SimLess: Simulate Serverless Workflows and Their Twins and Siblings in Federated FaaS

Sashko Ristov
University of Innsbruck
Computer Science Department
Innsbruck, Austria
sashko.ristov@uibk.ac.at

Mika Hautz
University of Innsbruck
Computer Science Department
Innsbruck, Austria
mika.hautz@student.uibk.ac.at

Christian Hollaus
University of Innsbruck
Computer Science Department
Innsbruck, Austria
christian.hollaus@student.uibk.ac.at

Radu Prodan
University of Klagenfurt
Institute of Information Technology
Klagenfurt, Austria
radu.prodan@aau.at

ABSTRACT

Many researchers migrate scientific serverless workflows or *function choreographies* (*FCs*) on Function-as-a-Service (FaaS) to benefit from its high scalability and elasticity. Unfortunately, the heterogeneity of federated FaaS hampers decisions on appropriate parameter setup to run FCs. Consequently, scientists must choose between accurate but tedious and expensive experiments or simple but cheap and less accurate simulations. Unfortunately, related works support either simulation models for *serverfull* workflows running on virtual machines and containers or partial FaaS models for individual serverless functions focused on execution time and neglecting various kinds of federated overheads.

This paper introduces *SimLess*, an FC simulation framework for accurate FC simulations across multiple FaaS providers with a simple and lightweight parameter setup. Unlike the costly approaches that use machine learning over time series to predict the FC behavior, *SimLess* introduces two light concepts: (1) *twins*, representing the same function deployed with the same computing, communication, and storage resources, but in other regions of the same FaaS provider, and (2) *siblings*, representing the same function deployed in the same region with different computing resources. The novel *SimLess* FC simulation model splits the *round trip time* of a function into several parameters reused among twins and siblings without necessarily running them. We evaluated *SimLess* with two scientific FCs deployed across 18 AWS, Google, and IBM regions. *SimLess* simulates the cumulative *overhead* with an average inaccuracy of 8.9 % without significant differences between regions for learning and validation. Moreover, *SimLess* uses measurements of a low-concurrency FC executed in a single region to simulate a high-concurrency FC with 2,500 functions in the other areas with an inaccuracy of up to 9.75 %. Finally, *SimLess* reduces the parameter setup effort by 77.23 % compared to other simulation approaches.

CCS CONCEPTS

• **Computing methodologies** → **Modeling and simulation**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

FaaS, modeling, performance, serverless, simulation, workflows.

ACM Reference Format:
Sashko Ristov, Mika Hautz, Christian Hollaus, and Radu Prodan. 2022. *SimLess*: Simulate Serverless Workflows and Their Twins and Siblings in Federated FaaS. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3542929.3563478

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563478

1 INTRODUCTION

Function-as-a-Service (FaaS) is a serverless computing technology, widely used in scientific computing in recent years [26, 43, 62]. Scientists build complex applications by orchestrating *serverless functions* in workflows or *function choreographies* (FCs) for batch [53], event-based [2], or real-time processing [27]. This paper focuses on batch processing FCs.

1.1 Motivation: FCs in federated FaaS

Cloud users usually deploy serverless applications across multiple cloud regions. Existing reports [17] state that half of the users deploy functions in more than one AWS region, and almost 10 % use more than three regions. For instance, users may deploy serverless API to multiple regions and run their serverless applications at a global scale [6]. Other users may exploit multi-clouds [42] by running AWS Lambda functions that store voice recording on the cheaper AWS Simple Storage Service (S3) combined with Google Speech-to-Text from Google Cloud Functions for higher accuracy. Moreover, many researchers reported that running workflow applications in federated clouds may reduce costs [20] or makespan with budget constraints and increase resource utilization [19], for example, by offloading tasks of an FC from the local edge devices [3, 49] or monolithic applications [10, 45] to multiple *federated FaaS* providers. Finally, FCs may achieve higher scalability [54] and resilience [52] in federated FaaS compared to a single cloud region. For conciseness, we will use the terms *function*, *provider*, and *region*, for serverless function, FaaS provider, and cloud region.

1.2 State-of-the-art limitations

Despite these benefits, researchers face the challenge of determining the "optimal" setup of FCs in federated FaaS that maximizes performance and minimizes costs, which requires an accurate estimation model of FC function executions. For complex problems, such as resource management or scheduling, researchers rely on simulators that allow cheaper, faster, and more flexible search for optimized parameter setups without costly development, deployment, and long FC executions. Unfortunately, the high computing and communication heterogeneity of federated FaaS infrastructures hinders accurate simulations because of three limitations.

1.2.1 Costly and tedious FC parameter setup in federated FaaS. Typical HTTP function invocations may generate considerable communication latency among the wide variety of users and provider regions (e.g., AWS, Google, Microsoft, Alibaba, IBM), possessing heterogeneous infrastructures [33], proprietary capacity, and scheduling policies [66]. Finding the "optimal" setup for an FC is a complex and costly operation involving an exponential number of search space possibilities across multiple providers, regions, and memory allocated to each function (among others). The infrastructure and memory assigned to a function affect the execution time, while the region proximity and provider's scheduling policy impact the overall overhead.

1.2.2 Lack of FC simulation models in federated FaaS. Existing FaaS simulation models primarily focus on homogeneous infrastructures within a single region hosting individual independent functions and ignore FC dependencies involving considerable overheads. Recent work [64] reported that half of the scientific applications contain more than one function, needing an FC simulation model. Other models use machine learning to predict the execution time of a function but neglect the inter-region latency. Considering these limitations, existing FaaS simulation models are hardly applicable to scientific FCs that usually transfer enormous amounts of distributed data and starve for resources that exceed the limit of 1,000 functions per region [50].

1.2.3 Serverfull workflow simulation model not applicable to FCs. Existing simulators focus on cloud infrastructures operating virtual machines and containers. Despite the accurate simulation of *serverfull* workflows, they need substantial revision for *stateless* FaaS workflows and resources. Serverfull simulation models for federated clouds assume that all tasks may run on each resource, usually determined by schedulers during runtime. In contrast, the FaaS paradigm specifies the function code, runtime system, and allocated resources (i.e., RAM and CPU) during function deployment. Unfortunately, the community lacks an accurate simulation model for serverless workflows (FCs), especially in federated FaaS.

1.3 Contribution: FC simulation in federated FaaS

We propose *SimLess*, a framework to simulate FCs across multiple providers and estimate functions' *round trip time (RTT)*, even without executing them. To our knowledge, *SimLess* is the first FC simulation model in federated FaaS that considers various overheads between users and providers related to network, concurrency, authentication, session, and FaaS. *SimLess* composes FCs in Abstract Function Choreography Language (AFCL) [53] that describes FCs as a high-level workflow of abstract functions f, decoupled from the *function implementations* $f\mathcal{I}$. Each function implementation $f\mathcal{I}$ may have multiple *function deployments* $f\mathcal{D}$ described, among others, by their deployment region and allocated memory, classified in this paper as *twins* and *siblings*.

1.3.1 Twins. We define the twins of an $f\mathcal{D}$ as all deployments on identical computing and communication resources but in different regions of the same provider, including any backend-as-a-service (BaaS) used by the function. All twins use the same common environment but with different overheads. Consequently, an $f\mathcal{D}$ has twins only if accesses storage in the same region. The functions that access the storage of another region do not have twins since the communication cost between regions is not reproducible for the $f\mathcal{D}$s in other regions. For instance, the twins of an $f\mathcal{D}$ `AWS_FRANKFURT_128` deployed with 128 MB and access to

an S3 storage in Frankfurt are all $f\mathcal{D}$s of the same $f\mathcal{I}$ deployed with 128 MB on other AWS regions than Frankfurt, but with access to the S3 storage within their region. The $f\mathcal{D}$ `AWS_FRANKFURT_128` with access to the S3 storage of other AWS regions (e.g., Tokyo, Virginia) does not have twins.

1.3.2 Siblings. We define the siblings of an $f\mathcal{D}$ as all deployments in the same region but with different allocated memory sizes, reporting different performances. All siblings share and reuse all parameters of their deployment region. For instance, the siblings of `AWS_FRANKFURT_128` $f\mathcal{D}$ are all $f\mathcal{D}$s deployed in AWS Frankfurt (`eu-central-1`) with a different memory size (than 128 MB). The storage access restriction defined for twins does not apply to siblings. Consequently, the `AWS_FRANKFURT_128` $f\mathcal{D}$ has siblings even if it accesses the storage of another region, as siblings have different computations but the same storage access costs.

1.3.3 SimLess simulation model. *SimLess*'s novelty is to reuse the parameters among the twins and siblings of an $f\mathcal{D}$ and to simulate their *RTT* without executing them. The *SimLess* FC simulation model extracts the overhead parameters from the *RTT* of an $f\mathcal{D}$ and applies it to its twins and siblings. Consequently, it estimates the *RTT* of all twins and siblings of a function without running them. For instance, if $RTT = 696$ ms for a function running on AWS Frankfurt, *SimLess* estimates that its twins in AWS California and Tokyo need $RTT = 1{,}468$ ms and $RTT = 1{,}747$ ms, respectively. On the other hand, if the twin in Tokyo runs a parallel loop with 1,000 iterations, it finishes within 3,967 ms due to the concurrency overhead. Finally, if the FC starts with the parallel loop, it runs 550 ms longer due to the session overhead.

1.3.4 SimLess evaluation. We developed the *SimLess* simulation framework to evaluate the *SimLess* FC simulation model. First, we run a single "no-op" FC in a few regions to determine the correct *SimLess* parameter setup. Then, we use the parameters to simulate two orthogonal scientific FCs, widely used by researchers: embarrassingly parallel Monte Carlo (MC) and data-bound Burroughs-Wheeler Alignment (BWA) [35] with a complex FC structure using cloud storage and external executables. We used 18 regions of AWS, IBM, and Google across three continents, running a total of 69,720 functions. The results revealed that *SimLess* estimates the *RTT* of twins with an average inaccuracy of 8.2 % for embarrassingly parallel MC and 15.9 % for BWA. A sensitivity analysis reported 77.23% cost savings by reusing the parameters for twins and siblings.

1.3.5 Contribution. This paper brings several contributions:

- publicly available semi self-configurable *SimLess*[1] simulation framework of FCs in federated FaaS;
- a novel FC-agnostic *SimLess* FC simulation model with 9.75% inaccuracy for twins of FCs with high concurrency of up to 2,500 functions;
- negligible costs for simulation parameter setup and 77.23 % lower costs compared to conventional methods that require to run all functions for all provider regions.

