Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

faas-sim: A Trace-Driven Simulation Framework for Serverless Edge Computing Platforms

Philipp Raith* | Thomas Rausch | Alireza Furutanpey | Schahram Dustdar

[1]Distributed Systems Group, TU Wien
Wien, Austria

Correspondence
*Philipp Raith, Email:
p.raith@dsg.tuwien.ac.at

Abstract

This paper presents *faas-sim*, a simulation framework tailored to serverless edge computing platforms. In serverless computing, platform operators are tasked with efficiently managing distributed computing infrastructure completely abstracted from application developers. To that end, platform operators and researchers need tools to design, build, and evaluate resource management techniques that efficiently use of infrastructure while optimizing application performance. This challenge is exacerbated in edge computing scenarios, where, compared to cloud computing, there is a lack of reference architectures, design tools, or standardized benchmarks. *faas-sim* bridges this gap by providing (a) a generalized model of serverless systems that builds on the function-as-a-service abstraction, (b) a simulator that uses trace data from real-world edge computing testbeds and representative workloads, and (c) a network topology generator to model and simulate distributed and heterogeneous edge-cloud systems. We present the conceptual design, implementation, and a thorough evaluation of *faas-sim*. By running experiments on both real-world test beds and replicating them using *faas-sim*, we show that the simulator provides accurate results and reasonable simulation performance. We have profiled a wide range of edge computing infrastructure and workloads, focusing on typical edge computing scenarios such as edge AI inference or data processing. Moreover, we present several instances where we have successfully used *faas-sim* to either design, optimize, or evaluate serverless edge computing systems.

KEYWORDS:
Simulation; Co-Simulation; Serverless Edge Computing; Edge-Cloud Continuum

1 | INTRODUCTION

Context & Background

Serverless edge computing is an emerging distributed computing model that applies principles from serverless cloud computing to edge computing systems[1]. In traditional serverless computing, serverless platforms hide the underlying infrastructure of massive cloud data centers from application developers while giving platform operators the controls to schedule and scale workloads to optimize operational goals[2]. The edge-cloud continuum extends the narrow cloud- or edge-centric view to a hierarchical geo-distributed system, ranging from resource-constrained edge devices to large-scale cloud data centers[3]. Applying the operational mechanisms that enable serverless computing in this new environment is highly challenging and subject to

active research[4,1,3]. Current cloud-centric orchestration services need to be adapted in their architecture[5], which is typically centralized and can form bottlenecks. While concepts like federated clouds can help connect multiple on-premises data centers and mitigate scalability issues[6], how current serverless platforms behave in these heterogeneous and geo-distributed scenarios is not well understood. New challenges introduced by the edge-cloud continuum include discovering and forming computing clusters[7], incorporating new computing device types while aiming for resource efficiency[8], or dealing with data and computation movement trade-offs[9].

Motivation

With the growing adoption of serverless edge computing and the complexity inherent to these systems, there is a pressing need for effective evaluation frameworks to assess the performance and help aid the design of serverless edge computing platforms. In cloud computing research, there is a long history of well-understood benchmarks[10], testbeds[11], and trace data sets[12] to perform systems evaluations. Evaluating edge computing systems is much more difficult due to the lack of established benchmarks, reference architectures, trace data sets, or real-world testbeds[13]. Researchers must invest considerable resources to create testbeds and benchmarks often tailored to the system they are evaluating[14]. This then often leads to evaluations that do not allow generalizable conclusions about the performance of the systems under the wide variety of conditions that the edge-cloud continuum provides. Simulators help to fill this gap by allowing quick iterations over ideas and evaluating their feasibility before deploying them on real-world hardware[15]. They allow the creation of synthetic infrastructure according to specific parameters, such as deployment density in geo-distributed settings or heterogeneity of compute nodes[13], thereby vastly expanding the evaluation parameter space. Moreover, simulations can facilitate use cases such as tuning serverless function adaptation strategies[16], resource planning, cost prediction, and application performance estimation. As we explore in this paper, simulators can even be used in co-simulation, where systems optimize themselves during runtime by evaluating different scenarios using the simulator.

