordingly in the parsing component. This part of the system satisfies requirement S.1 and S.2 and S.3.

**Experiment framework: simulation core.** The simulation core is the heart of the simulator, it initializes all the different FaaS services with the provided configuration parameters, seeds all the randomization methods to ensure repeatability, and then proceeds to start a loop where it sequentially schedules and sends invocation requests to the routing component in addition to providing time updates to the resource management component. This class thus serves as the clock and event feeder of the simulation. This part of the system satisfies requirement S.2 and S.7.

**Compute: routing.** The routing component is responsible for routing invocations to instances of their respective function, it operates off of a queue and uses a routing policy to determine which instance to route to. The process is described in more detail in Section 3.5. This part of the system satisfies requirement A.5.

**Compute: computation.** The computation component envelops all the logic surrounding the function instance, our model of a FaaS execution container. The instance is deployable through the use of the function deployer. Furthermore, deployment delays (cold starts, lookup delay, etc...) are modelled using a delay model. This part of the system satisfies requirements A.2, A.3 and A.4.

**Resource: management and scheduling.** The resource management component models both VM's (virtual machines) and instance hypervisors through the resource manager. Similar to VMMs (Virtual machine monitors) from AWS Lambda [23], instance hypervisors model the software that enables FaaS workload virtualization on top of VM's (i.e. it manages function instances). Moreover, this component provides an interface for resource scheduling through the resource scheduler. More details are presented in Section 3.7. This part of the system satisfies requirement A.6.

**Monitor: cost and usage monitoring.** The monitoring component is responsible for keeping track of every function's metrics during the simulation. It does so through the usage and cost monitors. More details are presented in Section 3.8. This part of the system satisfies requirements S.4 and S.5.

**Monitor: report.** Once the simulation is done, the monitoring component compiles together a report containing all of the metrics recorded in a series of formatted entries. The contents of the report are presented at the end of Section 3.8. This part of the system satisfies requirement S.6.

### 3.4 A walk through a simulation

In this subsection, we walk through a simulation step by step without going into much detail. However, we first need to define the important notion of a simulation cycle. A simulation cycle refers to the time interval between two entries in the trace. a typical simulation cycle consists of formulating Invocation requests according to the trace entry, routing them to a Function Instance or deploying new instances, sending time updates to the resources, and writing all the recorded metrics onto the report. Furthermore, To know more about what each component does, refer to Section 3.3.

**Initialization.** First, the user has to input both a trace directory and one or more configuration files. Depending on how many configuration files were inputted, the experiment framework launches one or more simulation cores (Experiments) with their respective parameters (parsed from the configuration files) and a copy of the parsed trace each.

**Simulation cycle: first time update.** At the start of every simulation cycle, the simulation core issues a time update to the resource management component. The update then propagates down all the way to function instances. This is done at this moment to appropriately update the state of the resources in preparation for the handling of invocation requests.

From here on, the cycle branches into one of two different scenarios.

**Simulation cycle: scenario 1: new function instance deployment.** For every invocation request in the request queue, the routing component attempts to route it to an existing available function instance. In the event where there is no available function instance to handle the request, the routing component asks the function deployer to deploy a new instance of the concerned function. The deployer then asks the resource scheduling component to select, according to the allocation policy, an appropriate instance hypervisor to deploy the instance on. Once all the aforementioned steps are completed, the instance is then deployed and its execution is delayed for a duration chosen according to the delay model, akin to a cold start delay.

**Simulation cycle: scenario 2: routing to an existing function instance.** In the event where there are available function instances of the concerned function. The routing component routes the invocation request to an instance chosen according to a routing policy.

**Simulation cycle: second time update.** At the end of every simulation cycle, the simulation core issues another time update to the resource management component. The update again, propagates down all the way to function instances. This is done at this moment to update the state of the resources in preparation for metric logging.

**Simulation cycle: monitoring.** Throughout the simulation, the system keeps track of various function and system specific metrics. The monitoring component tracks each and every function's metrics using function profiles (discussed in more detail in Section 3.8). All of the cycle's recorded metrics are systematically written to a report file at the end of every simulation cycle.

**Termination.** Once the simulation reaches its end time, the monitoring component outputs the reports. The simulation core then proceeds to terminate all of the running components.

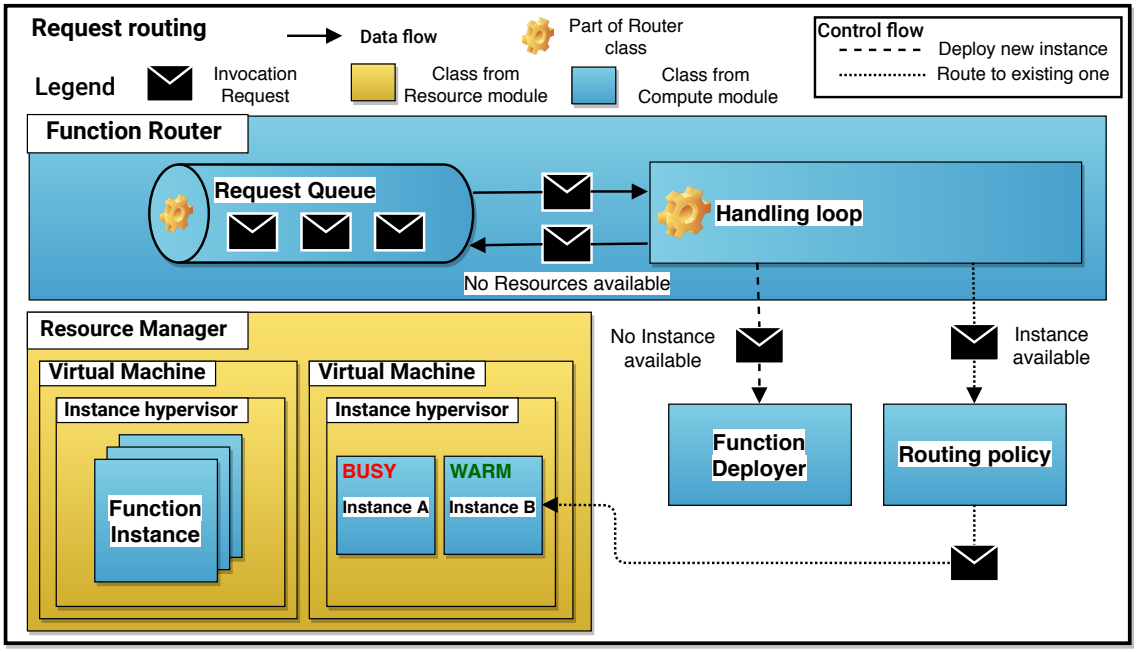



Figure 4: Overview of invocation request handling within the function router.

### 3.5 In-depth look at Routing

Request routing is an essential function of FaaS platforms. At the heart of this process is the **Function Router.** The router operates a request queue (customizable interface of a queue). Each **Invocation Request** is representative of one entry in a function trace and is composed of the following data fields:

1. Function identifier: unique id extracted from the trace file of the function
2. Number of invocations: total number of invocations specified in the trace entry
3. Time stamp: time specified in the trace entry.
4. Execution duration: this parameter represents the average execution time for each invocation in this request
5. Allocated CPU / memory: CPU (Mhz) and memory (mb) allocated to the each instance than handles a invocation from this request

To serve the requests, the router initiates a handling loop, where it goes through invocation requests one by one and for each, decides whether to route it to an existing instance, ask for an instance deployment, or, delay the request to the next simulation cycle. The process is illustrated in Figure 4.

We also include an interface to specify routing rules, which we conveniently name the **Routing Policy.** The policy specifies a set of of customizable rules for selecting an available function instance to route a request to. Once supplied with the invocation request, the routing policy returns an available instance chosen according to its specified routing logic. Some examples of routing policies are

1. The sequential routing policy which first filters the set of instances to only include instances that are available and have enough resources on their respective VMs and then returns the first instance in the filtered get.

2. The random routing policy which again filters the set of instances to only include instances that are available and have enough resources on their respective VMs and then returns a random instance from the filtered get using the simulation seed.

3. The least idle time routing policy which also performs the filtering and then proceeds to choose the available instance that has been idle for the least amount of time. This strategy allows instances that were idle for long to be terminated and prioritizes younger instances if possible.

We added this concept to the simulator to encourage research in the routing component of FaaS. A possible research challenge could be about improving branch prediction behavior of FaaS platforms. The issue is discussed in more detail by Shahrad et al. in Section 5.1 of [45].

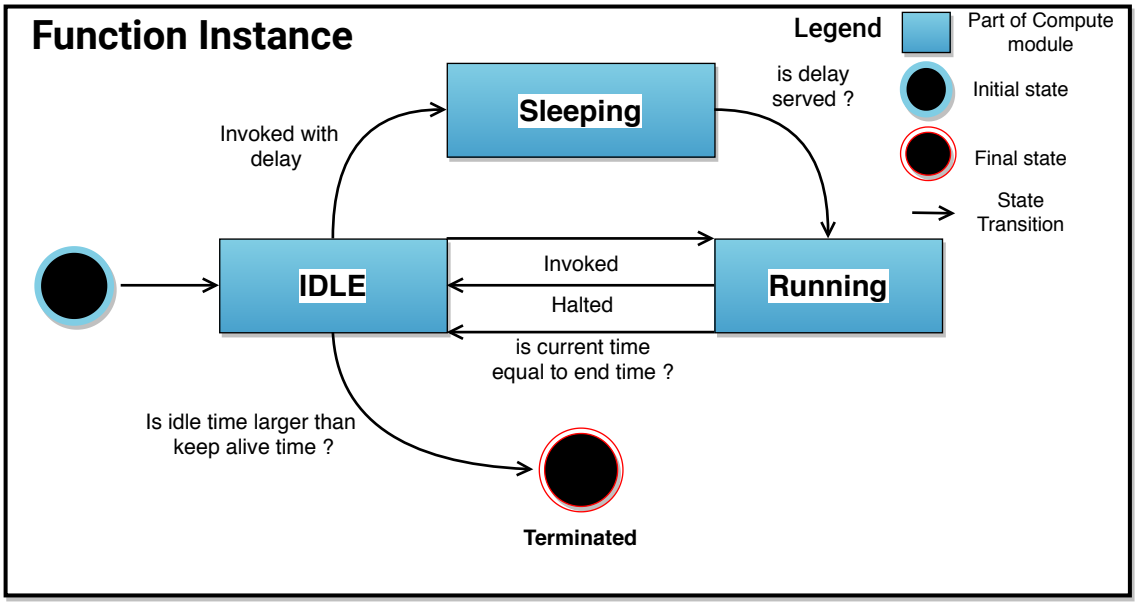



Figure 5: Function instance state machine diagram.

### 3.6 In-depth look at Computation

As part of the Computation component, we designed a simple model of execution containers. The **Function Instance** provides a small but functional API specifying methods for invoking, sleeping, halting and updating. Figure 5 outlines the different states of the function instance. Each state transition can only happen once per simulation cycle.

Once deployed by the function deployer, the instance enters the "Idle" state. When in this state, the instance is occupying a certain (configurable) amount of memory. From here on, every simulation cycle, the system proceeds to increment the instance's idle time counter until it either (1) reaches the keep-alive time limit and is then queued for termination (moves to "Terminated" state) or (2) is invoked by either the function deployer or router.

When invoked using the function deployer, the instance is set to sleep for a specified delay period (cold start) and thus enters the sleeping state. In this state, the instance still retains its provisioned resources. Moreover, the instance checks whether it has served its sleep duration during its update every simulation cycle. If served, the instance transitions to the running state and executes until a delayed end time. In the opposite case where it has been invoked without a delay (through the function router), the instance simply transitions into the "Running" state.

Once in the "Running" state, the instance is considered to be executing and can thus only transition to the "Idle" state if it has reached its execution end time or has been halted by either the resource manager or scheduler.