1.4 Outline

The paper has eight sections. Section 2 presents a preliminary investigation of the parameters that affect *RTT* of a function. Section 3 describes the FC application and deployment underlying the novel *SimLess* FC simulation model presented in Section 4. Section 5 details the *SimLess* system architecture. Section 6 discusses the thorough validation and evaluation results, followed by the *SimLess*'s advances beyond the state-of-the-art, along with its limitations, cost savings, and additional features in Section 7. Finally, Section 8 concludes the paper and summarizes the future work.

2 PRELIMINARY OBSERVATIONS

We conducted preliminary experiments using the *xAFCL* [54] FC management system to investigate the impact of the provider, region, authentication, and its internal FC position and dependencies on the function *RTT*. We conclude with five critical observations concerning the functions' *RTT*.

2.1 Regions similarity

Our evaluation of 21 AWS regions using CloudPing [34] revealed similar internal latency within regions of 1.25 ms to 5.25 ms, with an average of 3.44 ms. Additionally, AWS reported a stable bandwidth between virtual private clouds and S3 of up to 100 Gbit/s within a region[2]. Our recent study [51] on the cold and warm spawning of AWS, Google, and IBM functions revealed that homogeneous resources exhibit similar behavior. Especially AWS assigned 99.93 % of Intel Xeon E5-2670 processors during a month [33].

Corollary. A function using the internal storage of the same region will behave similarly if migrated as a twin to another region of the same provider.

On the other side, distributing a function and its storage access to different regions causes different data transfer times due to communication variability. Such functions do not have twins. For instance, the average latency of the AWS Frankfurt region measured using CloudPing is 18.22 ms to the EU, 122.45 ms to the US, and 189.63 ms to the Asia Pacific.

[1] https://github.com/sashkoristov/enactmentengine

[2] https://aws.amazon.com/premiumsupport/knowledge-center/s3-maximum-transfer-speed-ec2/

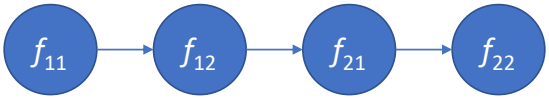


Figure 1: A "no-op" FC for function-agnostic overhead measurement and modelling.

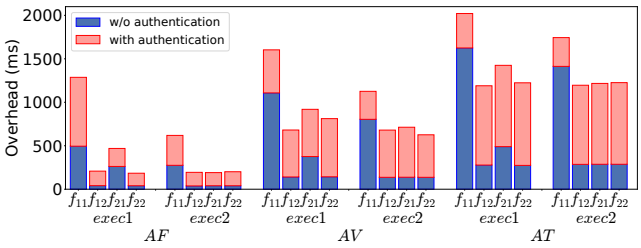


Figure 2: Overhead in two consecutive no-op FC executions on three regions: AWS Frankfurt (AF), North Virginia (AV), and Tokyo (AT).

2.2 Session and network overhead

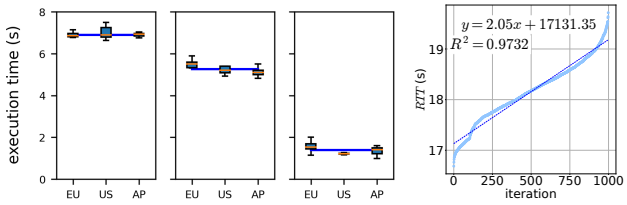
We investigated a *no-op* FC as a sequence of four functions, displayed in Fig. 1. We used two deployments $f\mathcal{D}_1$ and $f\mathcal{D}_2$ of the same no-op Python $f\mathcal{I}$ to determine the overhead of their position in an FC. The functions f_{11} and f_{12} run the $f\mathcal{D}_1$ deployment, while the functions f_{21} and f_{22} invoke $f\mathcal{D}_2$. We executed the no-op FC twice with authentication and twice without authentication in three AWS regions displayed in Fig. 2: Frankfurt (AF), Virginia (AV), and Tokyo (AT). The first execution (exec1) was a cold start, while the second execution (exec2) was a warm start.

Observation 1: Start functions have a higher overhead for cold and warm starts. We noticed a considerable overhead in the cold start exec1 of f_{11} and f_{21} [8, 38], much higher for f_{11}, however. Surprisingly, there was a notable overhead in the warm start exec2 of f_{11} too. Further investigation revealed that the initial region setup and session creation caused these overheads. We focus in this paper on the warm start model since *SimLess* is not aware if the function runs on an existing (warm) or a newly (cold) created container.

Observation 2: The overhead depends on the network proximity. The average overhead for functions increased with the distance to the region. We further noticed, e.g., an average warm start (exec2) overhead for functions f_{12}, f_{21}, and f_{22} of 39 ms in AF, 137 ms in AV, and even 287 ms in AT.

2.3 Authentication overhead

Observation 3: Authentication overhead depends on network proximity. Fig. 2 shows an additional authentication overhead that depends on the network proximity of the region.



(a) Average twin *ET* on AWS, Google, and IBM. **(b) 1,000 concurrent functions on AF.**

Figure 3: Preliminary simulation of a Monte Carlo π approximation.

The authentication overhead increases the average warm start (exec2) network overhead of the f_{12}, f_{21}, and f_{22} functions from Observation 2 by a factor of three on average, representing 156 ms in AF, 536 ms in AV, and 926 ms in AT.

2.4 Twins performance

Generally, a function's *execution time* (*ET*) affects its *RTT*, which mainly depends on the code and its assigned resources. We, therefore, investigated the twins' *ET* across multiple regions by running an MC approximation of π using a parallel loop of nine functions deployed with 128 MB of RAM on three regions: Frankfurt (EU), Virginia / Washington (US), and Tokyo (AP). We used the Serverless Application Analytics Framework [16] to measure the *ET* without introducing overhead inside the function code.

Observation 4: Twins have similar ET. Fig. 3a illustrates significant *ET* differences per provider, despite the same memory size allocated for all functions. The functions run fastest on IBM (1.4 s on average), followed by Google and AWS, which run with 3.5, respectively, five times longer. Nevertheless, the twins reported similar *ET* across all three regions and differed by up to 0.8 % from the average on AWS, 4.1 % on Google, and 13.6 % on IBM. The deviance is smaller than *ET*'s deviance within a region. Notably, the MC functions do not use external cloud services, which may cause different performance breaches in this observation. Our evaluation in Section 6.3 shows that this observation holds even for functions accessing storage in the same region.

2.5 Concurrency overhead

Spawning numerous concurrent functions is the main benefit of orchestrating functions in an FC. However, many researchers reported several seconds for spawning hundreds of functions on AWS Step Functions [53] and IBM [30]. On the other side, [40] reported the workload affecting functions' *RTT*. Consequently, models that use average values for concurrency overheads may result in substantial *RTT* estimation inaccuracy. To investigate the function concurrency

overhead, we executed a parallel loop with 1,000 functions of the same Monte Carlo simulation from Section 2.4 on AWS Frankfurt and measured RTT as presented in Fig. 3b.

Observation 5: Spawning numerous concurrent functions generates additional function overhead. Against the expectation of all functions finishing simultaneously, we observed a nearly linear cumulative overhead of 2.1 ms per function leading to significant inaccuracies if ignored. For example, a function with $RTT = 500$ ms running in a parallel loop with 1,000 iterations finishes after:

- 0.5 s using existing FaaS models that do not consider the concurrency overhead;
- $\frac{\sum_{k=0}^{999}(0.5+2.1\cdot k)}{1000} = 1.55$ s using the average delay of all 1,000 iterations;
- $0.5 + 2.1 = 2.6$ s using this observation.

Even worse, existing models consider the same ET for all 1,000 iterations, thereby simulating with higher inaccuracy. Notably, we omitted in this investigation a negligible 0.16 ms increase in the functions start time of the parallel loop.

3 FAAS SYSTEM MODEL

This section presents the FC application and FaaS deployment models used in the *SimLess* FC simulation model.

3.1 FC application model

We model a *function choreography* (FC) as a directed acyclic graph (F, D) with a set $F = \bigcup_{i=1}^{n} f_i$ of *functions* and a set $D = \{(f_i, f_j) \mid (f_i, f_j) \in F \times F\}$ of *precedence dependencies*. A dependency $(f_i, f_j) \in D$ expresses that f_j must start after f_i finishes, where $f_i \in pred(f_j)$ represents the predecessor set of f_j that includes f_i. Each FC has one or multiple *start functions* f_s with no predecessors and an *end function* f_e with no successors. A no-op f_e with $RTT = 0$ can express multiple exit functions and output data staging.

We assume a non-preemptive scheduling model since functions are stateless and impossible to suspend and resume. We neglect the transmission time between functions (i.e., time to "put bits on the wire") because providers limit the size of input and output data. Therefore, the propagation time (i.e., time a bit "travels through the wire") mainly impacts the network latency.

3.2 FaaS deployment model

We observe two differences between serverless FCs and serverfull workflows running on virtual machines.

Notion of resource. FaaS binds resources with a function, called *function deployment* $f\mathcal{D}$. While serverfull workflows clearly distinguish between resources (virtual machines) and application executables (e.g., JAR files, Python scripts, shell scripts, container images), functions need deployments, URLs, or names for invocation. During deployment, users must map the resources (i.e., provider, region, memory) along with the deployment package or *function implementation* $f\mathcal{I}$ (i.e., function code, including package dependencies [63]).

Data transfer between functions. Many serverfull workflow management systems (e.g., Pegasus [18]) or simulators (e.g., WorkflowSim [14]) separate data transfers between tasks from computations. Tasks usually exchange data through files that the workflow management system copies to the target virtual machines. In the FaaS deployment model, functions are stateless, invocations contain the data input, and return the data output in JSON format. Therefore, functions may exchange data implicitly through references, and all incoming data transfers happen inside functions during runtime, as files need to reside in the local file system. In other words, the RTT includes the data transfers.

Therefore, we model *measured completion time* $\widetilde{ct}(f)$ of a function f as the latest completion time of its predecessors $pred(f)$ plus its *measured round trip time* $\widetilde{RTT}$:

$$\widetilde{ct}(f) = \max_{f_p \in pred(f)} \left[\widetilde{ct}(f_p)\right] + \widetilde{RTT}; \qquad \widetilde{M} = ct\,(f_e)\,. \tag{1}$$

Eq. (1) also expresses *measured makespan* $\widetilde{M}$ of an FC through the completion time $ct\,(f_e)$ of the end function f_e.