Challenges

In recent years, numerous simulators have emerged for different system environments, such as cloud computing[17] IoT[18,19] serverless computing[20,15,21], and edge computing[22,23]. However, existing work does not thoroughly investigate the requirements, use cases, and solution approaches to address the challenges described in the previous subsection. Moreover, many state-of-the-art systems simulators, such as CloudSim[17] and its many derivatives[23,22], build on oversimplified resource models that have several limitations that we analyze in depth in this paper. From studying the current state-of-the-art and analyzing simulation requirements, we summarize the challenges for building a modern simulator for serverless edge computing platforms as follows:

(i) To seamlessly transfer simulation designs to real-world systems, the domain model should generalize edge-cloud simulators to include serverless computational models. (ii) The simulator should have strong support to evaluate orchestration strategies as they are essential for governing the overall behavior of function deployments. Specifically, serverless function adaptations (scaling, placement, and routing) should be first-class entities for users to modify and extend. (iii) Since serverless edge computing is still evolving, the simulation engine should strongly emphasize modularity and extensibility. (iv) High fidelity simulators are typically slow and challenging to implement but closely resemble real-world conditions. Conversely, low fidelity simulators are fast and straightforward to implement. Hence, the simulator should be configurable to accommodate the current needs of client programmers and platform designers and create low or high fidelity simulations. (v) The spatio-temporal context in simulations is highly application specific, i.e., it requires careful modeling of component interference and user access patterns. Consequently, client programmers and platform designers should be able to integrate custom resource models. (vi) The simulation should support programmable topologies that scale for large clusters to model the heterogeneous resource aggregates.

Contributions

We have built *faas-sim* to address these challenges and overcome the limitations of state-of-the-art simulators for serverless edge computing platforms. *faas-sim* is a trace-driven stochastic discrete-event simulation engine, enabling the fine-grained evaluation of serverless edge computing platforms on edge-cloud infrastructures. It simulates the deployment, execution, and resource consumption of serverless workloads on different types of computing hardware and comes with a flow-based network simulator based on the network topology synthesizer *Ether*[13]. Our code framework provides a generalized serverless system model that allows modeling a wide range of function adaptation mechanisms and platform optimization algorithms, as well as developing reusable benchmarks and experiments. The simulation engine has a flexible performance and resource modeling approach to evaluate common system performance indicators such as resource consumption, function execution time, data throughput, or network usage. *faas-sim* is trace-driven and relies on data from real profiling experiments. It ships with a wide range of profiled workloads and compute nodes that can be used out of the box to bootstrap experiments, but users can provide custom profiling

traces to model other workloads and hardware. *faas-sim* is integrated into the Edgerun project [1], offering extensive tooling to support researchers and practitioners performing edge-cloud experiments. It is written in Python, built using the discrete event simulation engine SimPy[24], and integrates seamlessly with modern data science tools. The objective of *faas-sim* and the complementary Edgerun ecosystem is to aid researchers and platform designers in developing and evaluating new serverless edge platforms. For example, the experimentation framework *Galileo*[14] can run experiments on testbeds and pre-process traces that *faas-sim* expects.

We summarize our contributions as follows:

- *faas-sim*, an open-source[2] serverless edge computing simulator based on the design and performance data of real-world platforms.
- *faas-sim* enables researchers and practitioners to design, implement, model, and evaluate serverless platform architectures and operational strategies such as function adaptations, scheduling, or load-balancing algorithms.
- An efficient performance and resource modeling approach that uses profiling data from real-world edge-cloud workloads and compute infrastructure.
- A set of profiling benchmark traces included in *faas-sim* to bootstrap simulations.
- An evaluation on a real-world testbed demonstrating the accuracy of our network simulator.
- A use case-based evaluation showing the capabilities of *faas-sim*, including resource planning and co-simulation-driven function adaptations.

Outline

The remainder of the paper is structured as follows. Section 2 introduces the simulation-driven design process of serverless edge computing. Specifically, we introduce two main tasks that serverless edge platforms must implement. Based on use cases, we show how our simulation can facilitate implementation and highlight the resulting simulation challenges. Afterward, Section 3 describes the conceptual model, architecture of *faas-sim*, and various models for resources, performance, and network. Section 4 represents the evaluation of our work. It is structured into multiple parts and evaluates *faas-sim*'s ability to model the simulation use cases we present in Section 2, shows traces that come with *faas-sim*, evaluates the network simulation and showcases its flexible domain model by implementing the reverse proxy component of OpenFaaS[25]. Afterward, Section 5 highlights the main similarities and differences between ours and other simulation engines. Section 6 concludes our work and outlines future tasks.