The last service we detail in this subsection is the **Function Deployer:** an interface for instance deployment. To model startup delays, the deployer refers to the delay model, a configurable composition of statistical distributions (distribution parameters extracted

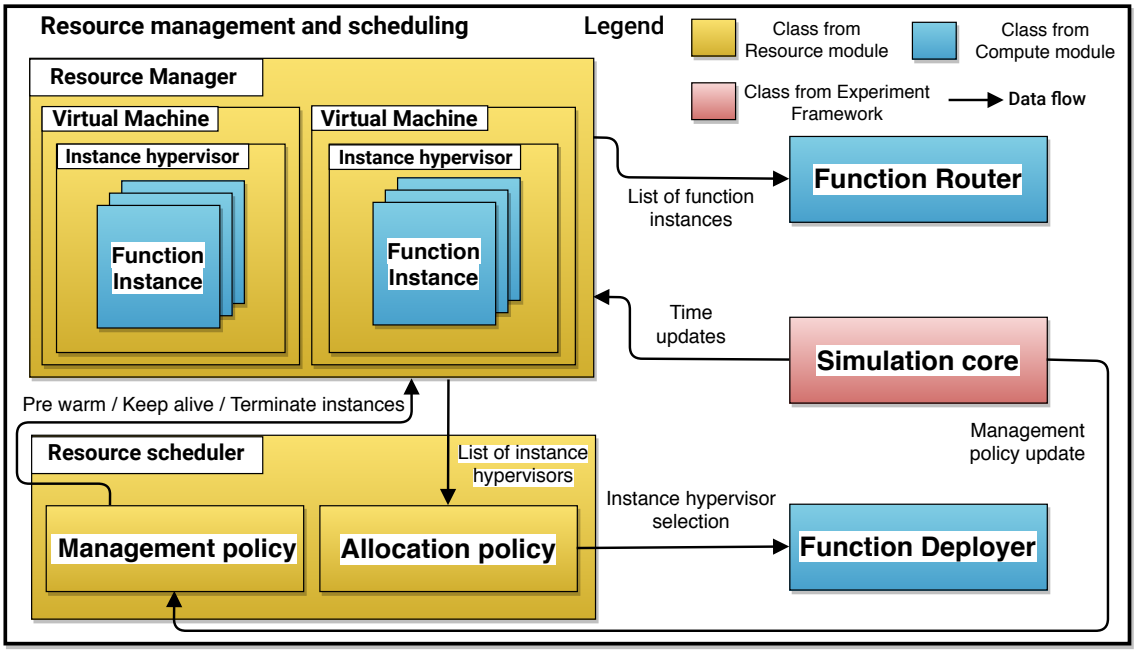



Figure 6: Overview of interactions between the resource manager, resource scheduler and the rest of the system

from Table 7 of [53]).

We went a step further as to model the cold start variations for different amounts of provisioned memory (trends extracted from this AWS Lambda experiment by Yan Cui [14]). We designed this interface with the future goal of fully modelling the various flavors of Runtime delays (Builds, loading libraries etc..), metadata and code lookup and networking delays of different platforms.

### 3.7 In-depth look at Resource Management and Scheduling

In this subsection we will detail the design of the **Resource Manager**, **Resource Scheduler** and **Instance Hypervisor.** But first, to understand the relevance of each component, we present an overview of their interactions with different parts of the system in Figure 6.

The resource manager embodies key operational logic such as relaying time updates to its sub-components, and furthermore exposes a complete API to access and manage virtual machines (VM), instance hypervisors and function instances. It is therefore used by every component in the system.

The instance hypervisor models the software that runs on top of traditional VMs in order to accommodate FaaS workloads. This concept is inspired from AWS Lambda's use of the Firecracker Virtual Machine Monitor [23]. Similar to the original concept, the instance hypervisor provides an interface for provisioning and managing large numbers of function instances. It therefore organically ties to other parts of the system such as the function deployer or the resource scheduler.

The resource scheduler operates two major operational parameters of the simulations. First, the **Allocation Policy** specifies a customizable set of rules for selecting an appropriate instance hypervisor for a function instance deployment. Identical to the routing policy, it provides a constraint-less API, leaving the user ample freedom to implement virtually any set of rules. Allocation policies start by filtering the set of instance hypervisors to only obtain ones with enough resources to accommodate the deployment, then, the policy branches to its custom behavior. Examples of allocation policies include: a sequential allocation policy which chooses the first instance hypervisor from the filtered set and the random allocation policy which instead, chooses a random instance hypervisor from the filtered set.

The second major parameter is the **Resource Management Policy:** an interface for managing the lifetimes of function instances. This concept is adapted from Microsoft Azure Functions [46]. The resource management policy specifies a set of rules for determining the following two parameters:

1. Pre-warming time: the duration spent before warming up new instances of the function. At the end of every execution, if the pre-warming time is not set to 0, the scheduler starts a timer with the pre-warming duration as a limit. Once the timer expires, the scheduler launches one function instance.

2. Keep-alive time: the duration an instance is kept alive either after it's been deployed, or after an execution (a slight difference with the original definition is that the keep-alive countdown starts after an execution and not when an execution happens). Furthermore, the scheduler enforces keep-alive during every simulation cycle by checking each function instance's idle duration counter.

The resource management policy interface however, provides a slightly more constraining API than the other policies in the system. To explain why, we first have to define the notion of idle time (IT): time duration between two successive invocation requests of the same function. Since we deem that most resource management policies will require to keep track of each function's IT, we added an update operation to the API which can be used to relay idle times to the management logic. The update operation is of course optional or useless to the functioning of certain policies (e.g. Fixed Keep Alive policy which sets the keep-alive parameter to be fixed). Therefore it does not affect any vital part of the simulation and can thus safely be omitted if needed.

One example of a popular resource management policy is the Fixed-Keep-Alive policy. It sets the pre-warming time to 0 (signalling the system to not perform any pre-warming of instances) and the keep-alive parameter to a fixed duration specified in the configuration file.

Another simple example of a resource management policy is the No-Termination (also named No-Unloading) policy. It sets both the pre-warm and keep-alive times to 0 and thus, does not terminate or pre-warm any instances.

### 3.8 In-depth look at Monitoring

To track every function's individual metrics, the **Usage Monitor** operates on a map of **Function Profiles**. Each profile contains a number of metrics, data structures and other characteristic elements. Some metrics are tracked per every simulation cycle (e.g. invocations per cycle) and thus written to the output file report at the end of every cycle, while others are tracked for the whole duration of the simulation (e.g. sum of invocations) and only displayed on the console at the end of the simulation.

Furthermore, to obtain the cost of the simulated computations, the usage monitor relays execution times to the **Cost Monitor** which then determines the cost using a configurable pricing model.

The report can contain various metrics per cycle such as invocations, cold starts, median delay duration, number of terminated instances, memory and CPU usage, etc... In total, the file output report ends up containing a row per function for every simulation cycle calculated in the following manner: $Report\ length\ (in\ rows) = Number\ of\ functions \times \frac{Trace\ duration}{Time\ interval}$.

When implementing the system, we use algorithms from [54] and [39] to minimize value storing and reduce the number of computation steps required to obtain certain values. Moreover, on an important note: attributes of function profiles are not limited to recorded metrics, they can also contain other function specific variables used by the system. We chose this centralized approach to to avoid having scattered value trackers (maps/ lists/ objects) across the system, thus improving the simulator's memory consumption model.

# 4 Experimental Setup

To appropriately showcase the real-world performance of the simulator, we incorporate real Function-as-a-Service (FaaS) platforms in our experiments. The first experiment we present in this section is a reproduction of Azure's Hybrid Histogram experiments from Shahrad et al.'s recent study on characterizing and optimizing real FaaS workloads in [46]. Throughout this first experiment, we show how we implemented a version of the Azure Hybrid Histogram resource management policy in Section 4.1. Then, we present our reproduction of the original experiment setup in the simulator in Section 4.2 followed by the results in Section 5.1.

The second experiment we present in this section will be a simulation of cold starts in AWS Lambda. Throughout this second experiment, we will record cold start occurrences during a real-world 24hour experiment in AWS Lambda with various different workloads (detailed setup in 4.3). We then record the same metric in simulations of the same workloads and compare them in Section 5.2.

## 4.1 Azure's Hybrid Histogram resource management policy

With their recent characterization study of a real-world Azure Functions workload, researchers from Microsoft have detailed in section 4.2 of [46] how they designed and implemented an adaptive resource management policy geared towards minimizing memory waste and cold starts. The policy's design is focused on overcoming the challenges caused by the strong heterogeneity of FaaS workloads. Throughout this subsection, we will detail how we implemented a version of Azure's Hybrid Histogram policy in the simulator. We will also mention the changes and assumptions we made to the implementation in addition to the challenges and difficulties we encountered along the way.

We build the policy using the resource management policy interface (detailed in Section 3.7). Therefore, we implement the update method to feed idle times (IT)[1] to the different components used in the Hybrid Histogram policy. Furthermore, the only elements exposed to the rest of system are the pre-warming and keep-alive times. From here on, this subsection is split into three parts each presenting the solution one of of three important challenges.

**Heterogeneous invocation patterns.** The researches describe a compact histogram structure that keeps track of, in their case, an application's[2] IT. Since the concept of applications does not exist within OpenDC Serverless, we set the histogram to track ITs per function instead of applications. The histogram earns its compactness property from limiting the range of times it accepts, therefore any IT larger than the histogram's limit is not recorded and counts as Out-Of-Bounds (OOB).

The policy then respectively designates the pre-warming and keep-alive times as the 5th and 99th percentiles of the IT distribution. In accordance with the original design, we incorporate a "margin" to give the policy room for error which we subtract from pre-warming times and add to keep-alive times. Furthermore, we implement this structure in OpenDC Serverless using the Frequency [3] class from Apache Commons math3.stat library

[1] Idle time: time between two successive invocations of a function

[2] Application: container that hosts the execution of individual functions in Azure [5]

[3] https://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/stat/Frequency.html

for Java.

**Confusing or incomplete invocation patterns.** To avoid blatant mispredictions, the Hybrid Histogram policy incorporates a check on whether the histogram is representative of its function's invocation pattern. The researchers have specified that they compute the coefficient of variation of the bin counts and then verify whether it lies within a specific threshold. If not, the policy resorts to using the histogram's limit as a fixed keep-alive duration until the histogram learns the function's invocation pattern. Our implementation mimics this approach.

On an efficiency note, while the current version of the simulator does not model the policy's value tracking overhead, we still opted for the researchers solution of using Welford's online algorithm for computing corrected sums of squares [54] to efficiently track the CV.

**Infrequent invocation patterns.** To solve this last issue, the Hybrid Histogram policy incorporates a time-series forecast component. The researchers describe its usage as needed for workloads with very infrequent invocations, which often exhibit a large amount of OOB (Out-of-Bounds) idle times. Similar to the original implementation, we use the ARIMA time-series forecast model [26].

While the researchers use the pmdarima[4] package for python, we use the auto.arima[5] method from R's forecast package for automatic estimation of model parameters (p, d, q) that produce the best fit (both of these methods are equivalent).

We found in some informal experiments that these methods struggle with seasonal differencing. Considering the functions for which ARIMA usage is appropriate usually present very infrequent complex invocation patterns with multiple seasonality and small amounts of ITs to work with, we found that the seasonality tests used by these ARIMA solutions can struggle to produce correct forecasts. Therefore similar to the researchers approach, we designed the time-series forecast component to be easily swap-able with other more effective models [42].

Lastly, the researchers specify an interesting albeit slightly confusing approach to computing the pre-warming and keep-alive times. Again, an error margin is used but in a different manner, to quote from [46] "if the predicted IT is 5 hours, we set the pre-warming window to 4.25 hours (5 hours minus 15%) and the keep-alive window to 1.5 hours (15% of 5 hours in each side of the IT prediction)."