4 *SIMLESS* FC SIMULATION MODEL

This section presents the novel *SimLess* FC simulation model that reuses parameters among twins and siblings to minimize the parameter setup cost with a low inaccuracy. Table 1 summarizes all notations and definitions used in the paper. The notation $\widetilde{P}$ denotes the metric P measured in a real execution, while $\overline{P}$ indicates its simulated values. We further use P_T and P_S to denote the corresponding metric for the function twin and sibling.

The *simulated round trip time* $\overline{RTT}$ of f is the sum between the *simulated overhead* $\overline{O}$ and *simulated execution time* $\overline{ET}$:

$$\overline{RTT} = \overline{O} + \overline{ET}, \tag{2}$$

where $\overline{O}$ includes all delays from the provider and the latency between the FC management system and the region, while $\overline{ET}$ represents the actual execution time of f. We define their simulation models in the following two sections.

4.1 Overhead $\overline{O}$ simulation

We further split the simulated overhead $\overline{O}$ into: (1) *session overhead* $\overline{SO}$ for the start function, caused by creating a session between the FC management system and the provider, (2) *network overhead* $\overline{NO}$ to the corresponding region of the running function, (3) *authentication overhead* $\overline{AO}$ of a function, (4) *FaaS overhead* $\overline{FO}$ for the provider to start the

Table 1: Notation summary.

Notation	Definition
RTT	Round trip time of f
ET	Execution time of f
O	Overhead of f
FO	FaaS overhead for starting f
SO	Network overhead of f
d	Average concurrency invocation delay
CO	Concurrency overhead for a parallel loop
AO	Authentication overhead of f
cr	Cryptography overhead
$ct(f)$	Completion time of f
M	Makespan of FC (simulated and measured)
δ_P	Simulation inaccuracy of parameter P
$\overline{P}$	Simulated value of parameter P
$\widetilde{P}$	Measured value of parameter P
P_T	Parameter P of function twin
P_S	Parameter P of function sibling

function, and (5) *concurrency overhead* $\overline{CO}$ upon starting multiple functions by the provider:

$$\overline{O} = \overline{SO} + \overline{NO} + \overline{AO} + \overline{FO} + \overline{CO} \tag{3}$$

We explain each overhead in detail in the following sections, based on the observations from Section 2.

Session overhead. $\overline{SO}$ of the start function f_s includes the overhead for DNS resolution and SDK parameter setup, based on Observation 1 (Section 2.2).

Network overhead. $\overline{NO}$ is the average measured network latency $\widetilde{NO}$, modeled based on the Observation 2 (Section 2.2). $\overline{NO}$ depends on the network proximity of the target region.

Authentication overhead. $\overline{AO}$ depends on the protocol used by the provider. For example, a two-way handshake authentication requires two round trips to the authentication server to establish a TCP connection (`SYN` and `ACK`) and one for the authentication itself (HTTP). On the other hand, one-way authentication requires only two round trips to the target region. We, therefore, model the r-way authentication overhead $\overline{AO}$ through the network overhead $\overline{NO}$ affected by the network proximity, based on Observation 3 (Section 2.3):

$$\overline{AO} = \overline{cr} + (r+1)\cdot\overline{NO} = \begin{cases} \overline{cr} + 3\cdot\overline{NO}, & \text{2-way authentication;} \\ \overline{cr} + 2\cdot\overline{NO}, & \text{1-way authentication.} \end{cases} \tag{4}$$

Cryptography overhead. $\overline{cr}$ in Eq. 4 is the time the provider needs to run the actual authentication. We assume that the cryptography overhead $\overline{cr}$ is constant in all regions of a provider and reusable for any hosted function.

Table 2: Reusable parameters for f's twins and siblings.

f	$\overline{SO}$	$\overline{NO}$	$\overline{cr}$	$\overline{AO}$	$\overline{FO}$	$\overline{d}$	$\overline{CO}$	$\overline{ET}$	$\overline{RTT}$
Twins	✓	$\widetilde{NO}$	✓	(4)	✓	✓	✓	✓	(7)
Siblings	✓	✓	✓	✓	✓	✓	✓	(8)	(8)

FaaS overhead. $\overline{FO}$ comprises the provider delay for receiving a request until starting the function. This overhead includes the service delay to forward the task to the endpoint and return the results, as well as the cold start of an enclosing container [13], among others. We assume that this overhead is constant for all regions of a provider.

Concurrency overhead. $\overline{CO}$ of a parallel loop, following on Observation 5 from Section 2.5, is a linear function of the *average concurrency invocation delay* d for $k-1$ iterations, since $\overline{FO}$ already comprises the first iteration overhead:

$$\overline{CO} = (k-1)\cdot\overline{d} \tag{5}$$

We describe the detailed methodology to determine $\overline{SO}$, $\overline{NO}$, $\overline{cr}$, $\overline{d}$, and $\overline{FO}$ in Section 6.2.1.

4.2 Function execution time $\overline{ET}$ simulation

We observe two ways to simulate the $\overline{ET}$ of a function f:

White-box $\overline{ET}$ *simulation.* may require significant development effort, such as: (1) extraction from provider logs (e.g., AWS Cloud Watch) or (2) adding an inspector within the function code (e.g., SAAF [16]).

Black-box $\overline{ET}$ *simulation.* used in *SimLess* measures the $\widetilde{RTT}$ and calculates $\overline{ET}$ based on Eq. (2): $\overline{ET} = \widetilde{RTT} - \overline{O}$.

4.3 Twins and siblings simulation

Learning the $\widetilde{RTT}$ of a function with more than 20 twins (for many provider regions) and hundreds of siblings (for different memory sizes) may be tedious, time-consuming, and costly. Therefore, we model all twins and siblings through the function $\widetilde{RTT}$ and several reusable overheads, displayed in Table 2. Since twin deployments are in different regions, we can reuse all parameters except $\overline{NO}$ and $\overline{AO}$, calculated based on the measured $\widetilde{NO}$ using Eq. (4) for $\overline{AO}$. On the other side, siblings cannot reuse the $\overline{ET}$ only. The $\overline{cr}$, $\overline{FO}$, and $\overline{d}$ overheads are constant for each provider and apply to any function running on any region. Finally, we model $\overline{RTT}_T$ and $\overline{RTT}_S$ through the function $\widetilde{RTT}$ in the next two sections.

4.3.1 Twins $\overline{RTT}_T$ simulation. The next equation holds for all twins, based on Observation 4 from Section 2.4 and Eq. (2):

$$\overline{ET}_T = \overline{ET} = \widetilde{RTT} - \overline{O} = \overline{ET} = \widetilde{RTT}_T - \overline{O}_T. \quad (6)$$

Since $\overline{SO}$, $\overline{cr}$, $\overline{FO}$, and $\overline{CO}$ are constant for twins (see Table 2), we calculate $\overline{RTT}_T$ based on Eqs. (3) and (4) as:

$$\overline{RTT}_T = \widetilde{RTT} + \overline{O}_T - \overline{O} = \widetilde{RTT} + \overline{AO}_T - \overline{AO} = \\ = \widetilde{RTT} + (r+1) \cdot (\overline{NO}_T - \overline{NO}). \quad (7)$$

Example. Let us assume a function deployed on AWS Frankfurt with 128 MB and executed once using warm start and authentication, with a measured $\widetilde{RTT} = 200$ ms. Let the network overhead to AWS Frankfurt be $\overline{NO} = 30$ ms. Following Eq. (4), $\overline{AO} = 3 \cdot \overline{NO} = 90\,\text{ms} + \overline{cr}$. Similarly, if the measured network overhead to AWS Virginia is $\overline{NO} = 140$ ms, then $\overline{AO} = 3 \cdot \overline{NO} = 420\,\text{ms} + \overline{cr}$. Applying Eq. (3) in Eq. (7), we obtain the $\overline{RTT}_T$ of the twin in AWS Virginia: $\overline{RTT}_T = 200 - (30 + 3 \cdot 30 + \overline{cr} + \overline{FO}) + (140 + 3 \cdot 140 + \overline{cr} + \overline{FO}) = 640$ ms, assuming that $\overline{cr}$ and $\overline{FO}$ are constant in both regions.

4.3.2 Siblings $\overline{RTT}_S$ model. All siblings have the same overhead $\overline{O}_S = \overline{O}$ of f since they reside within the same region.

State-of-the-art limitation. Many simulation frameworks, such as Workflowsim [14] and DynamicCloudSim [7], use a linear model to compute the execution time of a task when scaling the resources. However, this model is unsuitable for all tasks since speedup depends on sequential parts following Amdahl's Law [1]. Moreover, specific provider regions may behave differently when scaling the memory. For example, [12] reported a huge memory under-utilization on Google Cloud Functions and a linear benchmark application speedup of up to 512 MB, with no further performance gains despite the large memory size.

Scaled memory model. SimLess uses a scaling factor $s(m)$ to model the speedup for various sibling memory allocations, as defined in [32] for multiprocessors, where m represents the scaled memory. The factor $s(m)$ may lead to sublinear ($s(m) < 1$), linear ($s(m) = 1$), or superlinear speedup ($s(m) > 1$) [55]. We use the function execution time $\overline{ET}$ with the minimum possible memory of 128 MB as a baseline resource. Accordingly, we model the $\overline{ET}_S$ with the scaled memory m, and express $\overline{RTT}_S$ using $\widetilde{RTT}$ and $\overline{O}$ based on Eq. (2):

$$\overline{ET}_S = \overline{ET} \cdot \frac{s(m)}{m}, \quad \overline{RTT}_S = (\widetilde{RTT} - \overline{O}) \cdot \frac{s(m)}{m} + \overline{O} \quad (8)$$

For CPU-bound functions with $s(m) = 1$, Eq. (8) transforms into a linear speedup $ET_s = \frac{ET}{m}$. *SimLess* prioritizes the scaling factor if measured and provided by the user. Otherwise, it uses the default linear speedup.

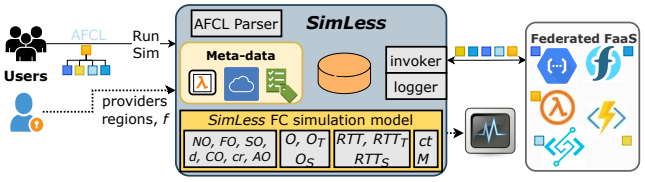


Figure 4: *SimLess* system architecture.