2 | SIMULATION-DRIVEN SERVERLESS EDGE COMPUTING DESIGN

In the following, we describe fundamental aspects of our work revolving around serverless edge computing and simulations. We introduce in Section 2.1 serverless edge computing in general and the tasks of designing platforms and function adaptation strategies which *faas-sim* intends to support. Therefore, we show in Section 2.2, use cases that facilitate the design process of platforms and address the simulation challenges that arise.

2.1 | Serverless edge computing

Serverless edge computing extends serverless computing to manage resources and applications across the edge-cloud continuum[1]. Serverless computing combines Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS)[2]. FaaS enables developers to split complex applications into multiple simple functions, package and upload them. The FaaS provider is responsible for placing and scaling function replicas and routing requests to running replicas. These are the main adaptation mechanisms of FaaS platforms to manage function deployments dynamically. Platform clients can invoke functions through various triggers. A client can send an HTTP request forwarded through a router (e.g., application-layer load balancer) to a function replica. Clients can publish messages in a queue, and brokers are responsible for forwarding them to function replicas. Other

[1] https://github.com/edgerun/
[2] https://github.com/edgerun/faas-sim

event-driven triggers include file and database changes.[2]. Function replicas are typically stateless and only have ephemeral storage. However, stateful function support is actively explored and commercially available (e.g., on Microsoft's Azure Functions platform[26]). Replicas cannot rely on the assumption that data is locally available and must fetch it from backend services. BaaS enables FaaS and manages various services such as databases, message middleware, routers, and caches. The autonomous adaptations of backend service components are crucial to succeed in the edge-cloud continuum and guarantee operational goals[27]. Adaptations are essential for serverless platforms and directly impact operational goals. Each of the adaptation mechanisms can have different implementations. For example, placement can have various implementations, such as optimization-based[9] or AI-driven[28]. Scaling can be reactive or proactive, while platforms can route centralized (e.g., a single gateway that all requests go through) or decentralized (e.g., where multiple gateways exist). These options are not exhaustive and should only highlight different ways of adapting function deployments. Interactions between the adaptation mechanisms are also complex. The system's performance after updating them can lead to unexpected behavior. Serverless edge computing represents a complex system facing many tasks and challenges. Figure 1 shows an example of a serverless edge computing platform. Note that requests are routed decentralized, but adaptations are made centrally in the cloud. This architecture might not be feasible due to the cloud's centralized scaling and placement components. However, a naive decentralization approach can lead to other problems, such as state sharing and consistency problems[29]. The figure also emphasizes that serverless platforms must implement adaptation mechanisms for FaaS and BaaS. The platform must place and scale function replicas and manage the backend services.