[4]https://pypi.org/project/pmdarima/

[5]https://www.rdocumentation.org/packages/forecast/versions/8.12/topics/auto.arima

| Fixed Keep Alive | Hybrid Histogram | No-Unloading |
|---|---|---|
| 5 min keep-alive | 1 hour histogram limit, 60,000ms | No instance termination |
| 10 min keep-alive | bin width, 50% OOB threshold, | |
| 20 min keep-alive | 10% histogram error margin, 15% | |
| 30 min keep-alive | forecast error margin | |
| 45 min keep-alive | 2 hours histogram limit, same parameters | |
| 90 min keep-alive | 3 hours histogram limit, same parameters | |
| 120 min keep-alive | 4 hours histogram limit, same parameters | |

Table 2: Resource management policy parameters for each experiment.

## 4.2 Serverless in the Wild reproduced: reproducing the Hybrid Histogram real-world experiment from the Serverless in the Wild [46] study in OpenDC Serverless

### 4.2.1 Experiment setup

In this subsection, we present our reproduction of the real-world experiment setup in Apache OpenWhisk from [46]. Similar to the source, we outline the comparison between Azure's Hybrid Resource Management policy and the Fixed keep-alive policy with variable parameters using real-world Azure Functions traces. Furthermore, we also detail each of the changes we made to the original procedure.

**Simulator setup.** We can only reproduce the experiment setting to the extent of how many common parameters are available between Apache OpenWhisk (real-world setting) and OpenDC Serverless (simulator). Nonetheless, we still manage to mimic a respectable number of important parameters presented in the list below. Furthermore, we use default values for the unspecified parameters.

1. The allocation policy is set to the default random policy (policy definition in Section 3.7)
2. The routing policy is set to the default random policy (policy definition in Section 3.5)
3. The resource management policies used in the experiment are Hybrid Histogram (policy definition in Section 4.1), Fixed-Keep-Alive (policy definition in Section 3.7) and the no-termination (no-unloading) policy (policy definition in Section 3.7).
4. The idle instance's memory usage is set to vary according to how much memory was consumed in the last execution of an idle instance. We set the factor to the default value of 20% (e.g. if an instance used 25 mb when last running, its idle memory usage would be 5mb)
5. Virtual Machines and their capacity : 18 VM's with 2 cores and 4GB of memory each

In addition to the above general parameters, each resource management policy has its own specific parameter(s) as shown in Table 2.

In total, we simulate 13 scenarios, of which, 8 are variants of the Fixed Keep Alive policy with varying timeout parameters, 4 are variants of the Hybrid Histogram policy with varying histogram limit parameters and one last scenario where we simulate the no-unloading policy (policy that does not terminate instances, detailed in Section 3.7). For each of these experiments, we ran three iterations with different seeds and plotted the error intervals to showcase the controllable and repeatable aspect of the simulator.

**Physical setup.** Our simulations ran on a 4 core Intel(R) Core(TM) i7-7700HQ CPU at 2.80GHz with 8GB of memory.

**Workload.** Similar to the real-world experiment, we selected 68 applications from the Azure Functions trace. The researchers specify that they sampled "mid-range popularity" applications. While the term "popularity" is loosely used with no explicit criteria in the paper, the description of Figure 5 from [46] points to application popularity as being the sum of invocations over all of its functions.

We therefore compute popularity for all of the 17,000+ applications in the trace, sort the list and set the median popularity as the central value of our sampling range . For example, if the median popularity is equivalent to 2000 invocations, we set the sampling bounds using a margin of 25% which lands us between 1,500 popularity and 2,500 popularity.

Overall, we drew a sample with a total popularity of 98,328 invocations. Contrary to the original experiment, we are not limiting the execution time to 8 hours, instead we are simulating the entirety of the sample duration, hence why the number of invocations are 8 times larger than the 12,383 invocations of the original experiment.

Furthermore, we detail how we converted the Azure traces to the OpenDC Serverless format in Section 4.2.2 and provide a small segment detailing how representative of the dataset the experiment sample is in Section 4.2.3.

**Recorded Metrics.** The researchers present their results using two main metrics, wasted memory time, or i.e. the total time an idle instance spends in memory and the percentage of cold starts. We thus set the simulator to record total invocations, total cold starts and wasted memory time for every function. Furthermore, in accordance with the real-world experiment, we normalize the wasted memory time of every policy to the 10 minute Fixed Keep Alive policy.

We can already preview a possible threat to the validity of the experiment: the real-world experiment's metrics were recorded per application, while our simulator records metrics per function. Since Azure loads entire applications into memory and not single functions, the wasted memory time of multiple functions is recorded as one. The simulator instead loads every function by itself and thus records wasted memory time values for each function of an application and aggregates them (possibly amounting to a larger value than if computed per application). This difference in metric granularity can produce strong variations as seen later in the results. Regardless, we chose to report this metric and furthermore included both per function and per application versions of the cold start percentage distributions.

### 4.2.2 Generating realistic OpenDC Serverless traces from the Azure Functions trace

On June 17th 2020, Shahrad et al. publicly released the first official real-world FaaS workload trace from Microsoft Azure Functions on Github [11]. Along with the released dataset, they provide a characterization and optimisation study of said dataset in [46]. In this subsection, we present our procedure for generating realistic OpenDC Serverless traces from the Azure dataset. (code available as Jupyter notebook in [21]).

The Azure trace comes in 14 sets of three files representing 14 days of execution history. The three files are detailed in the list below.

1. The first of the three files is a history of invocations per function. Since the invocations were binned at 1 min intervals, this file contains 1440 columns (1440 minutes, 24 hours) per function, each containing the precise number of invocations in that specific minute.

2. The second file contains distributions of execution time per function. The researchers report, for each function, the number of invocations, the minimum, average, and maximum execution times over the number of invocations across 24-hours.

3. The third and last file contains distributions of allocated memory per application (A function app is the container that hosts the execution of individual functions in Azure [5]). For this metric, the researchers sampled each application's memory every 5 seconds, which they then proceed to average every minute. The researchers report both the 1 minute average over these 5-second sample and the sample count. Moreover, they also provide percentiles of the distribution of average allocated memory.

Furthermore in each of the three files, each function has three anonymized identifications columns: a hash of the owner's id, a hash of the function's id and a hash of the application id. By matching these fields across all three files, we compile one CSV file for each day of the trace where each function is followed by all the columns describing its invocations, execution time and allocated memory. From here on the generation procedure is split onto three parts.

**Generating invocations.** Since we have the exact number of invocations per minute for every function, we simply compute the empirical cumulative density function (ECDF) of the invocation bins using the ECDF module from the statsmodels python library. We then use the ECDF to generate invocations for the OpenDC Serverless trace.

**Generating execution times and allocated memory.** Considering we do not have access to the actual sampled values for both of these metrics, we need to use the provided average, min, max and size (sample size for execution times and sample count for allocated memory) values to estimate distribution parameters. Since execution times and allocated memory are both strictly positive values, we decided to use a log normal distribution to generate execution time and allocated memory values.

To estimate the mean and standard deviation of the distribution, we wrote an R script that estimates the maximum log likelihood for different values of $\mu$ (mean) and $\sigma$ (standard deviation) using the min, max, average and size parameters provided in the trace (code available in [20]). We then use the log normal distribution to generate realistic execution time and allocated memory values for the OpenDC Serverless trace.

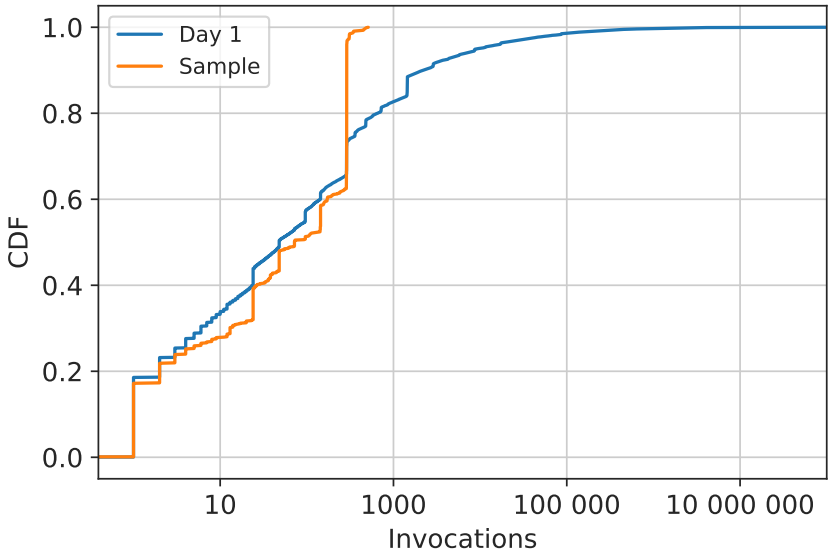



Figure 7: Distribution of invocations per function of both the sample and day 1 of the Azure Functions Trace. (The horizontal axis is logscale.)

#### 4.2.3 Representativeness of the experiment sample

In this subsection, we aim to show that the experiment sample drawn in Section 4.2 is representative of the first day of the Azure Functions trace. To achieve so, we visualize and compare both the sample's and the first day's distributions of execution times per function and allocated memory per application (code available as Jupyter notebook in [21]).

**Comparing distributions.** Figure 7 shows the cumulative density function (CDF) of invocations per function of both the experiment sample and the first day of the Azure Functions trace. Since we are only using day 1 of the trace as a reference, we point out the similarity between the first day of the trace and the entire 14 days of the trace presented in Figure 5a from the characterization study by Shahrad et al. in [46]. This observation makes day 1 of the dataset a viable reference to measure how representative of the trace the sample is.

Moreover, we see that the sample's distribution follows relatively close trends to those of the first day of the trace. The cutoff around 800 invocations is due to the functions belonging to mid-range popularity applications.

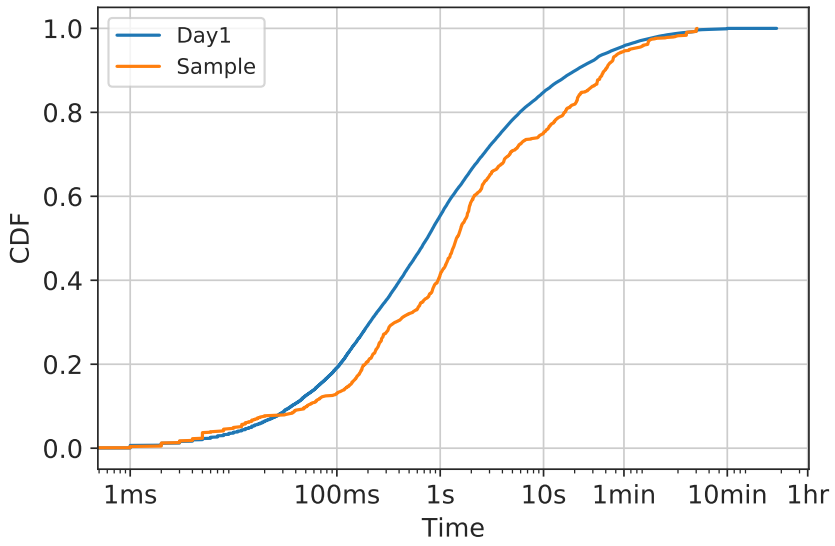



Figure 8: Distribution of execution time per function of both the sample and day 1 of the Azure Functions Trace. (The horizontal axis is logscale.)