4.4 *SimLess* FC simulation accuracy

We use the simulated $\overline{RTT}$ in Eq. (1) to determine *simulated completion time* $\overline{ct}(f)$ of each function f in the FC. The simulated completion time of the end function $\overline{ct}(f_e)$ represents the *simulated makespan* $\overline{M}$ of the entire FC. *SimLess*'s goal is not to minimize the makespan (e.g., using a new scheduling algorithm) but to provide an accurate simulation of $\overline{RTT}$ for each FC function and the overall $\overline{M}$.

We define the simulation *inaccuracy* δ_P for various parameters $P \in \{O, RTT, ct, M\}$ as the relative deviation of the simulated $\overline{P}$ to the measured metric $\widetilde{P}$:

$$\delta_P = \left| \frac{\widetilde{P} - \overline{P}}{\widetilde{P}} \right| \cdot 100\%. \quad (9)$$

5 *SIMLESS* SYSTEM ARCHITECTURE

Fig. 4 presents the *SimLess* system architecture that implements the *SimLess* FC simulation model using several components described in the following paragraphs.

AFCL parser. Users specify FCs in AFCL and provide their input in JSON format. The AFCL parser translates the AFCL into an executable FC and reads the $f\mathcal{D}$s and the number of invocation instances.

Invoker. The invoker runs each $f\mathcal{D}$ following the control and data-flow specified in the AFCL definition of the FC.

Logger. The logger stores FC data in a MongoDB *execution log*, including the function name, URL, region, provider, allocated memory, start time, completion time, $\widetilde{RTT}$, and parallel loop (required by $\overline{CO}$). Finally, *SimLess* runs a CronJob to process the logs, update the average measured $\widetilde{RTT}$ for each executed $f\mathcal{D}$, and calculate the simulated $\overline{ET}_T$ and $\overline{ET}_S$.

Meta data. The user can configure the metadata for all $f\mathcal{D}$s, including the $\overline{FO}$, $\overline{SO}$, $\overline{d}$, and $\overline{cr}$ overheads for each provider and the $\overline{NO}$ parameter for each region.

SimLess FC simulation model. SimLess simulates the *RTT* of an $f\mathcal{D}$ (instead of invoking it), using the *SimLess* FC simulation model presented in Section 4. After simulating all $f\mathcal{D}$s in an FC, *SimLess* presents the $\overline{RTT}$ of each simulated $f\mathcal{D}$ and the overall $\overline{M}$ to the user console.

5.1 Priority-based simulation

SimLess uses three *priority rules* to estimate the $\overline{RTT}$ of $f\mathcal{D}$s:

(1) *Reuse* $\widetilde{RTT}$ measured in a previous $f\mathcal{D}$ execution;
(2) *Calculate* $\widetilde{RTT}_T$ using a previous twin run and Eq. (7);
(3) *Calculate* $\widetilde{RTT}_S$ using a previous sibling run and Eq. (8).

We give higher priority to twins following Observation 4 from Section 2.4, that their average $\widetilde{ET}$ deviance is lower than the $\widetilde{ET}$ deviance within a region. Scaling resources may provide sublinear speedup (even slowdown) or superlinear speedup [55]. Section 7.2 discusses the possible drawbacks.

5.2 Concurrency simulation

$f\mathcal{D}$s of a parallel loop have different $\widetilde{RTT}$ due to the concurrency overhead $\overline{CO}$, explained in Section 4.1 and Eq. (5). Therefore, *SimLess* removes $\overline{CO}$ and considers RTT of a non-parallel $f\mathcal{D}$ execution before updating it in the metadata by averaging it with the previously logged inv invocations:

$$\overline{RTT} = \frac{\overline{RTT} \cdot inv + (\widetilde{RTT} - \overline{CO})}{inv + 1}. \tag{10}$$

5.3 AFCL extensions

The semi-automated *SimLess* architecture uses AFCL [53] for FC definition, while function implementations may use arbitrary programming languages. Providers take a similar approach, such as AWS Step Functions using JSON and IBM Composer using JavaScript to orchestrate FCs. We added two AFCL extensions to support simulation.

`SimValue` *property.* of the `dataOut` port allows customizing a function output, since *SimLess* does not support dynamic runtime variables offered by ACFL. For example, suppose the dynamic output of a function represents the number of parallel loop iterations. In that case, we need to hardcode this parameter to a constant (e.g., five in Fig. 5) to allow its simulation without the function execution.

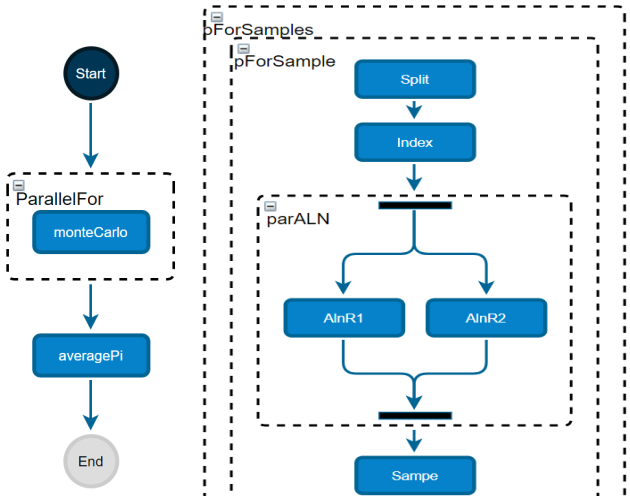
`deployment` *attribute.* An $f\mathcal{D}$ invocation by location uses the information (e.g., Amazon Resource Name) stored in the AFCL `resource` field. Although it contains the provider and the region, the absence of memory information prohibits *SimLess* from determining twins and siblings. The `deployment` attribute addresses this problem by providing the required provider, region, and memory information.

6 *SIMLESS* EVALUATION

This section presents the experimental design and discusses the diversity of used benchmark FCs. Afterward, we present the methodology to determine the parameter setup for all 18 regions, validated by comparing the measured $\widetilde{O}$ and simulated overhead $\overline{O}$. Furthermore, we present evaluation results with low and high concurrency FCs.

```
- function:
    name: "MonteCarlo"
    deployment: "AWS_eu-central-1_128" #  provider_region_memory
    dataOuts:
    - name: "counter"
      type: "number"
      properties:
      - name: "simValue"
        value: 5                # specified value for simulation
    properties:
    - name: "resource"
      value: "arn:aws:lambda:eu-central-1:xxx:function:MC"
```

Figure 5: AFCL example of an $f\mathcal{D}$ specifying (i) the simulated `dataOut` port `counter` and (ii) the `deployment` attribute to determine twins and siblings.



(a) MC FC. **(b) BWA FC.**

Figure 6: Experimental FCs for *SimLess* validation.

6.1 Experimental design

Our experimental design comprises a set of simulated applications and a validation methodology.

6.1.1 Simulated applications. We simulated complementary real-world compute and data-bound FCs, widely used by the serverless and workflow research communities.

No-op FC. presented in Fig. 1 studies the overheads by avoiding execution time ET instability. Researchers widely use no-op functions [13, 15, 30, 57, 67] that minimize the impact of cloud performance instability [7, 33, 59] and provide an RTT comprising only overheads (i.e., $\widetilde{RTT} = \widetilde{O}$).

Monte Carlo (MC) π approximation. is an embarrassingly parallel FC (Fig. 6a) that scales with the number of `monteCarlo` functions in a parallel loop. Afterwards, the `averagePi` function collects results to approximate the π number. MC is a

compute-bound FC intensively used in serverless research [4, 5, 22, 23, 52, 60, 61], and its functions do not use BaaS. We used two concurrency sizes in our evaluation: (1) MC_{10} for low-concurrency with nine `monteCarlo` functions, and (2) $MC_{1,000}$ for high-concurrency 999 `monteCarlo` functions.

Burroughs-Wheeler Alignment (BWA) [35]. is an FC that maps low-divergent sequences against a large reference genome from the Escherichia coli DNA. BWA is popular in research [25, 54, 56, 68] due to its compute and data-bound requirements. BWA orchestrates five functions in a complex FC with two nested parallel loops and a parallel section. BWA reads two 50 MB files storing sample reads and one file (5 MB) storing the reference genome (`sample`). BWA starts with an outer parallel loop `pForSamples` that computes multiple samples. BWA runs four instances for each sample, leading to 20 functions per sample. We explore two concurrency sizes: (1) BWA_{20} computing a single sample with low-concurrency, and (2) $BWA_{2,500}$ running 2,500 functions that process 125 samples with high-concurrency. The BWA functions access storage and generates around 1 GB of data per sample.

6.1.2 Validation methodology. We developed 54 functions of three FCs, deployed in 18 carefully-selected regions of three providers (i.e., AWS, Google, IBM) with significant diversity in network proximity across three continents (see Table 3). For each provider, we used three regions for simulating the parameter setup and the other three regions of the respective continents for validating them. We performed no execution on any region for validation and observed no significant difference between learning and validation regions (see Section 6.2.2). For conciseness, we denote each region by concatenating the provider's and region's first letters. For instance, `AF` denotes AWS Frankfurt.

To generalize our findings, we ran a total of 69,720 functions. We repeated every experiment five times, similar to recent work [11], and calculated the average of the last four to omit the cold start effects. We conducted all experiments in the morning (CET) and did not detect significant performance differences in the investigated regions of the three providers. We carefully selected the FCs' problem size to produce a makespan shorter than the keep-alive time so that each execution was a warm start for all functions.

6.2 *SimLess* parameter setup

This section validates all parameters affecting the overhead.

6.2.1 Parameter setup methodology. Fig. 7 presents a four-step methodology to set up the parameters of the *SimLess* FC simulation model. Each step determines one or two parameters and uses all known parameters in the next steps.

(1) $\overline{NO}$: We measure $\widetilde{NO}$ from the University of Innsbruck to each region using CloudPing [34] for AWS regions, GCPing [24] for Google regions, and the overhead to the closest AWS region for IBM without a known tool.

(2) $\overline{FO}, \overline{SO}$: We run the no-op FC without authentication and calculate $\overline{FO} = \widetilde{RTT} - \overline{NO}$ for the last three functions (warm start). Next, we obtain $\overline{SO}$ by subtracting the average $\widetilde{RTT}$ of the last three functions from the $\widetilde{RTT}$ of the first.

(3) $\overline{AO}, \overline{cr}$: We calculate $\overline{AO}$ by subtracting the unauthenticated $\widetilde{RTT}$ from the authenticated $\widetilde{RTT}$. We obtain $\overline{cr}$ using Eq. (4) considering a two-way authentication ($r = 2$) for AWS and a one-way ($r = 1$) for IBM, while $\overline{AO} = 0$ for Google.