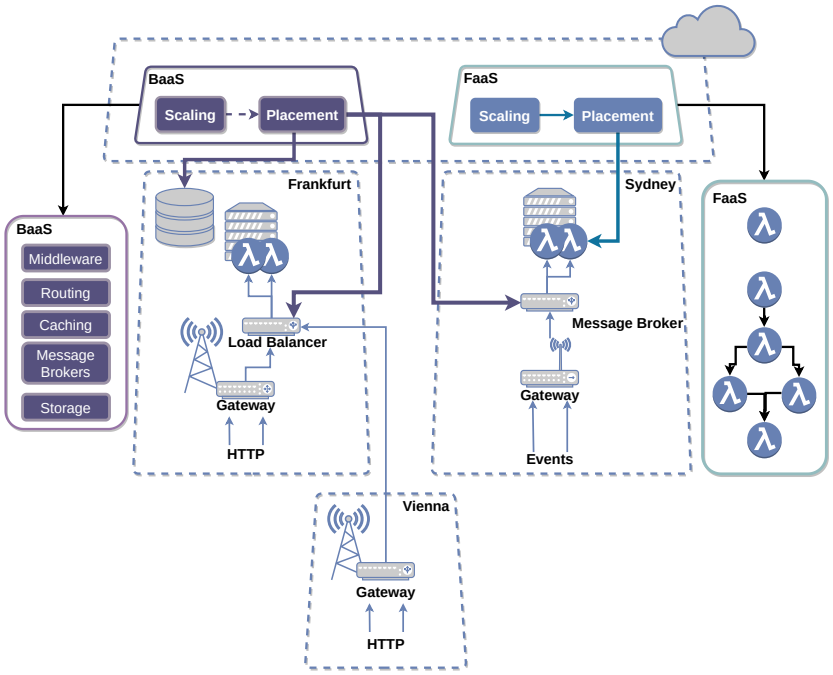
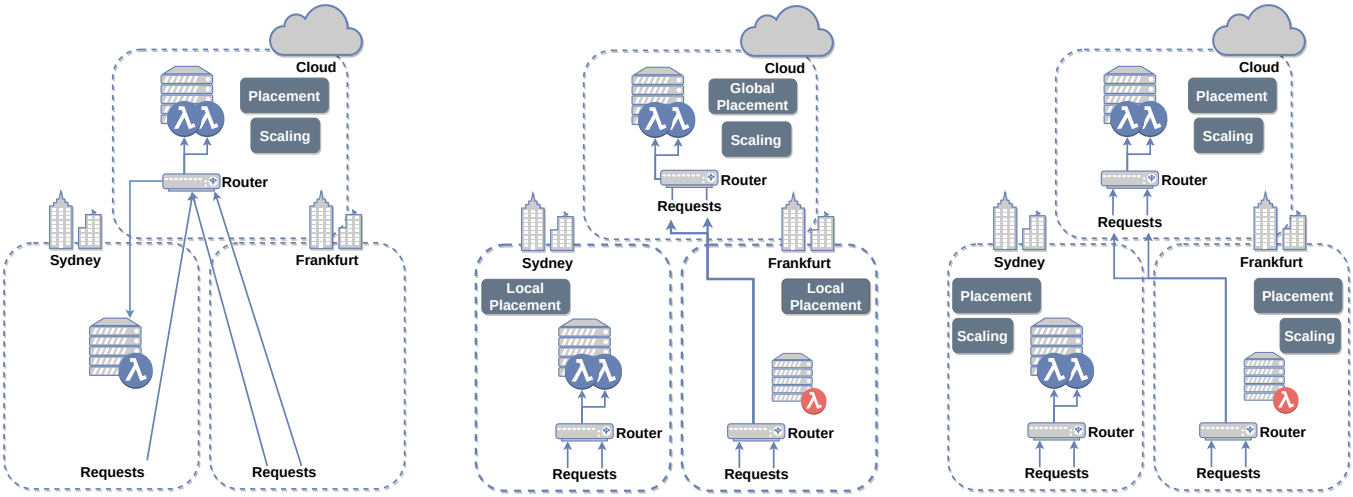


Figure 1 Example platform with decentralized gateways and centralized scaling and placement components

Operational goals can range from performance (e.g., response time) to privacy (e.g., execution location) and are crucial for emerging application paradigms (e.g., Edge Intelligence[30]). Edge Intelligence emphasizes and requires the execution of applications across the edge-cloud continuum. Applications may require ultra-low latency (e.g., < 10 milliseconds), have strict privacy constraints (e.g., health data), and are formulated as complex workflows consisting of multiple applications, each having different requirements (e.g., AI pipelines)[31]. Serverless adaptations are not only based on requirements but must consider environmental changes. A key characteristic of the edge-cloud continuum is its dynamic and heterogeneous environment. Therefore, platforms must be resilient, and their adaptation mechanisms must continuously evolve and cope with changes. These changes stem from the infrastructure side (e.g., nodes fail, new nodes join), from new function requirements (e.g., privacy), or unseen functions. This introduction to serverless edge computing should give a broad overview of the complex inter-dependencies between the platform architecture and serverless function adaptations. However, we want to give more details about these two aspects of platform design and how they can benefit from *faas-sim* and the simulation challenges they introduce.