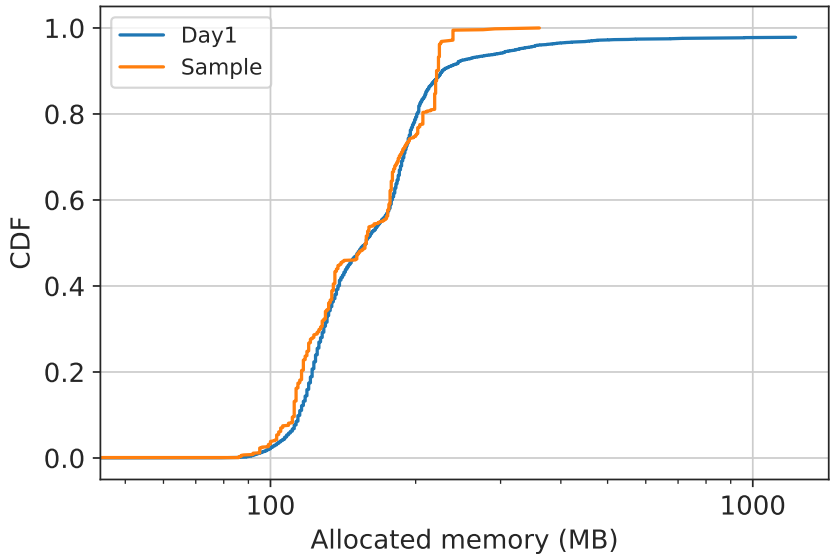



Figure 9: Distribution of allocated memory per application of both the sample and day 1 of the Azure Functions Trace. (The horizontal axis is logscale.)

Furthermore, Figure 8 shows the distributions of execution time, here, the sample excludes extreme outliers ($>$ 5 min). Finally, Figure 9 also excludes outliers ($>$ 200MB) for the distribution of allocated memory per application.

### 4.3 Cold Lambda: Simulating cold starts in AWS Lambda

In this subsection, we present a validation experiment to assess the simulator's capabilities of reproducing real-world Function-as-a-Service (FaaS) platforms behavior in the context of resource management. We achieve this by running multiple workloads with various invocation patterns in both the simulator and a real-world FaaS platform for the duration of 24 hours.

We picked AWS Lambda as our target platform for this experiment. The motives behind picking Lambda over Azure, the obvious contender in the context of this work, are mostly backed by the operational similarities and differences between the simulator and the two platforms. We list the three most important reasons below:

1. Contrary to Azure, AWS Lambda serves one request per function instance in parallel as illustrated in Figure 2 of [53] and mentioned in the lambda concurrency documentation [16]. This also concurs with the simulator's design.

2. The simulator's Instance Hypervisor component is similar in concept to the Firecracker Virtual Machine Monitor used in AWS Lambda [23].

3. Contrary to Azure, AWS Lambda provisions resources per function and not per application as explained in the fifth research question of [38]. This happens to concur with the simulator's design.

4. Lambda's CloudWatch Insights interface allows for easy querying of logs [1], this helps us extract the necessary information in a convenient manner

#### 4.3.1 Experiment setup

**Simulator setup.** Later on while performing the experiments, we found that AWS Lambda dynamically tweaks the lifetime of its function instances. Since we do not have access to the specifics of this policy, we manually tweaked the keep-alive parameter for each workload in order to obtain comparable amounts of cold starts.

**Physical setup.** Our simulations ran on a 4 core Intel(R) Core(TM) i7-7700HQ CPU at 2.80GHz with 8GB of memory.

**Lambda setup.** To simulate arbitrary execution times, we wrote a simple Lambda function that sleeps for the provided duration. We moreover duplicated the function for us to be able to run and log each workload independently in parallel. Then, we set all the functions to be triggered through a REST API using Amazon's API Gateway service.

Since we are planning to use workloads from the Azure Functions dataset, we set up a python script to perform invocations according to their timestamps in the trace. We furthermore incorporated a delay of 1 second between concurrent invocations of the same function to later obtain a log entry for every individual invocation (binned every second).

**Metrics.** For this purpose, we use the query interface of AWS CloudWatch insights to extract the exact date and time, execution time and initialization time (cold start) of every invocation. Furthermore, we set the simulator to record the same metrics.

**Workload.** Evaluating resource management behavior requires realistic workloads with a broad range of invocation patterns. Therefore, we randomly sampled 4 different workloads from the first day of the Azure dataset. For every sample, we tweaked the criteria to obtain the following invocation patterns:

1. Timer invocation pattern: less than 10 minute interval, around 300 invocations
2. Frequent invocation pattern: invocations every few minutes, around 500 total invocations
3. Infrequent invocation pattern: infrequent bursts of invocations, around 200 total invocations
4. Very infrequent invocation pattern: very infrequent invocations, around 25 total invocations

Furthermore, we define two versions of every workload, with and without concurrent invocations. With this distinction, we aim to assess the cold start patterns caused by bursts of concurrent invocations separately.

# 5 Experiment Results and Analysis

Throughout this section, any words highlighted in *italics* are considered to be key findings from the results.

## 5.1 Serverless in the Wild Reproduced

In the original experiment [46], the researchers present both a simulation experiment using the first week of the trace, and a real-world experiment in Apache Open Whisk using a sample. Since we showed in Section 4.2.3 that our sample is, to an extent, representative of day 1 of the dataset. We reproduce all the relevant data visualisations from the paper regardless of whether they were performed in simulation or reality.

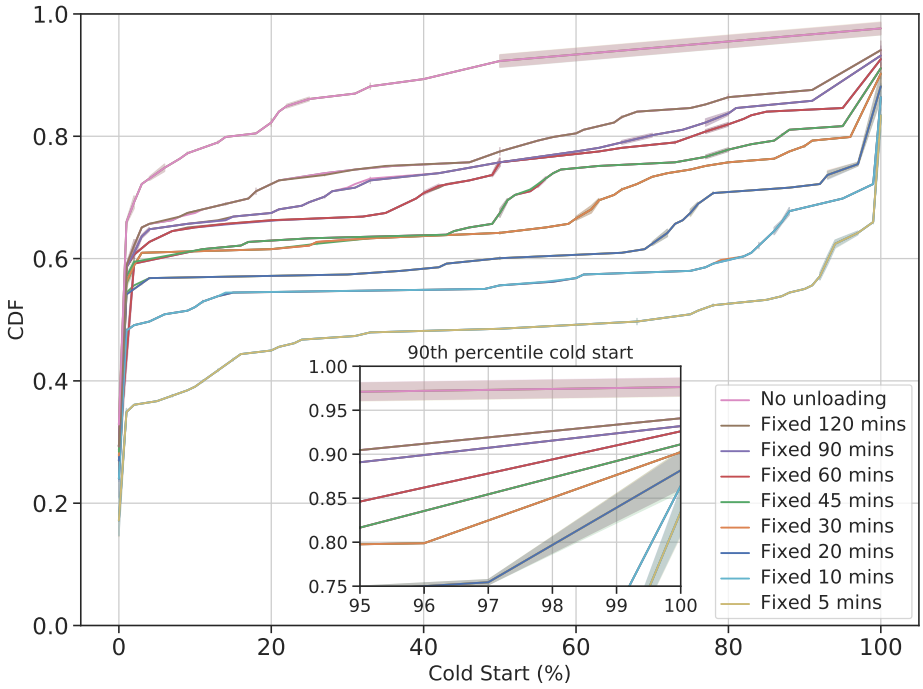
### 5.1.1 Results

In accordance with the order of the original experiment, We start by evaluating the fixed keep alive policy, we examine how the length of the timeout parameter affects the percentage of cold starts per function and per application[6]. We can already observe in Figure 10 that our simulation results closely follow the trends from Figure 14 of [46].

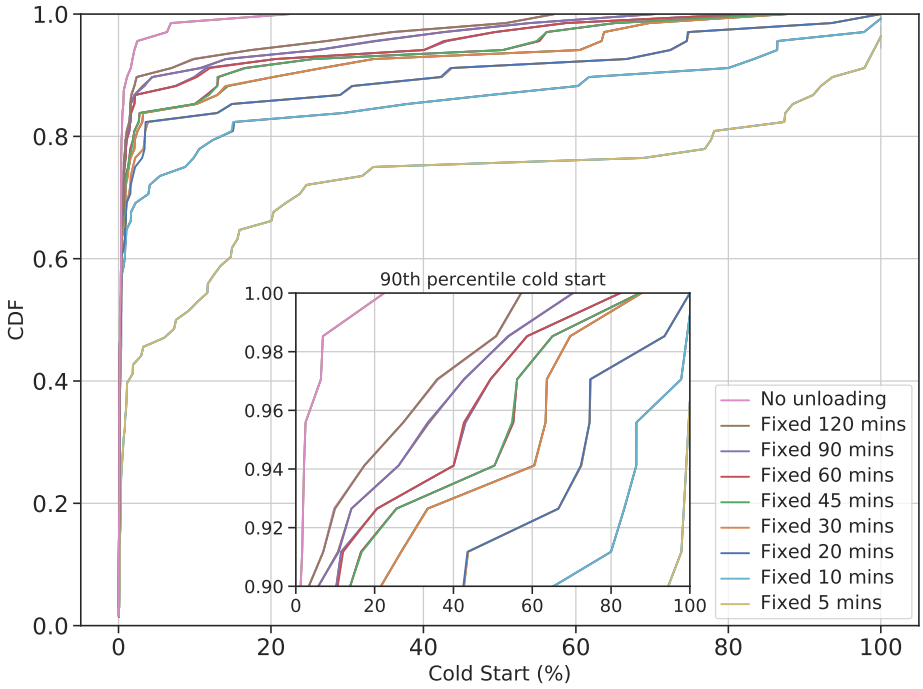
We look at Figure 10a first and observe that the the no-unloading policy performs best since it does not terminate any instances. *Yet even the No-Unloading policy can still register instances of 100% cold starts,* an effect that is due to some functions containing only one invocation in the whole week. The fixed policies show, as expected, a downward trend from the highest to the lowest keep-alive. Naturally, longer timeout policies produce less cold starts but remain more expensive to operate.

Upon taking a closer look at the 90th percentile in Figure 10b, we see that the 99th percentile application cold start decreases from 100% to around 55% between the 10min and 120min keep-alive variants. *While this downwards trend might seem rather drastic compared to that of Figure 14 from [46]*, we attribute this to our sample not containing applications with less than median popularity[7]. We chose not to model the latter since the cold start pattern of extremely low popularity applications with infrequent invocation patterns does not improve significantly with longer keep-alive parameters.

[6]Application: container that hosts the execution of individual functions in Azure [5]
[7]popularity: total number of invocations.




(a) Per Function




(b) Per Application

Figure 10: Cold start distribution of the fixed keep-alive policy, as a function of the keep-alive length.