(4) $\overline{CO}$: Since the no-op FC has no concurrency, we use the MC FC to determine the average concurrency invocation delay $\overline{d}$ and $\overline{CO}$, using Eq. (5).

After repeating this methodology for learning three provider regions (see Table 3), we calculate the average $\overline{SO}$, $\overline{cr}$, and $\overline{FO}$ parameters for each provider. Finally, we apply Eq. (3) to calculate each region's overhead $\overline{O}$.

6.2.2 Parameter setup validation. We use the no-op FC (with $\overline{ET} = 0$) to calculate the parameters that affect the overhead $\overline{O}$ and the inaccuracy δ_O by comparing the simulated $\overline{O}$ and the measured $\widetilde{O}$. We observe in Table 4 that the closest regions in Frankfurt report a higher inaccuracy of up to $\delta = 19.6\,\%$ for `AF` because the relative deviance is higher for smaller overheads $\widetilde{O}$. For AWS and IBM regions in US and Asia Pacific, the inaccuracy $\delta_O \leq 4.16\,\%$, while the maximum absolute deviance for all regions is 37 ms for `AF`. Google generated the lowest $\widetilde{O}$, which does not include $\overline{AO}$ that may cause higher inaccuracies for `GV`, for example.

Table 3: Learning and validation regions for *SimLess* parameter setup.

Goal	*Cont.*	*AWS*	*IBM*	*Google*
Learn	EU_1	`AF`: Frankfurt	`IF`: Frankfurt	`GB`: Belgium
	US_1	`AV`: Virginia	`IW`: Washington	`GV`: Virginia
	AP_1	`AT`: Tokyo	`IT`: Tokyo	`GT`: Tokyo
Validate	EU_2	`AL`: London	`IL`: London	`GL`: London
	US_2	`AC`: California	`ID`: Dallas	`GI`: Iowa
	AP_2	`AS`: Sidney	`IS`: Sidney	`GH`: Hong K.

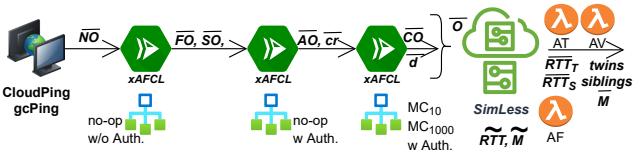


Figure 7: Overhead $\overline{O}$ computation methodology for twins' and siblings' $\overline{RTT}$ estimation without execution.

Table 4: Parameter setup results (in ms) for no-op FC, with the maximum inaccuracy δ_O per provider in bold.

Region	$\overline{SO}$	$\overline{NO}$	$\overline{cr}$	$\overline{AO}$	$\overline{FO}$	$\overline{O}$	$\widetilde{O}$	δ_O[%]
AF		30		166		226	189	**19.6**
AV	550	140	76	496	30	666	676	1.48
AT		267		877		1174	1200	2.17
IF		30		136		223	204	**9.31**
IW	152	140	76	356	57	553	577	4.16
IT		267		610		934	930	0.43
GB		59				82	95	13.7
GV	112	131	–	–	23	154	131	**17.6**
GT		275				298	307	2.93

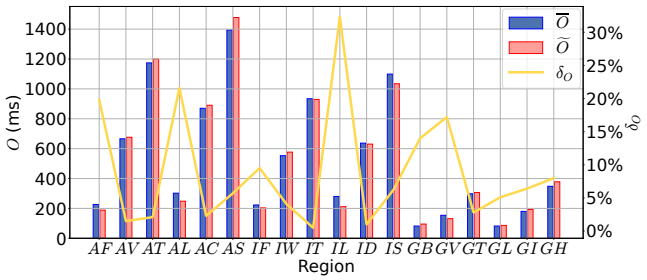


Figure 8: Simulated versus measured overheads of no-op FC and their inaccuracy.

6.2.3 Overhead $\overline{O}$ validation. After configuring the simulation parameters for each region, we validate them for the twins deployed in other regions (see Table 3). We reuse the overheads $\overline{cr}$ and $\overline{FO}$ from Table 4 in all regions of a provider and measure $\widetilde{NO}$ in each region to calculate $\overline{AO}$. Fig. 8 reports an average inaccuracy $\delta_O = 8.9\,\%$ for all six regions of each provider. We see no major difference between the three regions used for learning the parameters and the other three used for validating them. On average, the inaccuracy worsens between $\delta_O = 7.9\,\%$ to $9.8\,\%$, mainly due to IL's high inaccuracy of $32.3\,\%$ caused by its limited computational capabilities, inefficient container management, slow network [41], or free use at IBM. The average inaccuracy decreased to $\delta_O = 7.5\,\%$ excluding IL. In general, the inaccuracy is higher for "closer" regions since even a small absolute difference in overhead causes a high relative error.

6.2.4 Average concurrency invocation delay $\overline{d}$. We ran $MC_{1,000}$ on AF, AV, and AT to determine the concurrency invocation delay $\overline{d}$ by calculating the $\widetilde{RTT}$ increase for all 999 monteCarlo functions. The results were stable for the three regions (i.e., 2.1 ms, 2.2 ms, 2.4 ms), or $\overline{d} = 2.23$ ms on average.

Provider scalability. We restrict our analysis to AWS regions that scale up to 1,000 concurrent functions, while Google and IBM suffer from scalability problems, as reported in [30, 66]. Indeed, running over 100 concurrent functions on Google and IBM resulted in connection interruptions or considerable outliers, such as increases in $\widetilde{RTT}$ of up to 27.2 s in GT, 35.8 s in IW, and even 57.6 s in IL. Such performance instability analysis goes beyond the scope of this paper, which focuses on the concurrency overhead.

6.3 Low concurrency evaluation

We evaluated *SimLess* using the low-concurrency MC_{10} and BWA_{20} FCs using the parameter setup determined by the no-op FC. We conducted a sensitivity analysis to estimate the $\overline{RTT}_T$ of twins by using the measured $\widetilde{RTT}$ of $f\mathcal{D}$s with the highest inaccuracy $\overline{\delta}_O$.

6.3.1 MC_{10} FC evaluation. We implemented and deployed MC_{10} on each provider with 128 MB and ran it on all 18 regions presented in Table 3. We used the measured $\widetilde{RTT}$ for the AF, IF, and GV regions with the maximum δ_O per provider (see Table 3) to simulate the $\overline{RTT}_T$ and makespan $\overline{M}$ of the corresponding twins in other regions. Fig. 9 presents the simulated $\overline{RTT}_T$, $\overline{M}$ and the measured $\widetilde{RTT}_T$, $\widetilde{M}$ including δ_{RTT} and δ_M of the monteCarlo twins for the EU and US regions, due space limitations. We included $\overline{SO}$ in the overhead of monteCarlo, which is the start function of MC_{10}.

Google performance instability. We experienced an enormous $\widetilde{RTT}$ deviance of around two anchors for all six Google regions, in particular $\widetilde{RTT} = 3.5$ s for GH and 18.6 s for the other regions. A detailed investigation using the SAAF inspector [16] revealed an unusually high warm start $\widetilde{ET}$ for the first function execution and partially for the second (warm) repetition. Starting from the third (warm) repetition, $\widetilde{RTT}$ stabilized towards the higher anchor $\widetilde{RTT} = 18.6$ s. To overcome this instability, we used the higher Google $\widetilde{ET}$ since the highest completion time of all functions in a parallel loop affects the makespan.

monteCarlo *RTT inaccuracy.* Fig. 9 displays an average inaccuracy of $\delta_{RTT} = 6.3\,\%$ for all regions, except AF, IF and GV with $\delta_{RTT} = 0$. *SimLess* simulation model showed an average inaccuracy of $\delta_{RTT} = 1.2\,\%$ for AWS, $4.9\,\%$ for IBM, and $12.7\,\%$ for Google twins, with the highest inaccuracy of $24.1\,\%$ for GI. The averagePi function shows a similar average inaccuracy of $\delta_{RTT} = 2.1\,\%$ for AWS, $11.1\,\%$ for IBM, and $3.3\,\%$ for Google twins.

Makespan inaccuracy. Fig. 9 shows a δ_M similar to δ_{RTT} of the monteCarlo function, which attenuates the impact of averagePi. On average, *SimLess* generates $\delta_M = 8.8\,\%$, ranging between $1.3\,\%$ for AWS twins and $19.1\,\%$ for Google

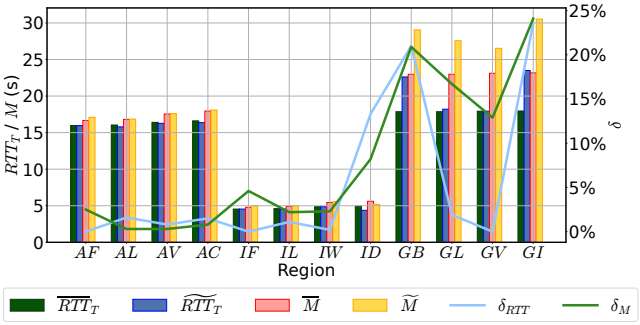


Figure 9: Simulated versus measured RTT and makespan of `monteCarlo` twins, and their inaccuracy.

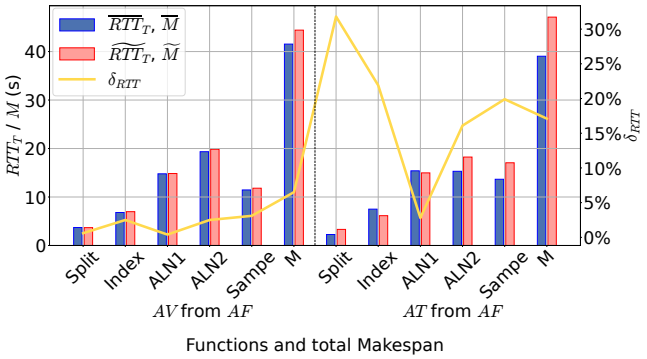


Figure 10: Simulated versus measured RTT of BWA_{20} twins and their inaccuracy.

twins. Notably, $\overline{M} \neq \widetilde{M}$ and $\delta_M > 0$ although $\delta_{RTT} = 0$ for the `AF`, `IF`, and `GV` regions, since the `monteCarlo` function with the largest $\widetilde{RTT}$ in the parallel loop decides its makespan.