(a) Centralized Platform Architecture **(b)** Hybrid Platform Architecture **(c)** Decentralized Platform Architecture

Figure 2 Three possible Platform Architectures for Serverless (Edge) Computing

2.1.1 | Platform architecture

In this work, the platform architecture determines the deployment (e.g., location of execution) and the responsibilities of the serverless function adaptation components, and which technologies the execution model consists of (e.g., function runtime). Thus, a simulation tool should be able to model these different aspects, from the placement of components to the underlying function runtimes. To highlight the difficulty in selecting an architecture, we briefly describe three platform architectures that are currently deployed or plausible for serverless edge computing platforms. This comparison should emphasize the need for tools that can evaluate different platforms without the burden of implementing each platform on a real-world system. Figure 2 depicts these architectures on an infrastructure that consists of three computing clusters exist (i.e., Cloud, Sydney, and Frankfurt). It is plausible that the number of computing clusters is much higher and consists of heterogeneous devices and network conditions. Thus, introducing the simulation challenge to be flexible regarding scenario definition. The first architecture is the typical cloud-centric Centralized Platform Architecture. It deploys a single entry point for users in the cloud, and placement and scaling components are also situated there. In this example, all components reside in the cloud, and requests must travel from the edge to the data center. This architecture has several shortcomings and represents how most commercial and open-source serverless platforms are built (e.g., OpenFaaS[25], Knative[32], Fission[33], OpenWhisk[34], AWS Lambda[35]). The main shortcoming is that requests must travel to the cloud to be handled, which increases network latency and diminishes the advantages of function instances at the edge. The Hybrid Platform Architecture shows an approach that deploys routing components at the edge and deploys a two-step placement approach[16]. The main differences to the Central Platform Architecture are decentralized router instances and the decentralized placement approach. The scaling component still resides in the cloud and decides when new function instances are necessary (or when to remove instances). The global placement component decides the computing cluster, and the local placement component must select a node. This architecture scales better because the global placement component can use simple heuristic algorithms while the local placement component can use complex algorithms. The drawback of this approach is that the cloud components can still be a bottleneck, which the design must incorporate. The Decentralized Platform Architecture deploys placement, scaling, and routing components in each computing cluster[36]. This architecture offers the advantage of decentralized components and high scalability. However, implementing this design can be challenging as it must coordinate placements and routing between computing clusters.

Other variants are possible; for example, the placement components can have distributed implementations that increase the complexity but offer high scalability[29]. To explore the performance impact of different platform architectures and underlying infrastructure, simulators are a good tool for getting first results and building confidence in a particular design. To this end, simulations must be customizable and extendable to adapt and model different platform designs quickly. Moreover, simulation users should be able to model and configure various system parameters easily, which introduces the challenge of simulation

configurability. Simulators should ship with a basic set of system assumptions so researchers can get started with baseline experiments quickly, but also allow high configurability through custom performance models (see Section 3.2.5), parameterizable infrastructure topologies (see Section 3.2.7), or adding completely custom simulation components.