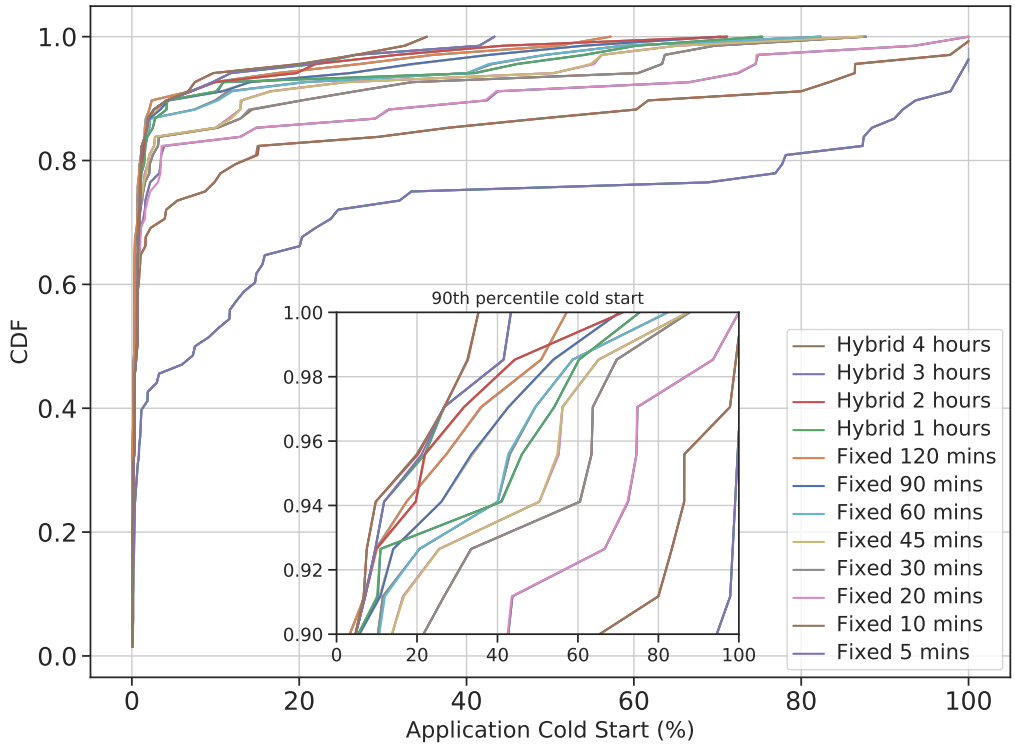
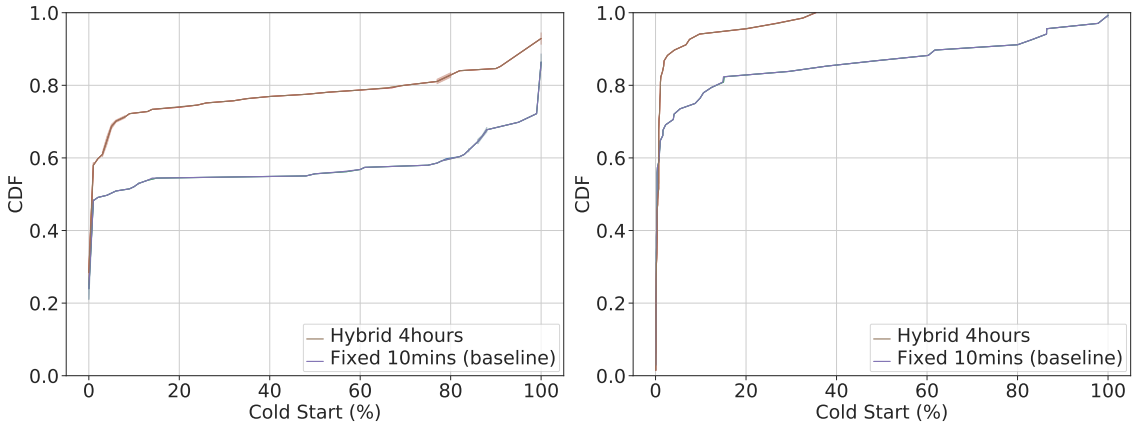



Figure 11: Cold start distribution of the hybrid and fixed keep-alive policies per Application.

We then move on to evaluating the impact of the hybrid histogram policy. We first present an overview visualization of all the different policies used in the experiment in Figure 11.

In this comparison, we see a significant reduction of 20% in the 99th percentile cold start going from the best case fixed variant (120mins) to the best case hybrid variant (4 hours). This observation further reinforces Shahrad et al.'s claim that *the Hybrid Histogram resource management policy is particularly effective for applications with infrequent invocation patterns, which usually reside within the 4th quartile of the distribution.*

For the real-world experiment in Apache Open Whisk the researchers chose to only report the best case hybrid variant alongside the 10 minute fixed policy. Figure 12 shows the equivalent of Figure 20 from [46]. *Again, our results follow similar trends.*




(a) Per Function (b) Per Application

Figure 12: Cold start distribution of fixed keep-alive and hybrid policies in OpenDC Serverless.

While our cold start percentage results were fairly inline with those of the original experiment, the same cannot be said about the trade off between cold starts and wasted memory time. Unfortunately, we have not been able to reproduce the positive memory waste values from Figure 15 of [46] as shown by our simulation results in Figure 13.

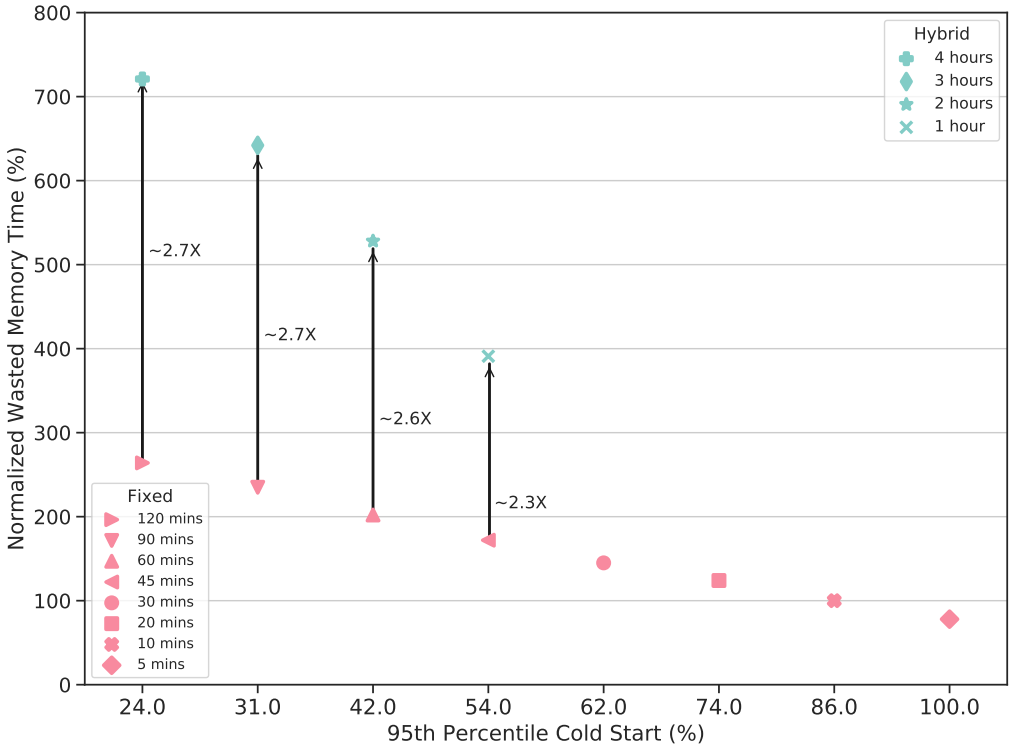



Figure 13: Trade-off between cold starts and wasted memory time for the fixed keep-alive policy and the Hybrid Histogram policy per Application

Before we discuss these trade-off results, it is important to note that the researchers originally used the 75th percentile to report cold starts. The motivation behind their choice was to highlight fluctuations in the cold starts of applications that benefit the most from Hybrid Histogram, notably the ones with infrequent invocation patterns. By applying that same reasoning to our sample, we found the 95th percentile to be a better fit.

In terms of cold starts, our results are mostly inline with those presented in the paper, the 1/2/3 and 4 hour hybrid variants respectively map to the 45/60/90 and 120 min fixed variants. Moreover, due to the sample being mostly composed of median popularity applications[8], the rest of the fixed variants perform rather poorly.

In terms of memory waste, we observe the complete opposite of what the original experiment's results portray. The hybrid variants waste on average, 2.5x more memory time than the fixed variants. We present below a list of the possible reasons behind this strong difference in results.

1. As mentioned in the recorded metrics paragraph of Section 4.2.1. Azure records memory time per application, while our simulator records memory time per function and aggregates the values. This can cause exponential variations in wasted memory time.

2. Since the OOB threshold parameter is not specified in the original experiment, we arbitrarily set it to a default value of 50%. This is particularly important because the policy spends an arbitrary amount of time accepting OOB values and reverting back to using a wasteful Fixed Keep Alive strategy. With a lower threshold, the policy could switch to using the time-series forecast component quicker. This approach however, could hinder the performance in terms of cold starts.

3. Outliers in the idle time distribution can heavily affect the value of the prediction. Despite that we use the 5th and 99th percentiles to set the pre-warming and keep-alive times, we see in a facet grid of all the functions presented in our experiment analysis notebook [19], that only a small number of functions from the sample heavily waste memory time.

4. The original version of the Hybrid Histogram policy tracks both pre-warming and keep-alive times per application. Naturally, the tracking granularity has been lowered to functions in OpenDC Serverless. This can cause a number of significant variations in the results, for instance, generalizing the behavior of a single function to multiple function can reduce effects caused by outlier Idle times in the histogram or time series components, and thus generally produce more grounded predictions. While this approach would likely affect the cold start performance it would also likely have a positive effect on wasted memory time. A possible solution to this issue in the future is to average the two parameters between functions of the same application.

[8]Median popularity application: applications with around 1500 to 2500 invocations per week in the context of the Azure Functions trace.

### 5.1.2 Advantage of using simulation.

Shahrad et al. ran two 8 hour executions in Apache OpenWhisk, one for the 4-hour hybrid policy and one for the 10 minute fixed policy, each respectively consisting of 12,383 invocations. Both of the executions were performed using a total of 19 VM's, one VM with 8 CPU cores and 8GB of memory for the OpenWhisk services and 18 other VM's with 2 CPU cores and 4GB of memory each. In contrast, our simulations ran on a 4 core Intel(R) Core(TM) i7-7700HQ CPU at 2.80GHz and 8GB of memory.

Knowing the total execution time and the physical resources used for both the real-world experiment and the simulated reproduction, we compute the speedup achieved from running both experiments in simulations:

$$\text{Experiment in simulation: } 57\text{s} \times 4 \text{ cores} \equiv 0,0158333\text{h} \times 4 \text{ cores} \tag{1}$$

$$\approx 0.0633332 \text{ core-hours} \tag{2}$$

$$\text{Experiment in reality: } 16\text{h} \times [8 + (2 \times 18)] \text{ cores} = 704 \text{ core-hours} \tag{3}$$

Accordingly with the results of equations 1 and 3, we obtain a substantial speed up of $\approx$ 11,116 core-hours ($\approx$ 1 core-year and 3 core-months savings) when conducting the experiments in simulation.

We went a step further as to also estimate the costs of the simulation and real-world experiment assuming the researchers were using Azure VMs. We use the Azure pricing calculator to calculate the price. Furthermore, we include the name of the instances for reproducibility reasons.

$$\text{Experiment in simulation: 1 A3 (4 cores, 8 GB RAM) x 1 Hours} \times \tag{4}$$

$$0,0158333 \text{ h} \equiv 0.24 \text{ \$} \times 0,0158333 = 0.0038 \text{ \$} \tag{5}$$

$$\text{Experiment in reality: 1 A4 (8 cores, 16 GB RAM) x 16 Hours} + \tag{6}$$

$$18 \text{ A2 (2 cores, 4 GB RAM) x 16 Hours} = 42.14\$ \tag{7}$$

As expected, equations 4 and 6 reveal a substantial cost reduction of $\approx$ 11089 times when conducting the experiments in simulation ($\approx$ 42.13\$ savings, relatively to the cost of running in the cloud). Furthermore, we did not account for experiment repeats (3 in our simulation, none in the real-world experiment) and the difference in our workload sizes (ours being 8 times larger).

### 5.1.3 Summary

The results presented in this section show that the simulator is capable, to an extent, of reproducing the results from the original experiment [46]. At first we presented reproduced simulations of the original simulation experiments from the paper, then we presented a simulation of the original real-world experiment experiment.

We started by exploring the cold start distributions of our workload using different variants of the fixed keep alive and the Hybrid Histogram policy which we found to mostly be aligned with the original results. To be more precise, our simulations showed the fixed policies to be more effective than in the original experiments. We attributed the latter to our sample not having less than median popularity applications[9] which are usually applications with infrequent invocation patterns (unfavorable for Fixed Keep Alive policies).