6.3.2 BWA_{20} FC evaluation. BWA is a complex FC whose functions run external executable files and access storage, requiring considerable development and porting effort on various providers. Therefore, we only ported the functions for AWS and IBM and deployed them on `AF`, `AV`, `AT`, `IL`, `ID`, and `IW` regions with 1 GB of memory. We configured all functions to access the closest AWS S3 from `AF`, `AV`, and `AT`.

AWS inaccuracy. Similar to MC_{10}, we ran BWA_{20} in the `AF` region and used the measured $\widetilde{RTT}$ to determine the simulated $\overline{RTT}_T$ of the function twins in the `AT` and `AV`. Fig. 10 compares the measured $\widetilde{RTT}_T$ and the simulated $\overline{RTT}_T$, showing an average inaccuracy of $\delta_{RTT} = 1.8\,\%$ for all five BWA_{20} twins in *AT*. Similarly, we observed an average inaccuracy of $\delta_{RTT} = 3.4\,\%$ for all BWA_{20} function twins in *AV*. *SimLess* generated a high makespan inaccuracy of $\delta_M = 6.5\,\%$ for `AT` and 7.1 % for *AV*, by accumulating all function inaccuracies except `Aln1`, whose RTT is smaller than `Aln2` in the parallel section. The simulated $\overline{M}$ considers the average measured $\widetilde{RTT}$ in the parallel loop, which is smaller than measured $\widetilde{M}$ that uses the maximum, confirming the observation in [56].

IBM inaccuracy. We applied the same methodology for IBM and used the measured $\widetilde{RTT}$ of the functions in `IL` to simulate $\overline{RTT}_T$ of the twins in `ID` and `IW`. However, *SimLess* generated a higher inaccuracy of $\delta_{RTT} = 18.2\,\%$ for `ID` and even 35.1 % for `IW` because of unstable WAN links between IBM functions and AWS S3. Surprisingly, the simulated $\overline{M}$ was more accurate with $\delta_M = 18.5\,\%$ for `IL` and 18.2 % for `IW`. For some functions, the simulated $\overline{RTT}_T$ was higher than the measured $\widetilde{RTT}_T$ with a lower cumulative inaccuracy δ_M.

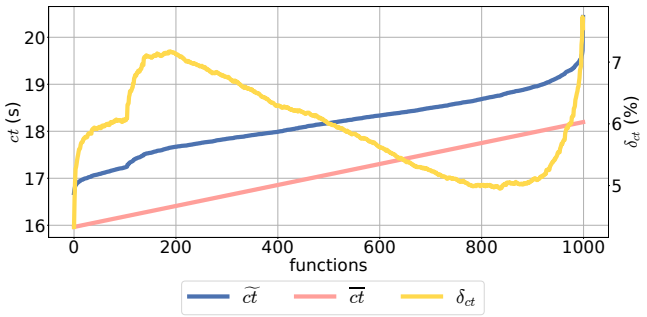


Figure 11: Simulated versus measured completion time of $MC_{1,000}$ functions on `AF` and their inaccuracy.

6.4 High concurrency evaluation

We evaluate *SimLess* using high-concurrency by running $MC_{1,000}$ and $BWA_{2,500}$ FCs. Since the concurrency limit AWS Lambda functions (1,000) is lower than of $BWA_{2,500}$, we present $\widetilde{ct}$ to better capture the model accuracy.

$MC_{1,000}$ FC. Fig. 11 presents the measured completion time $\widetilde{ct}$ of the `monteCarlo` function executed on *AF*, and its simulated $\overline{ct}$ after applying average concurrency invocation delay $\overline{d}$. We observed that $\overline{ct}$ follows $\widetilde{ct}$ with an average inaccuracy of $\delta_{ct} = 5.9\,\%$ for all functions, ranging between 4.3 % to 7.7 %. *SimLess* generated overall makespan inaccuracy of $\delta_M = 7.55\,\%$. Applying $\overline{d}$ on the other five regions achieves a similar inaccuracy of up to $\delta_{ct} = 10.85\,\%$ for *AC*.

$BWA_{2,500}$ FC. We further apply $\overline{d} = 2.23$ ms determined using the $MC_{1,000}$ executions to the $BWA_{2,500}$ FC. We used the *AF*, *AV*, and *AT* AWS regions to evaluate the $\overline{ct}$ with the same AWS S3 setup as for BWA_{20}, due to the high memory assignment and long duration of the $BWA_{2,500}$ FC. *SimLess* also used the measured $\widetilde{RTT}$ in *AF* to determine $\overline{RTT}_T$. Fig. 12 presents the simulated $\overline{ct}$ and measured completion times $\widetilde{ct}$, demonstrating a low average inaccuracy of $\delta_{ct} = 10.1\,\%$

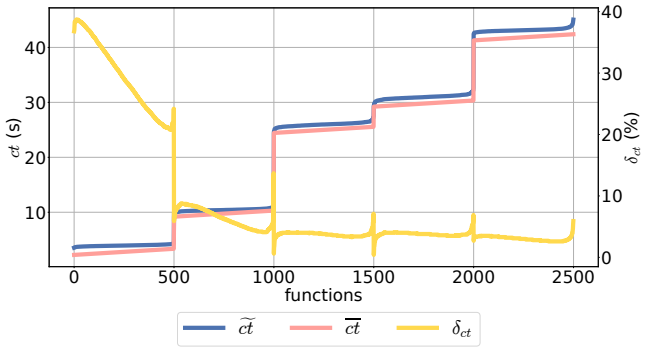


Figure 12: Simulated versus measured completion time ct of $BWA_{2,500}$ functions on AF and their inaccuracy.

on average. Similar to BWA_{20}, the average inaccuracy of $\delta_{ct} = 27.6\,\%$ is significantly higher for the `Split` function, and as low as $\delta_{ct} = 5.7\,\%$ for the others.

7 DISCUSSION

This section compares *SimLess* with related research, presents its advances beyond the state-of-the-art, and discusses cost savings for parameter setup, other features, and limitations.

7.1 Related work

7.1.1 FC management systems. Hyperflow [39] is among the first systems that run FCs on specific regions of various providers. MPSC [3] can migrate an FC execution between AWS and IBM. xAFCL [54] runs FC functions across multiple providers with incorporated resilience [52]. However, none of these systems offers a simulation model that learns the *RTT* of FCs across various providers to optimize executions.

7.1.2 Serverfull simulation frameworks. CloudSim [9] is a popular simulation framework of scheduling and resource provisioning algorithms in elastic Clouds, focused on virtual machine parameters with workflow and serverless platform support. Piraghaj *et al.* [47] extended CloudSim to simulate resource management and container scheduling rather than applications. WorkflowSim [14], GroudSim [44], and DynamicCloudSim [7] introduce several workflow-specific parameters without FaaS platform support.

7.1.3 FaaS simulation frameworks. SimFaas [65] is an emulator that estimates the functions' runtime, cold start effects, and economic costs in FaaS platforms. DFaaSCloud [29] and OpenDC Serverless [31] simulation frameworks for FaaS platforms focus on functions' *ET* without considering the entire *RTT* in FCs. SimFaaS [37] simulates individual functions with fixed memory allocation in a single region and models characteristics such as average response time, cold start, and concurrent instances.

Novelty. SimLess advances the state-of-the-art through a novel method that reuses simulation data among twins and siblings to reduce the cost of parameter setup without significant accuracy tradeoffs. To our knowledge, *SimLess* is the first simulation framework of complete FC across multiple regions of various providers in federated FaaS.

7.1.4 ML-based simulation. The work in [36] introduced an ML model that predicts the functions' *ET* without considering the overhead and the *RTT*. FaaStest [28] uses self-optimizing machine learning to predict a function workload. FaaSSimulator [48] simulation tool supports scheduling across FaaS and virtual machines. Sizeless [21] predicts the function *ET* by monitoring its siblings using AWS Lambda and Node.js programming language without considering twins' *RTT* in other regions.

Machine learning limitations. While machine learning may predict function behavior, it has several deficiencies. Firstly, it requires execution time series for each function, which is a tedious, time-consuming, and costly operation in federated FaaS. In contrast, *SimLess* can accurately simulate FCs without even running twins and siblings. Secondly, evolutionary learning methods are computationally-intensive in the large search space of federated FaaS. Finally, predicting *ET* of short-running tasks, such as serverless functions, may generate higher inaccuracy, as reported by [46].

7.1.5 Comparison with other simulators. The main innovation of the *SimLess* FC simulation model is the overhead $\overline{O}$. Since existing simulators do not consider overheads but focus on the *ET* in a single region, we consider a direct comparison with *SimLess* as unfair. Nevertheless, other researchers achieved similar results to the *SimLess* parameter setup. For instance, [13] reported 100 ms in AWS to authenticate and schedule functions, corresponding to the aggregated cryptography and FaaS overheads in *SimLess*: $\overline{cr} + \overline{FO} = 107$ ms.

7.2 *SimLess* limitations

We discuss several limitations that may hinder the *SimLess* framework from accurately simulating FC in federated FaaS.

Implementation. SimLess implementation restricts the FC composition to AFCL. However, the *SimLess* simulation model is generic and applicable to any FC management system.

FaaS provider dependency. SimLess FC simulation model relies on overheads (i.e., network, FaaS, concurrency) that affect the function *RTT*, measured for three providers (i.e., AWS. IBM, Google). Whenever these providers change parts of their infrastructure, it may be necessary to repeat the parameter setup procedure. Nevertheless, *SimLess* allows

easy setup of these parameters in the metadata by running only 120 no-op functions per provider in a few seconds.

Constant execution time of twins ET_T across regions. The assumption that $ET_T = ET$ may not be valid for all functions and regions. Still, our evaluation showed that this assumption is valid even for AWS twins that use storage in their region. For example, the BWA twins on AWS and IBM have similar execution times despite accessing storage (see Fig. 10).

Cold start ignorance. SimLess is unaware if a function execution is a warm or a cold start and, therefore, does not consider cold start effects. To address this limitation, one can schedule function invocations to keep alive an appropriate number of containers and to warm up functions, as proposed by the predictive method in [58].

Fixed input data. We evaluated *SimLess* using functions invoked with fixed input data delivering constant execution time. For example, changing the fixed number of points (i.e., from 1,000,000) in the MC FC would produce a considerably different *RTT* that may increase the inaccuracy. Similar examples are synchronization tasks such as `averagePi`, whose *RTT* depends on the size of the `monteCarlo` parallel loop. One may create different deployments in the metadata to avoid this inaccuracy for various parallel loop sizes.