2.1.2 | Serverless function adaptation strategies

The platform architecture determines where the adaptation components reside and how they interact (e.g., centralized vs. hybrid). However, the three main adaptations (i.e., scaling, placement, routing) govern the placement of function instances and backend services. We highlight their challenges in the edge-cloud continuum and briefly discuss implementations from the literature. This discussion highlights the complexity of designing function adaptations for serverless edge computing platforms, from which there are simulation use cases researchers and practitioners alike can benefit from. We categorize the challenges adaptation strategies face into application and infrastructure. Edge-cloud applications exhibit high amounts of spatio-temporal workloads. Due to systems spanning multiple cities or countries, the workload is generated at different rates in different places. Therefore, adaptation techniques must optimally place function instances in the edge-cloud continuum to process requests efficiently. Emerging application paradigms (e.g., Edge Intelligence) can have stringent requirements that platforms must fulfill. Moreover, these applications also exhibit high heterogeneity in resource usage and complexity. Complex AI pipelines are a prime example in which not only different hardware accelerators must take into account but also dynamic application flows can change the path of execution (i.e., which functions must be executed)[37,31]. We can split the challenges from the heterogeneous infrastructure into node-level and topology-level. Nodes offer different capabilities and capacities. Capabilities can be hardware accelerators or other physical attachments exclusive to specific devices (e.g., cameras); therefore, adaptations must be aware of correctly placing function instances that require capabilities. Multi-tenancy leads to performance interference, which can negatively impact resource-constrained nodes. Further, some capabilities are exclusive to individual function instances (e.g., hardware accelerators); therefore, adaptation techniques must carefully use those resources. The topology can consist of computing clusters that have different capacities and capabilities. Due to heterogeneous networks, nodes have different network conditions and can leave or enter the topology, causing instability. Simulations help understand the ability of function adaptation strategies to handle these types of heterogeneity. Previous challenges have been extensively covered in literature but are fundamental to orchestration strategies and serverless edge computing[1,38]. It becomes clear that implementing adaptation techniques in the edge-cloud continuum is complex. Therefore, extensive evaluation of adaptation techniques is crucial for a successful deployment. However, it also introduces several challenges the simulation must address for a viable solution. First, the simulation must incorporate well-known components in the core to evaluate adaptation techniques. These components include a scheduler, an autoscaler, and load balancers. Simulation users should be able to extend and implement novel solutions for these components. At the same time, the simulation should use a realistic domain model that maps simulation components to real-world ones. The last challenge revolves around the scenarios and request patterns during simulations. As previously mentioned, spatio-temporal context is important in edge-cloud systems (e.g., people moving around the city[16]), and simulations must be able to model this aspect.

Our simulation can support the development of serverless edge platforms by allowing rapid evaluation of different designs that propose individual solutions to these challenges. Therefore, we present simulation use cases that can help design and understand platform designs.

2.2 | Simulation use cases

Before discussing serverless edge computing platform simulations, we want to describe their use cases. Specifically, we want to highlight four use cases that can facilitate the design process of serverless edge computing platforms based on the tasks and their challenges identified in Section 2.1.1 and Section 2.1.2. Figure 3 shows each task's corresponding simulation use case and the simulation challenges it creates.

2.2.1 | Resource planning

Resource planning determines the necessary physical hardware capacities to serve workloads adequately[7]. The simulation can help understand platform providers and which architectures suit their infrastructure. They also can perform simulations in which the infrastructure varies, which can support future investments in new equipment. In addition, simulations can also determine the

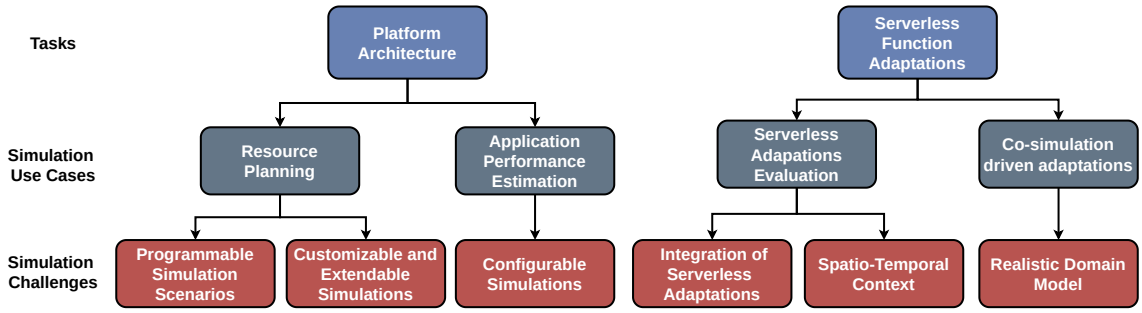


Figure 3 Simulation use cases that facilitate implementation of tasks for serverless edge computing platforms.

highest workload the infrastructure can withstand. The simulation tool must support automated topology and workload creation to perform many simulations to enable platform providers to identify possible interesting infrastructures.

2.2.2 | Application performance estimation

Application performance estimation has various facets and not only includes the estimation of performance-related indicators such as the response time of functions but can also include other factors such as Quality of Service and cost[20]. Customers can get estimates for deploying their functions, while platform providers can use them to see whether different platform architectures impact the cost in advance. The application performance estimation use case requires simulation users to freely adapt the simulation to their specific use cases.