Moreover we added a visualization of our own that includes all the different policies in one graph. The latter showed that the hybrid policies are particularly effective for applications with infrequent invocation patterns.

We followed with a visualization of the trade-off between wasted memory time and cold start performance which we unfortunately found not to be aligned with the original results in terms of wasted memory time. Nevertheless, we presented a list of possible reasons for the strong variations found in our experiment.

Lastly, we showed that our simulations reduced both temporal and material cost by a factor of $\approx 11000$ times compared to the real experiments.

For reproduction purposes, the complete input, results analysis and plotting script of this experiment is publicly available in [19] [34].

[9] Median popularity application: applications with around 1,500 to 2,500 invocations per week in the context of the Azure Functions trace.

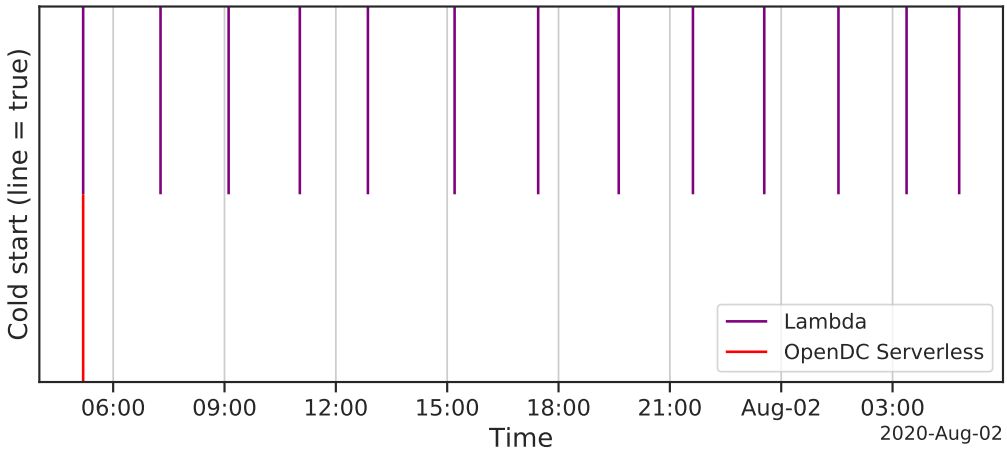



Figure 14: Comparison of cold start events over time between AWS Lambda and OpenDC Serverless for the timer invocation pattern workload.

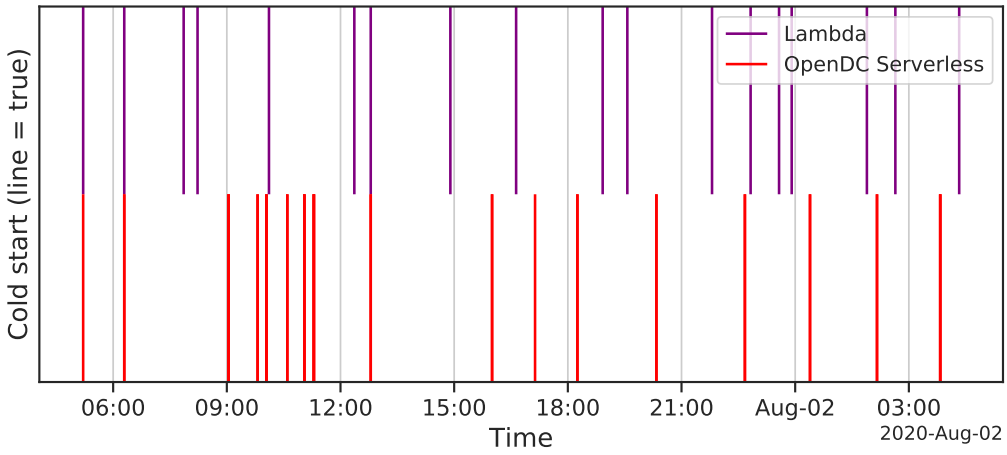
## 5.2 Cold Lambda

Throughout this subsection, we present the results of this experiment using comparative event plots of the cold start occurrences in AWS Lambda and the simulator. As mentioned before in the experiment setup, the simulator is set up differently for every workload, therefore we will also mention how we set up the simulator for every workload.

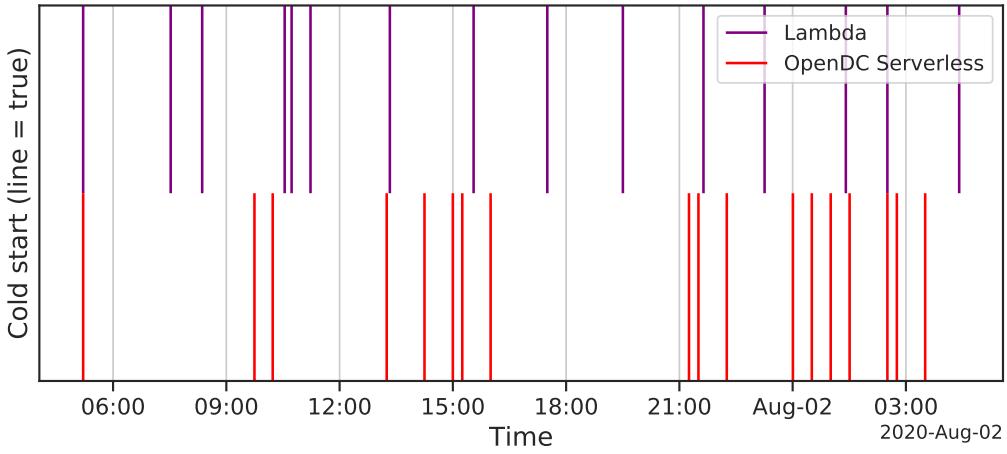
### 5.2.1 Results

**Timer invocation pattern.** Naturally, for this workload, we set the keep-alive parameter to the timer interval (5 min). Of course, this produces a single cold start in the entire simulation. While we expected similar results from AWS Lambda, we were met with a very different outcome.
It appears from the results shown in Figure 14 that AWS Lambda automatically terminates the function instances around every 2h:30min. We can think of *this periodic termination of instances as a refresh procedure for function instances within AWS Lambda.* Moreover, this behavior can most likely be associated with our usage of the free tier since AWS Lambda presents a paid solution to obtain predictable cold start times in [6].

There were no differences between the concurrency and no concurrency versions of this workload, hence the reason we decided to only present the no-concurrency variant.



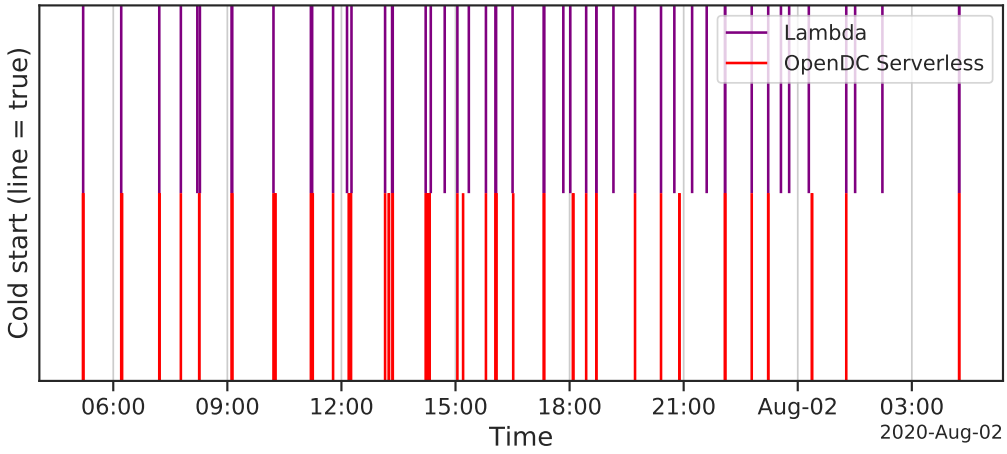

(a) With concurrent invocations.



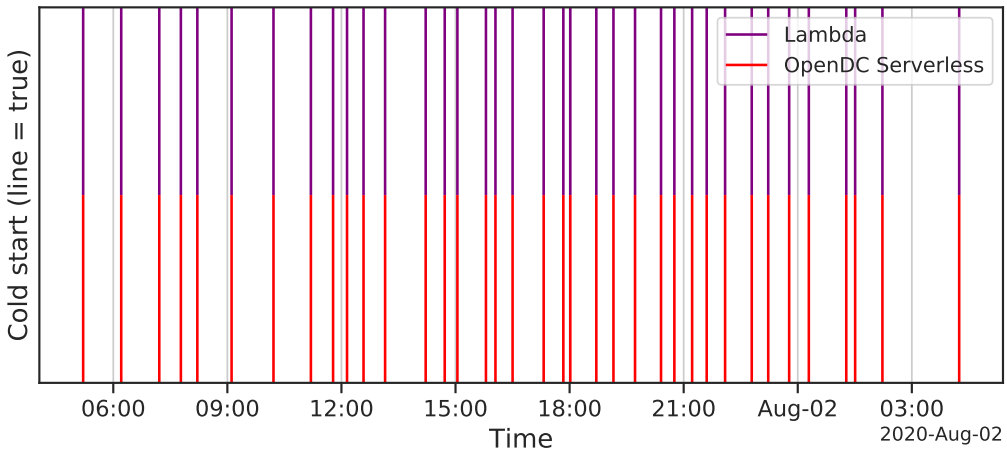

(b) Without concurrent invocations.

Figure 15: Comparison of cold start events over time between AWS Lambda and OpenDC Serverless for the frequent invocation pattern workload.

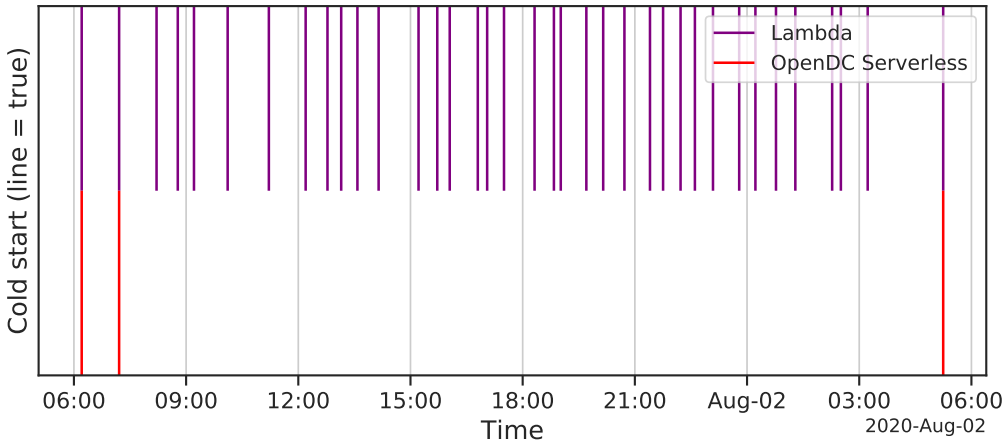
**Frequent invocation pattern.** For this workload, we set the keep-alive parameter to 14 minutes for the experiment with concurrent invocations and 5 minutes for the other. In total, both the simulations and experiments produced around 18 cold starts each. We can see in Figure 15 that most of the simulator's cold start predictions are not precise for this workload, more specifically, we observe that the simulator's run shows bursts of cold starts while Lambda's run does not. Therefore, we deduce from the lack of burst cold starts in AWS Lambda that *the platform uses some sort of expected load prediction to pre warm its instances.* Agache et al. mentioned the use Little's law (definition in [37]) in section 4.1.2 of their recent publication about the Firecracker VM technology [23] used for function instances in AWS Lambda.




(a) With concurrent invocations.




(b) Without concurrent invocations.




(c) Without concurrent invocations using Azure's Hybrid Histogram policy.