Same region BaaS restriction. SimLess reuses parameters between function twins with BaaS use within the same region (or no use). Functions using storage of another region, for example, may have siblings but no twins since the data transfer and the execution times differ. Still, moving computing closer to data, rather opposite, is better due to: (1) network proximity with higher internal bandwidth (e.g., up to 100 Gbit/s in AWS regions), and (2) zero charges for outgoing traffic when downloading or uploading data. One should consider the difference in overheads when invoking twins.

Performance instability. As for any simulation, performance instability may increase *SimLess* simulation inaccuracy. Over one month, an investigation of the AWS, Google, IBM, and Microsoft Azure functions revealed several sinusoidal patterns for *ET* and disk I/O throughput within one day [33]. Moreover, some providers use up to seven different CPU models, affecting the simulation inaccuracy despite the minimal CPU utilization variance. For short running functions with $ET \longrightarrow 0$, however, *RTT* mainly depends on the overhead O, evaluated in Section 6.2.2.

Centralized data collection in federated FaaS. We configured *SimLess* based on data gathered from the centralized *xAFCL* workflow management system [54], causing overheads to each region. An alternative decentralized approach in federated FaaS that submits parts of the FC across the regions to minimize overheads requires a tradeoff between *RTT*, concurrency, and decentralization costs, including data transfers. We see two development challenges for decentralization in federated FaaS: (1) complex FC transformations for running embarrassingly parallel loops, and (2) dynamic storage selection considering network proximity. Compliant with the *SimLess* definition of twins, Apache LibCloud allows developers to dynamically select the BaaS driver (i.e., AWS S3, Google Cloud Storage) but still needs rewriting and redeploying the function.

7.3 Parameter setup cost

Apart from the low inaccuracy, *SimLess* reduces the cost of parameter setup measured in the number of function learning invocations compared to the related works.

Number of function invocations. To initiate the parameter setup, *SimLess* executes nine no-op FCs of four functions each, repeated five times with and without authentication, totalling 360 functions (i.e., $9 \cdot 4 \cdot 5 \cdot 2 = 360$). Additionally, it executes the $MC_{1,000}$ FC in three AWS regions to determine $\overline{d}$, leading to a total of 15,360 functions. These runs complete the parameter setup of the three providers (i.e., AWS, IBM, Google) and the concurrency setup for AWS. On top, *SimLess* requires five repetitions of each FC in one region of each provider leading to $5 \cdot 3 \cdot 10 = 150$ MC_{10} functions in three regions and $5 \cdot 2 \cdot 20 = 200$ BWA_{20} functions in two regions.

SimLess parameter setup savings. Conventional methods require to run all functions for all provider regions, thereby needed 69,000 effective functions executions for four scientific FCs (MC_{10}, BWA_{20}, $MC_{1,000}$, and $BWA_{2,500}$) to learn their behaviour. Based on this analysis, *SimLess* FC simulation model reduces the number of function invocations by 77.23%. Moreover, *SimLess* is already able to accurately simulate MC and BWA with an arbitrary degree of parallelism within AWS. We conclude that *SimLess* needs negligible cost and effort to set up parameters for the simulation.

7.4 Additional features

We built several additional features in the *SimLess* framework besides those evaluated in this paper.

Cost simulation. SimLess can simulate costs and cost models of all providers configured in the metadata. However, we omitted this evaluation due to space limitations and its relatively simple computation based on $\overline{ET}$.

Failure simulation. SimLess can simulate failures based on historical data and specify alternative functions using AFCL.

Sibling RTT simulation. Although *SimLess* supports a scaling factor configuration $s(m)$, we did not simulate $\overline{RTT}_S$ because this factor is function-specific. Preliminary measurements showed an appropriate linear speedup model

$s(m) = m$ for the compute-bound MC FC. In general, learning the scaling factor $s(m)$ may be a costly operation that reuses data between siblings but still requires experiments in one provider region to simulate the corresponding twins in other regions. For example, learning the scaling factor for functions with 128 MB, 256 MB, and 512 MB allows *SimLess* to simulate their twins across all regions of a provider. Reusing data for twins is especially important for functions that need the highest allocated memory, such as BWA.

8 CONCLUSIONS AND FUTURE WORK

We presented *SimLess*, a novel semi-self-configurable simulation framework of FCs in federated FaaS based on two original light concepts: (1) function twins deployed with the same computing, communication, and storage resources in other regions of the same FaaS provider, and (2) siblings, representing the same function deployed in the same region with different computing resources. *SimLess* goes beyond existing cloud simulators that require manual resource speed and task complexity configurations to determine execution time, which partially describes their behavior in a single region, but not in a federated FaaS subject to other overheads.

Validation. Extensive validation results demonstrate that the self-configurable parameters defined by the *SimLess* simulation model are provider-dependent and FC agnostic. Running a simple no-op FC on three regions is sufficient to determine all parameters within a provider region with a slight inaccuracy of 7.9 %. Applying them to other regions of the same provider leads to a slightly higher inaccuracy of 9.8 % for twins, making the parameter setup costs negligible.

Novelty. SimLess is the first FC simulation framework that models important overheads, including provider-specific (i.e., session $\overline{SO}$, FaaS $\overline{FO}$, concurrency invocation $\overline{d}$, cryptography $\overline{cr}$) and region-specific (i.e., network $\overline{NO}$, authentication $\overline{AO}$). Such a parameter setup allows running a function in one region and simulating its twins in the other regions of the same provider. *SimLess* is FC-agnostic and reuses the $\overline{SO}$ and $\overline{d}$ parameters directly affected by the FC. We used a no-op FC to learn $\overline{SO}$ and applied it to the $MC_{1,000}$ FC to learn $\overline{d}$, further used for the $BWA_{2,500}$ twins. However, the benefits of the *SimLess* FC simulation model are mainly for twins because of the function-dependent scaling speedup $s(m)$. Nevertheless, *SimLess* allows similar linear speedup approximations to reduce the parameter setup for siblings.

Evaluation. Sensitivity analysis revealed that running FCs with a low concurrency of 20 functions in a single region is sufficient to simulate a highly concurrent FC with 2,500 functions with 77.23 % lower costs. *SimLess* simulates the $\overline{RTT}_T$ of all twins from the measured $\widetilde{RTT}$ in one region, without any execution time $\widetilde{ET}$ measurements. Unfortunately, scalability issues within IBM and Google constrained the evaluation of highly-concurrent FCs to AWS twins. For instance, the MC_{10} FC runs fastest on IBM, while $MC_{1,000}$ is slowest on IBM and requires a ten times longer execution time (i.e., 50 s).

Future work. We plan to extend *SimLess* in three directions.

(1) *FC scheduling with dynamic storage:* While this paper used a static storage setup, we intend to extend the *SimLess* FC simulation model to specify storage or buckets as function input data dynamically. We will research a data transfer model across regions that split the execution into download, computation, and upload times. Then, we will update the twins' definition to share the computation time rather than the entire execution time. This approach allows FC schedulers to determine not only the function deployment $f\mathcal{D}$ to invoke but also the dynamic input buckets to download and the storage location of output data.

(2) *Deployment time prediction.* While the *SimLess* FC simulation model estimates the RTT of all twins across the globe, it is unaware of their deployment. In a real scenario, for example, a user may prefer to run the already deployed twin in Tokyo with an overhead of 1.2 s instead of deploying it in AWS Frankfurt and running it with the total overhead of at least $1.8 + 0.226 \approx 2$ s [54]. To bridge this gap, we will further extend the *SimLess* FC simulation model to estimate the deployment time of each function implementation, which depends on its deployment package size, network overhead, and authentication overhead [54].

(3) *Multi-objective FC scheduling.* Finally, we will research a multi-objective scheduler based on the previous two future works for evaluating the tradeoff of deploying new functions in closer regions, reusing already deployed functions, and caching intermediary data.

ACKNOWLEDGMENTS

This research received funding from:

- *Land Tirol*, under contract F.35499;
- *European High-Performance Computing Joint Undertaking*, under grant agreement 951745 (FF4EuroHPC project and CardioHPC experiment);
- *European Union*, under grant agreements 101016835 (Horizon 2020 DataCloud project) and 101093202 (Horizon Europe Graph-Massivizer project).

REFERENCES

[1] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67 (Spring))*. ACM, Atlantic City, New Jersey, 483–485. https://doi.org/10.1145/1465482.1465560

[2] Aitor Arjona, Pedro García López, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2021. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems* 124 (2021), 215–229. https://doi.org/10.1016/j.future.2021.06.004

[3] Austin Aske and Xinghui Zhao. 2018. Supporting Multi-Provider Serverless Computing on the Edge. In *International Conference on Parallel Processing Companion (ICPP '18)*. ACM, OR, USA.

[4] Daniel Barcelona-Pons and Pedro García-López. 2021. Benchmarking parallelism in FaaS platforms. *Future Generation Computer Systems* (2021).

[5] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures *(Middleware '19)*. ACM, Davis, CA, USA, 41–54.

[6] Stefano Buliani. 2017. Building a Multi-region Serverless Application with Amazon API Gateway and AWS Lambda. https://aws.amazon.com/blogs/compute/building-a-multi-region-serverless-application-with-amazon-api-gateway-and-aws-lambda/ Accessed: 2022-09-10.

[7] Marc Bux and Ulf Leser. 2015. DynamicCloudSim: Simulating heterogeneity in computational clouds. *Fut. Gen. Comp. Syst.* 46 (2015), 85–99. https://doi.org/10.1016/j.future.2014.09.007

[8] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *European Conference on Computer Systems (EuroSys '20)*. ACM, Heraklion, Greece, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[9] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience* 41, 1 (2011), 23–50.

[10] L. Carvalho and Aletéia Patrícia Favacho de Araújo. 2020. Remote Procedure Call Approach using the Node2FaaS Framework with Terraform for Function as a Service. In *International Conference on Cloud Computing and Services Science - CLOSER,*. SciTePress.

[11] Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, and Frédéric Suter. 2020. Developing accurate and scalable simulators of production workflow management systems with WRENCH. *Fut. Gen. Comp. Syst.* 112 (2020), 162 – 175.

[12] Mohak Chadha, Anshul Jindal, and Michael Gerndt. 2021. Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions. *CoRR* abs/2107.10008 (2021). arXiv:2107.10008 https://arxiv.org/abs/2107.10008 Accepted in IEEE Cloud 2021.

[13] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20)*. ACM, Stockholm, Sweden, 65–76. https://doi.org/10.1145/3369583.3392683

[14] W. Chen and E. Deelman. 2012. WorkflowSim: A toolkit for simulating scientific workflows in distributed environments. In *International Conference on E-Science*. 1–8.