2.2.3 | Serverless adaptation evaluation

It is challenging to operationalize adaptations in dynamic systems like the edge-cloud continuum, and requires continuous re-evaluation and optimization at both design-time and run-time. Adaptation approaches range from threshold-based over heuristic approaches to AI-driven strategies. Different ways of evaluating adaptation techniques exist. Testbeds and emulations help evaluate adaptation strategies on realistic setups that remain on a small scale[16,39]. They offer realistic results but lack scale and can be expensive. Additionally, results can be hard to reproduce because setups have many variables (e.g., OS version, library versions). While efforts exist to make testbed evaluations reproducible[14] (e.g., by streamlining the definition, execution, and analysis of experiments), many factors depend on resource configuration. Emulations bridge the gap between real-world behavior and scalability of simulations[40]. Both approaches rely on the available hardware and its configuration, making it hard to reproduce results. Simulations, while highly dependent on the implementation (i.e., performance, resource models, etc.), offer high reproducibility and can support large-scale scenarios. Researchers and system designers must be able to extend and develop their own serverless function adaptation algorithms into the simulation. A simulation tool should offer baseline adaptations, generic APIs to easily interface with the system (e.g., reasoning over the cluster topology, accessing simulated resource usage, or triggering function deployment), and ways to capture spatio-temporal context to model things like clients moving through the geo-distributed topology.

2.2.4 | Co-simulation driven adaptations

The co-simulation-driven adaptation use case focuses on operationalizing adaptations in real-world platforms. Serverless function adaptations rely on thresholds, heuristic-based optimizations, or AI-driven strategies. These adaptation strategies rely on parameter tuning, historical data, and possible optimization processes (e.g., AI training)[41,42]. Observing the system for changes is crucial once an adaptation strategy has been deployed. These changes can come from the applications (e.g., new applications or new requirements) or the environment (e.g., network saturation, node failure). In any case, the adaptation strategies must incorporate new circumstances during runtime. If adaptation strategies only interact with the real world and learn using historical data, updated configurations might need to be updated when deployed. One of the key characteristics of the edge-cloud continuum is its dynamic nature, and platforms have to autonomously and quickly adapt. In the best-case scenario, they can proactively update strategies to possible future scenarios. The concept of co-simulation tackles this issue by using an underlying simulation

engine to quickly update deployed adaptation techniques by simulating possible future scenarios[43]. Therefore, simulations must use a realistic domain model to provide interoperability between the implementation in the simulation and the real-world system.

3 | FAAS-SIM

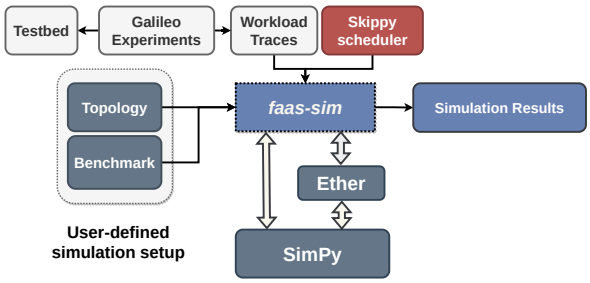


Figure 4 *faas-sim* overview

faas-sim is a trace-driven stochastic discrete-event simulation framework built on SimPy[24]. It was initially developed to evaluate how well *Skippy*[9], a generic container scheduling system based on the Kubernetes scheduling logic, could make trade-offs between data and computation movement in edge computing infrastructure scenarios. To that end, *faas-sim* primarily simulates serverless workload execution and network data transfer. Figure 4 shows an overview of simulation inputs, internal components, and simulation outputs of *faas-sim*.

The default workload scheduling system is Skippy, which is highly customizable and models a wide range of placement strategies but can also be replaced with a completely custom scheduler. To simulate workloads, *faas-sim* uses workload traces from profiling experiments on real-world testbeds to simulate the execution time and resource usage of running serverless workloads on different types of computing hardware. Our *Galileo* experiment framework[14] can facilitate the execution and preprocessing of experiments on a real-world testbed to generate traces for new functions and devices. For network simulation, it uses a higher-level flow-based network simulation that is part of the *Ether* edge network topology synthesizer[13].