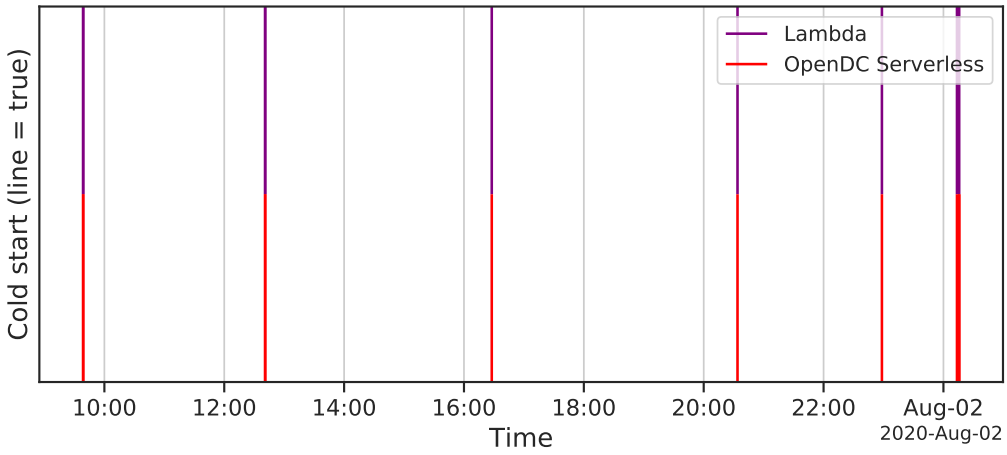
Figure 16: Comparison of cold start events over time between AWS Lambda and OpenDC Serverless for the infrequent invocation pattern workload.

**Infrequent invocation pattern.** For this pattern, we respectively set the keep-alive parameter to 30 and 10 mins for the concurrency and no concurrency workloads. We start to see a significant amount of cold starts. Precisely 37 cold starts for the no concurrency workload and 53 for the other. Figure 16 shows the comparative cold start event plots of this scenario.

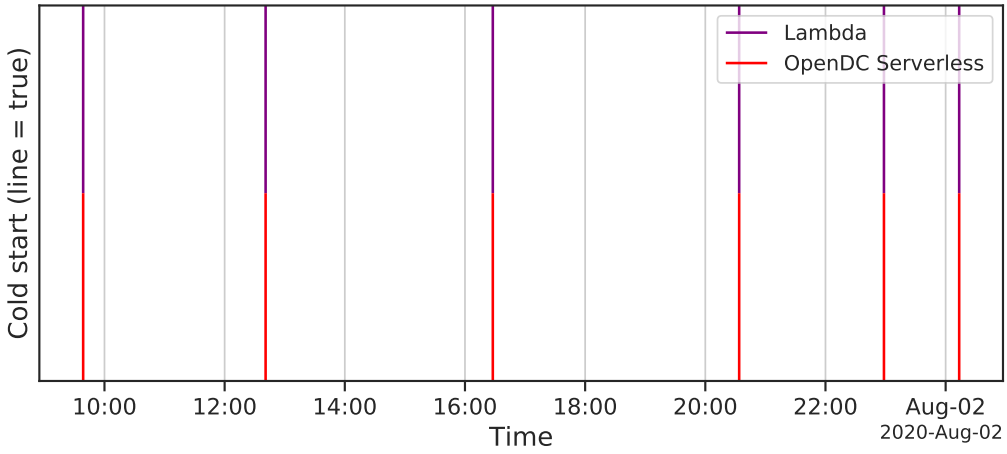
Similar to OpenDC Serverless, AWS Lambda fires up multiple function instances when it receives multiple concurrent requests. This can be seen in 16a in the form of bold bursts of cold starts for both the platform and the simulator's results. 16b shows a true-to-life simulation of the no-concurrency scenario.

However, in both cases, AWS Lambda seems to struggle with setting an effective keep-alive duration for this scenario. We deduce that *the platform uses a conservative resource management policy*[10], which most likely does not adapt quickly when dealing with infrequent invocation patterns (even when the pattern is relatively periodic like in 15b). To demonstrate our deduction, we show in Figure 16c, a better performing run of the same trace using the Hybrid Histogram resource management policy (defined in Section 4.1) with the same parameters as the 4 hour variant in Table 2 in Section 4.2.1.

[10]conservative adaptive resource management policy: policy that is less susceptible to variations in a function's idle times and thus less inclined to adjust the pre-warming and keep-alive parameters




(a) With concurrent invocations.




(b) Without concurrent invocations.

Figure 17: Comparison of cold start events over time between AWS Lambda and OpenDC Serverless for the very infrequent invocation pattern workload.

**Very infrequent invocation pattern.** Similar to all other FaaS platforms, AWS Lambda struggles to minimize cold starts for workloads with a very infrequent invocation pattern. The simulator easily reproduces this scenario in Figure 17. We found the results of this last variant to be particularly not interesting due to the fact that very infrequent invocations usually lead to a cold start.

### 5.2.2 Summary

Considering we do not have access to AWS Lambda's resource management policy, we have previously stated that we manually tweak the keep-alive parameter for each workload in order to obtain comparable amounts of cold starts. With the introduction of a human factor in the experiments, we cannot issue a concrete judgement on the simulator's ability to effectively simulate an AWS Lambda execution pattern. Nevertheless, we were able to extract key findings from the results.

Through the timer invocation pattern experiment, we discovered that AWS Lambda automatically refreshes its function instances after a certain period (2h30min in this scenario).

The simulation of the frequent invocation pattern showed that a regular run with a simple fixed keep-alive parameter exhibits bursts of cold starts. The same workload in AWS Lambda yielded a similar amount of cold starts, but did not exhibit the same bursty behavior seen in the simulation. Therefore, we deduced that AWS Lambda was able to somehow predict these cold start bursts, and was thus using pre-warmed pools of instances to mitigate the issue.

The infrequent invocation pattern experiments showed the most successful simulations thus far. Both the experiments with and without concurrent invocations were simulated in a relatively true to life manner. The infrequent invocation pattern, albeit relatively periodical, exhibited a large number of cold-starts in both the real and simulated runs, likely due to the use of the Fixed Keep Alive policy on our side and a conservative adaptive resource management policy[11] on AWS Lambda's side. We therefore showcased the improvements from using a policy such as Azure's Hybrid Histogram policy (explained in Section 4.1) on the same workload.

Lastly, we showcased a successful simulation of the very infrequent invocation pattern. Although not very interesting, the results showed that very infrequent workloads suffer from systematic cold starts in AWS Lambda. It is however important to note that the latter statement is not conclusive since we could possibly expect different results from a longer experiment for this workload specifically.

For reproduction purposes, the complete input, results analysis and plotting script of this experiment are publicly available in [19].

[11] conservative adaptive resource management policy: policy that is less susceptible to variations in a function's idle times and thus less inclined to adjust the pre-warming and keep-alive parameters

# 6 Analysis of Limitations and Future Improvements

Throughout this thesis, we mentioned a number of limitations regarding the simulator and the conducted experiments. In the following subsections, we will explore the most significant limitations with detail and discuss possible solutions and further improvements. Furthermore, with this discussion, we aim to motivate the Function-as-a-Service (FaaS) community to pursue research in the field of FaaS simulation.

## 6.1 Workload granularity and scalability

When designing our trace format for the simulator, we limited the granularity of the trace by opting to include averaged execution times and allocated resources. This was done to permit the usage of large trace files in the future, however, the lowered precision omits data that could be considered important for some workloads (e.g. precise usage fluctuation between concurrent executions).

Considering there is only one publicly available FaaS trace [11] which was recorded with reduced precision (invocations are binned in intervals of 1 min) and that FaaS operates on relatively low granularity workloads compared to IaaS (Infrastructure-as-a-Service) platforms such as Amazon EC2 [7] that operate on VMs. Future function traces could be recorded at a far higher precision (e.g. invocation per entry) producing extremely large amounts of data.

Such traces would raise scalability issues with the current design of the routing component (detailed in Section 3.5). Thus, future research could be conducted on a distributed approach to request routing.

## 6.2 Absence of physical resource layer modelling

The OpenDC simulator provides some valuables features such as VM and physical machine simulation. The current implemented version of OpenDC Serverless is not fully integrated with the existing OpenDC simulator and thus omits modelling the physical resource layer of FaaS platforms. We therefore built the Resource module to be integrable with the existing components of the OpenDC simulator.

With a full future integration, the simulator could be capable of modelling hardware heterogeneity, an issue which makes predicting function execution time relatively difficult due to each request possibly being deployed on different types of hardware.

Previous work presents examples of hardware heterogeneity in FaaS platforms. Wang et al. discuss AWS Lambda, Microsoft Azure Functions and Google Cloud Functions in [53] while Figiela et al. discuss IBM Cloud Functions in Section 6.7 of [28]).

## 6.3 No auto-scaling support

By having the user specify a set amount of VMs, the simulator does not model an essential aspect of FaaS platforms: auto-scaling based on the workload. Future improvements to the design could include an interface for custom auto-scaling protocols which link to the resource management component (detailed in 3.7).

## 6.4 Limited cost modelling

Part of our requirements for the simulator was a cost modelling component. While we did implement an interface to cost modelling, we only modelled the cost according to the public pricing models of popular FaaS platforms and not the real cost of FaaS operations.

Contrary to VMs or physical servers where cost is determined by tracking up-time, tracking cost in FaaS is not as straightforward. Each platform puts a relatively similar price on execution time and provisioned memory, however, the actual under the hood costs have never been made available to the public, likely because it can reveal the profit margins of large cloud providers.

Moreover, as pointed out in the previous subsection, the simulator in its current state does not support auto-scaling of VMs nor does it model physical machines. Therefore, solutions such as determining the cost by tracking VM usage are not only inaccurate as we show in the next paragraph, but also non-trivial to implement in the current state of OpenDC Serverless since a complete cost model would require the implementation of auto-scaling support for it to provide price variations for different simulation settings.

Furthermore, it is necessary to point out that in the search for a cost tracking solution, we cannot simply estimate FaaS operation costs from CPU/RAM/VM/Physical machine usage alone. This article [4] by Amiram Shachar for the medium reveals the slim portion occupied by function execution costs in TimerCheck's (a service built entirely on top of AWS Lambda) bill. Instead the majority of costs relate to networking (API Gateway), monitoring (AWS CloudWatch) and other serverless services offered by AWS Lambda. Considering most functions execute for a small amount of time, this case could apply to lots of other serverless users. This further diversifies the notion of FaaS costs and makes modelling them accurately a far bigger challenge than originally suspected.

In conclusion, with the current publicly available information about FaaS pricing and the current state of the simulator's design, we place cost modelling at a lower priority behind other future improvements such as auto-scaling support and network/IO modelling. Nevertheless, we remain hopeful for the simulator's ability to accurately model FaaS costs in the future.

## 6.5 No custom workflow-scheduling support

One of the many software-engineering challenges of the FaaS model described by the SPEC Cloud group in [50] is workflow composition.

Function compositions are necessary for modern applications with interdependent functions which require a certain level of state-management. The current version of the simulator only includes a sequential workflow-scheduler, namely the simulation core (described briefly in Section 3.3). The simulation core is unfortunately not a completely customizable workflow-scheduling interface, and thus does not allow for implementation of custom workflow scheduling protocols.

The SPEC RG Cloud reference architecture for FaaS architectures (described closely in Section 2) does however provide a specification for a complete workflow management system which could be added in the future by adding the missing components next to the simulation core. This would likely also require further modifications in the routing component of the simulator (described in Section 3.5).