[15] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2021. RFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing. *CoRR* (2021).

[16] Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, and Wes Lloyd. 2020. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software. In *International Workshop on Serverless Computing (WoSC'20)*. ACM, Delft, Netherlands, 67–72. https://doi.org/10.1145/3429880.3430103

[17] dashbird. 2019. State of Lambda functions in 2019 by Dashbird. https://dashbird.io/blog/state-of-lambda-functions-2019/. Accessed: 2022-06-15.

[18] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[19] Javier Diaz-Montes, Manuel Diaz-Granados, Mengsong Zou, Shu Tao, and Manish Parashar. 2018. Supporting Data-Intensive Workflows in Software-Defined Federated Multi-Clouds. *IEEE Transactions on Cloud Computing* 6, 1 (2018), 250–263. https://doi.org/10.1109/TCC.2015.2481410

[20] Juan J. Durillo, Radu Prodan, and Jorge G. Barbosa. 2015. Pareto tradeoff scheduling of workflows on federated commercial Clouds. *Simulation Modelling Practice and Theory* 58 (2015), 95–111. https://doi.org/10.1016/j.simpat.2015.07.001 Special Issue on TECHNIQUES AND APPLICATIONS FOR SUSTAINABLE ULTRASCALE COMPUTING SYSTEMS.

[21] Simon Eismann, Long Bui, Johannes Grohmann, Cristina L Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. *arXiv preprint arXiv:2010.15162* (2021).

[22] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the Costs of Serverless Workflows. In *Int. Conf. on Performance Engineering (ICPE)*. ACM, Canada, 265–276.

[23] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. 2020. Serverless End Game: Disaggregation enabling Transparency. *arXiv preprint arXiv:2006.01251* (2020).

[24] Google. 2022. GCPing. https://gcping.com/ Accessed: 2022-06-15.

[25] Nicholas Hazekamp, Nathaniel Kremer-Herman, Benjamin Tovar, Haiyan Meng, Olivia Choudhury, Scott Emrich, and Douglas Thain. 2018. Combining Static and Dynamic Storage Management for Data Intensive Scientific Workflows. *IEEE Trans. on Par. and Distr. Sys.* 29, 2 (2018), 338–350. https://doi.org/10.1109/TPDS.2017.2764897

[26] Michael T Heath. 2018. *Scientific Computing: An Introductory Survey, Revised Second Edition*. SIAM.

[27] Sanghyun Hong, Abhinav Srivastava, William Shambrook, and Tudor Dumitras. 2018. Go Serverless: Securing Cloud via Serverless Design Patterns. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, Boston, MA.

[28] Shay Horovitz, Roei Amos, Ohad Baruch, Tomer Cohen, Tal Oyar, and Afik Deri. 2018. FaaStest-Machine Learning Based Cost and Performance FaaS Optimization. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 171–186.

[29] Hongseok Jeon, Chunglae Cho, Seungjae Shin, and Seunghyun Yoon. 2019. A CloudSim-Extension for Simulating Distributed Functions-as-a-Service. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. 386–391. https://doi.org/10.1109/PDCAT46702.2019.00076

[30] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Symposium on Cloud Computing*. 445–451.

[31] S Jounaid. 2020. OpenDC Serverless: Design, Implementation and Evaluation of a FaaS Platform Simulator. Ph.D. thesis, Vrije Universiteit Amsterdam.

[32] Alan H. Karp and Horace P. Flatt. 1990. Measuring Parallel Processor Performance. *Commun. ACM* 33, 5 (May 1990), 539–543. https://doi.org/10.1145/78607.78614

[33] Daniel Kelly, Frank Glavin, and Enda Barrett. 2020. Serverless Computing: Behind the Scenes of Major Platforms. In *IEEE International Conference on Cloud Computing (CLOUD)*. 304–312. https://doi.org/10.1109/CLOUD49709.2020.00050

[34] Michael Leonhard. 2022. cloudping.info . https://www.cloudping.info/ Accessed: 2022-06-15.

[35] Heng Li and Richard Durbin. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26, 5 (2010), 589–595.

[36] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2021), 615–632. https://doi.org/10.1109/TPDS.2020.3028841

[37] Nima Mahmoudi and Hamzeh Khazaei. 2021. SimFaaS: A Performance Simulator for Serverless Computing Platforms. In *Int. Conf. on Cloud Computing and Services Science (CLOSER '21)*. 1–11.

[38] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: A Benchmark Suite for Serverless Computing. In *ACM International Conference on Distributed and Event-Based Systems (DEBS '20)*. ACM, Montreal, Canada, 73–84. https://doi.org/10.1145/3401025.3401738

[39] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* 110 (2020), 502–514. https://doi.org/10.1016/j.future.2017.10.029

[40] Johannes Manner and Guido Wirtz. 2019. Impact of Application Load in Function as a Service. (2019).

[41] Horácio Martins, Filipe Araujo, and Paulo Rupino da Cunha. 2020. Benchmarking serverless computing platforms. *Journal of Grid Computing* 18, 4 (2020), 691–709.

[42] Microsoft Azure. 2022. Multicloud solutions with the Serverless Framework. https://docs.microsoft.com/en-us/azure/architecture/example-scenario/serverless/serverless-multicloud Accessed: 2022-09-10.

[43] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. 2017. A Serverless Real-Time Data Analytics Platform for Edge Computing. *IEEE Internet Computing* 21, 4 (2017), 64–71. https://doi.org/10.1109/MIC.2017.2911430

[44] Simon Ostermann, Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. 2010. GroudSim: An Event-Based Simulation Framework for Computational Grids and Clouds.. In *Euro-Par Workshops (Lecture Notes in Computer Science, Vol. 6586)*. Springer, 305–313.

[45] Stefan Pedratscher, Sasko Ristov, and Thomas Fahringer. 2022. M2FaaS: Transparent and fault tolerant FaaSification of Node.js monolith code blocks. *Future Generation Computer Systems* 135 (2022), 57–71. https://doi.org/10.1016/j.future.2022.04.021

[46] Thanh-Phuong Pham, Juan Durillo, and Thomas Fahringer. 2020. Predicting Workflow Task Execution Time in the Cloud Using A Two-Stage Machine Learning Approach. *IEEE Transactions on Cloud Computing* 8, 1 (2020), 256–268. https://doi.org/10.1109/TCC.2017.2732344

[47] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. 2017. ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers. *Software: Practice and Experience* 47, 4 (2017), 505–521.

[48] Anja Reuter, Timon Back, and Vasilios Andrikopoulos. 2020. Cost efficiency under mixed serverless and serverful deployments. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 242–245. https://doi.org/10.1109/SEAA51224.2020.00049

[49] Sebastián Risco, Germán Moltó, Diana M Naranjo, and Ignacio Blanquer. 2021. Serverless workflows for containerised applications in the cloud continuum. *Journal of Grid Computing* 19, 3 (2021), 1–18.

[50] Sashko Ristov and Philipp Gritsch. 2022. FaaSt: Optimize makespan of serverless workflows in federated commercial FaaS. In *2022 IEEE International Conference on Cluster Computing (CLUSTER '22)*. IEEE, Heidelberg, Germany, 182–194. https://doi.org/10.1109/CLUSTER51413.2022.00032

[51] Sashko Ristov, Christian Hollaus, and Mika Hautz. 2022. Colder Than the Warm Start and Warmer Than the Cold Start! Experience the Spawn Start in FaaS Providers. In *Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED '22)*. ACM, Salerno, Italy, 35–39. https://doi.org/10.1145/3524053.3542751

[52] Sasko Ristov, Dragi Kimovski, and Thomas Fahringer. 2022. FaaScinating Resilience for Serverless Function Choreographies in Federated Clouds. *IEEE Transactions on Network and Service Management* (2022), 1–1. https://doi.org/10.1109/TNSM.2022.3162036

[53] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2021. AFCL: An Abstract Function Choreography Language for serverless workflow specification. *Fut. Gen. Comp. Syst.* 114 (2021), 368 – 382.

[54] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2021. xAFCL: Run Scalable Function Choreographies Across Multiple FaaS Systems. *IEEE Transactions on Services Computing* (2021), 1–1. https://doi.org/10.1109/TSC.2021.3128137

[55] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. 2016. Superlinear Speedup in HPC Systems: why and when?. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*. Gdansk, Poland, 889–898.

[56] Roland Mathá, Sasko Ristov, Thomas Fahringer, and Radu Prodan. 2020. Simplified Workflow Simulation on Clouds based on Computation and Communication Noisiness. *IEEE Transactions on Parallel and Distributed Systems* 31, 7 (2020), 1559–1574. https://doi.org/10.1109/TPDS.2020.2967662

[57] Francisco Romero, Gohar Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. *CoRR* (2021).

[58] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, Lausanne, Switzerland, 753–767. https://doi.org/10.1145/3503222.3507750

[59] S. Ristov, R. Mathá and R. Prodan. 2017. Analysing the Performance Instability Correlation with Various Workflow and Cloud Parameters. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 446–453. https://doi.org/10.1109/PDP.2017.80

[60] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2021. Toward Multicloud Access Transparency in Serverless Computing. *IEEE Soft.* 38, 1 (2021), 68–74. https://doi.org/10.1109/MS.2020.3029994

[61] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez. 2021. Outsourcing Data Processing Jobs with Lithops. *IEEE Transactions on Cloud Computing* (Nov. 2021), 1–1. https://doi.org/10.1109/TCC.2021.3129000

[62] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud *(Middleware*

'18). ACM, Rennes, France, 1–8.

[63] Sasko Ristov, Stefan Pedratscher, Jakob Wallnoefer, and Thomas Fahringer. 2021. DAF: Dependency-Aware FaaSifier for Node.js Monolithic Applications. *IEEE Software* 38, 1 (2021), 48–53. https://doi.org/10.1109/MS.2020.3018334

[64] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX ATC 20*. 205–218.

[65] Erwin van Eyk. 2021. SimFaaS. https://github.com/erwinvaneyk/simfaas Accessed: 2021-10-22.

[66] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *USENIX Annual Technical Conference*. Boston, MA, USA, 133–145.

[67] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2021. Restructuring Serverless Computing with Data-Centric Function Orchestration. *arXiv preprint arXiv:2109.13492* (2021).

[68] Qimin Zhang, Nathaniel Kremer-Herman, Benjamin Tovar, and Douglas Thain. 2018. Reduction of Workflow Resource Consumption Using a Density-based Clustering Model. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 1–9. https://doi.org/10.1109/WORKS.2018.00006