## 6.6 Validity of the reproduction procedure in experiment A

We presented in 4.2 a small reproduction of the experiments by Shahrad et al. [46] in OpenDC Serverless. While we tried to reproduce the original experiment setup to the highest extent we could, there are some limitations to the validation of the experiment. In this subsection, we will highlight and detail some of the apparent flaws with the reproduction procedure following a set of reproducibility recommendations summarized by Scheuner et al. in Section 6.5 of their multi-vocal review of FaaS literature [44]. The recommendations are summarized from the methodological principles for reproducible performance evaluation in cloud computing described by Papadopoulos et al. in [43].

**P1: Report the number of iterations.** We reported 3 iterations for every resource management policy we bench marked in the experiment setting paragraph of Section 4.2.1.

**P2: Experiments should be conducted in different (possibly randomized) configurations of relevant parameters to cover a representative sample of the space of the controlled variables.** We mentioned in the description of Figure 13 in Section 5.1 that some variations between our results and the original results were possibly solvable by using different configurations for our simulations. However, we did not follow this principle due to a lack of time on the work's timeline. Since we are making the code and sample public, the improvements suggested could be implemented in a future iteration of the experiment.

**P3: Explicit experiment setup and P4: publish analysis and experiment code** We fully described our experiment setup in Section 4.2.1 and fully published the sample, experiment configuration, simulator code and results analysis script in [20].

**P5: Probabilistic result description of measured performance.** Similar to the original experiment, we use CDF's with confidence intervals to report the majority of our results, a possible future improvement would be to report statistical figures for the trade-off between wasted memory time and cold start performance results shown in Figure 13

**P6: Use appropriate statistical tests** Unfortunately we were not able to perform any statistical tests on the results due to a lack of time. This counts as a large limitation of this experiment since we do not provide enough information for a robust statistical analysis of the comparative measurements between the original and reproduced results.

**P7: Include measurement units in all figures** we include measurement units in all of our reported figures.

**P7: Report cost model** We do not report a cost model since we are conducting a the experiment in simulation, however we do report all the software and methods we used in the experiment setup (Section 4). Therefore it is possible, to an extent, to construct a cost model from this thesis.

# 7 Conclusion

Serverless computing or Function-as-a-Service (FaaS) is a promising cloud paradigm in its infancy. To fulfill serverless computing's vision of simplifying the increasingly complex state of cloud technologies, we need to enable and support the exploration of its architectural and operational concerns, simplify, stabilize and render accessible the process of experimentation in FaaS and motivate the community to explore new and existing FaaS workloads.

In this work, we proposed OpenDC Serverless. The first configurable, open-source trace-based FaaS simulator. To answer **RQ1**, *How can a FaaS simulator model the essential elements of FaaS architectures?*, we designed the system using state-of-the-art concepts from the SPEC RG Cloud reference architecture for FaaS [49]. We identified and extracted the essential components of a FaaS platform from the reference architecture [49] in Section 2.2.1, and implemented them in the simulator's design. Lastly, we built an easy-to-convert-to trace-format that captures the essential metrics recorded in a FaaS workload. We later showed in the experiments that the trace-format easily maps to the recently released Azure Functions dataset [17].

To answer **RQ2**, *How can a configurable FaaS simulator allow for exploration and testing of different or custom operational policies?*, we adapted the function execution pattern from the SPEC RG Cloud reference architecture for FaaS [49] in Section 2.2.2 and, designed customizable interfaces for allocation, resource management and request-routing policies that enable simulation of different configurations of this pattern.

We also designed an experiment framework coupled with a metric monitoring system which we have proven to be capable of producing valid results through a reproduction of the experiments from [46]. For the purpose of reproducing the latter experiments, we implemented the Azure Hybrid Histogram custom resource management policy and reproduced the real-experiments using real workloads from the Azure Function dataset [17]. The simulation results have shown similar trends to those of the real-world experiments.

To answer **RQ3**, *Can FaaS simulation reduce the temporal and material costs of experiments in the field?*, we calculated the actual temporal and material savings from conducting the Azure Hybrid Histogram experiments in simulation instead of conducting them in the cloud, and have obtained a speedup and savings factor of $\approx$ 11,000 times. This indicates that FaaS simulation can enable low-cost experiments, therefore, improving the accessibility of systems research in FaaS.

Lastly, to answer **RQ4**, *How can simulation of FaaS platforms provide a controllable environment for repeatable experiments, but maintain the accuracy of results provided by real-world experiments?*, we have implemented proper seeding and randomization control in the simulator's experiment framework. We then showcased the repeatable and controllable aspect of the simulator in our experiments by conducting multiple repeats with various different seeds. The results of our reproduction experiment showed accurate trends with expected deviations. Furthermore, we have included the modelling of hardware heterogeneity as a future improvement to the system in our list of limitations.

Through OpenDC Serverless, we offer the research community an early extensible framework for research and education in FaaS simulation, enabling accessible exploration of different operational and architectural FaaS patterns in a low-cost, controllable and repeatable experimentation environment.

We concluded this work with a list of limitations and future improvements in an appeal to motivate FaaS stakeholders to pursue research on highly configurable, open-source FaaS simulation in the future. In these limitations we set some future key directions for future work in FaaS simulation.

We first raised a concern over the scalability of FaaS simulation with lower granularity workloads since the sheer amount of trace data produced by FaaS platforms is prone to increase exponentially with more precise recordings, thus, future work could focus on more efficient approaches to FaaS simulation.

Furthermore, with more complex traces comes the need for function composition, a concept with increasing support in the research community [48], [25]. We aim for FaaS simulation to support workflow composition in the future.

We discussed the impact of infrastructure heterogeneity on the performance of FaaS platforms and highlighted the need for modelling this concept in future FaaS simulations.

Lastly, we set the spotlight on the difficulties of accurately modelling the cost of operating FaaS platforms, as thus, future studies could focus on economy-driven resource provisioning.

*The OpenDC Serverless software source code, and all different code and input used throughout this study, are available in [20].*

# References


[1] Analyzing log data with cloudwatch logs insights, 2017. [Online; accessed Aug 8th, 2020] https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76.

[2] Function-as-a-service market by user type (developer-centric and operator-centric), application (web  mobile based, research  academic), service type, deployment model, organization size, industry vertical, and region - global forecast to 2021, 2017.

[3] Cloud computing, once loved for its simplicity, is now a complex beast, 2018. [Online; accessed Aug 15th, 2020] https://www.forbes.com/sites/joemckendrick/2018/09/12/cloud-computing-once-loved-for-its-simplicity-is-now-a-complex-beast.

[4] The hidden costs of serverless, 2018. [Online; accessed Aug 16th, 2020] https://www.medium.com/@amiram$_2$6122/$the-hidden-costs-of-serverless-6ced7844780b$.

[5] Create a function app from the azure portal., 2019. [Online; accessed Aug 13th, 2020] https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-function-app-portal.

[6] New for aws lambda – predictable start-up times with provisioned concurrency, 2019. [Online; accessed Aug 12th, 2020] https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/.

[7] Amazon ec2 secure and resizable compute capacity in the cloud., 2020. [Online; accessed Aug 13th, 2020] https://aws.amazon.com/ec2/.

[8] Apache openwhisk. open source serverless cloud platform, 2020. [Online; accessed Aug 14th, 2020] https://openwhisk.apache.org.

[9] Aws lambda., 2020. [Online; accessed Aug 14th, 2020] https://aws.amazon.com/lambda/.

[10] Aws lambda customer case studies, 2020. [Online; accessed Aug 14th, 2020] https://aws.amazon.com/lambda/resources/customer-case-studies/.

[11] Azure functions trace 2019., 2020. [Online; accessed June 18th, 2020] https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md.

[12] Bullet point list of what the azure function traces will contain (pre-release commit)., 2020. [Online; accessed Mar 6th, 2020] https://github.com/Azure/AzurePublicDataset/commit/16055245f56127074f6c401a3d0cc1f5b948ff7e.

[13] Google cloud functions., 2020. [Online; accessed Aug 14th, 2020] https://cloud.google.com/functions.

[14] How does language, memory and package size affect cold starts of aws lambda ?, 2020. [Online; accessed Aug 5th, 2020] https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76.

[15] The kubernetes native serverless framework., 2020. [Online; accessed Aug 15th, 2020] https://kubeless.io.

[16] Managing concurrency for a lambda function, 2020. [Online; accessed Aug 5th, 2020] https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[17] Microsoft azure functions., 2020. [Online; accessed Aug 14th, 2020] https://azure.microsoft.com/en-us/services/functions.

[18] Open source, kubernetes-native serverless framework., 2020. [Online; accessed Aug 15th, 2020] https://fission.io.

[19] Opendc serverless experiments analysis., 2020. [Online; accessed Aug 18th, 2020] https://github.com/atlarge-research/opendc-serverless/tree/master/experiment-analysis.

[20] Opendc serverless source code and experiment scripts., 2020. [Online; accessed Aug 18th, 2020] https://github.com/atlarge-research/opendc-serverless.git.

[21] Opendc serverless trace generation scripts., 2020. [Online; accessed Aug 18th, 2020] https://github.com/atlarge-research/opendc-serverless/tree/master/trace-generation.

[22] Openfaas, serverless functions, made simple., 2020. [Online; accessed Aug 15th, 2020] https://www.openfaas.com.

[23] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

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

[25] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 89–103, 2017.

[26] G. E. Box and D. A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association*, 65(332):1509–1526, 1970.

[27] S. Eismann, J. Scheuner, E. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup. A review of serverless use cases and their characteristics SPEC RG Cloud working group. 05 2020.

[28] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience*, 30(23):e4792, 2018.

[29] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[30] A. Iosup, G. Andreadis, V. Van Beek, M. Bijman, E. Van Eyk, M. Neacsu, L. Overweel, S. Talluri, L. Versluis, and M. Visser. The opendc vision: Towards collaborative datacenter simulation and exploration for everybody. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 85–94. IEEE, 2017.

[31] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113. IEEE, 2011.

[32] H. Jeon, C. Cho, S. Shin, and S. Yoon. A cloudsim-extension for simulating distributed functions-as-a-service. In *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 386–391. IEEE, 2019.

[33] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[34] S. Jounaid. Azure dataset sample for opendc serverless simulator, Sep 2020.

[35] H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[36] P. Leitner and J. Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)*, 16(3):1–23, 2016.

[37] J. D. Little. A proof for the queuing formula: L= $\lambda$ w. *Operations research*, 9(3):383–387, 1961.

[38] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[39] J. M. Lyon and T. L. Ward. Sequential calculation of the median. *Computers & Industrial Engineering*, 4(1):31–39, 1980.

[40] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[41] S. K. Mohanty, G. Premsankar, M. Di Francesco, et al. An evaluation of open source serverless computing frameworks. In *CloudCom*, pages 115–120, 2018.

[42] I. Naim, T. Mahara, and A. R. Idrisi. Effective short-term forecasting for daily time series with complex seasonal patterns. *Procedia computer science*, 132:1832–1841, 2018.

[43] A. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. Abad, J. Amaral, P. Tůma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Transactions on Software Engineering*, PP:1–1, 07 2019.

[44] J. Scheuner and P. Leitner. Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software*, 170:110708, 2020.

[45] M. Shahrad, J. Balkind, and D. Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075, 2019.

[46] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423*, 2020.

[47] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup. Is big data performance reproducible in modern cloud networks? In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 513–527, 2020.

[48] E. van Eyk. The design, productization, and evaluation of a serverless workflow-management system. 2019.

[49] E. Van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, and A. Iosup. The SPEC-RG reference architecture for faas: From microservices and containers to serverless platforms. *IEEE Internet Computing*, 23(6):7–18, 2019.

[50] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann. A SPEC RG Cloud group's vision on the performance challenges of faas cloud architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 21–24, 2018.

[51] E. Van Eyk, A. Iosup, S. Seif, and M. Thömmes. The SPEC cloud group's research vision on faas and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 1–4, 2017.

[52] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*, 22(5):8–17, 2018.

[53] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC' 18)*, pages 133–146, 2018.

[54] B. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.