All functionality *faas-sim* is implemented using Python and SimPy primitives. SimPy's simulation engine is single-threaded and uses a single event queue to model time progression in discrete steps. It leverages Python's generator concept, allowing users to implement simulation processes using the `yield` keyword to emit simulation events and programmatically advance the simulation clock. *faas-sim* uses SimPy's base mechanisms to facilitate the entire simulation and requires no additional extensions. As shown in Figure 4, *faas-sim* and *Ether* use SimPy and, therefore, can interact with each other in the process simply by in-memory communication using standard Python. A single Python process simulates the network and the serverless platform. A *Benchmark* is the programmatic execution of a user-defined simulation scenario, which may include workload generation based on specific patterns and allows users to plug in custom workloads or function simulators. The simulation result is a set of Pandas data frames that include fine-grained workload traces and network simulation results that allow the calculation of system key performance indicators such as average function execution, execution cost, resource usage, or network usage.

faas-sim provides pre-defined benchmarks, topologies, workload traces, and scheduling strategies, which users can individually customize. Specifically, *faas-sim* provides (a) traces from several common edge computing devices and representative serverless workloads such as AI-based image recognition, numerical computations, or AI-based speech-to-text; (b) request generators that model real-world workload patterns; (c) common edge computing system topologies and cluster configurations that are synthesized from real-world edge computing use cases[13]; (d) implementations of the container scheduling strategies presented in previous works by the authors[9,16] that can be used as baselines for new strategies.

We outline its conceptual model before explaining the implementation and architecture of *faas-sim* in Section 3.2.

3.1 | Conceptual model

The conceptual model of functions, deployments, and running replicas is shown in Figure 5. The API of the conceptual model can be found in a dedicated code repository[3]. It serves as a unified API collection, allowing implementations for different environments, such as *faas-sim*.

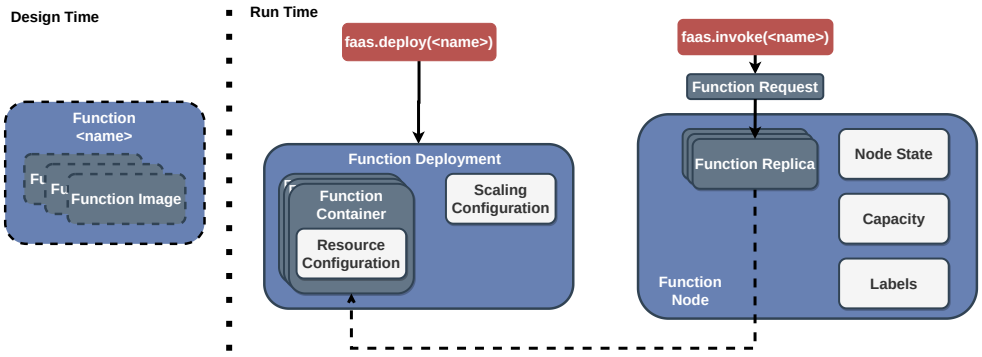


Figure 5 Conceptual model of functions and their deployment.

We split the conceptual model of functions into two parts: *Design time* and *Run time*. The *Design Time* includes *Functions* and *Function Images* and gives simulation users the possibility to change the implementation of functions without redefining *Functions*. It decouples the interface from the implementation and enables higher flexibility when defining functions.

- A *Function* is the highest level of abstraction and refers to some functionality identified by a name that can be invoked with a *Function Request*. For example, a *Function* could be an object detector named "detect-objects" that takes an image as input and returns the bounding boxes and labels of objects in the picture.
- A *Function* comprises several *Function Image* instances, one for each deployment platform. A *Function Image* is conceptually the code that implements a function on a specific deployment platform. For example, our "detect-objects" function could have one version that uses the GPU and one that uses a TPU (an AI accelerator). The reason for this additional abstraction is the way container platforms like Docker deal with multiple computing architectures. Docker images group different CPU architectures via a manifest to a multi-arch image. A *docker pull* command will pull the correct image based on the node's architecture. However, there is no way to include additional platform aspects such as GPUs or TPUs. If two container images exist for the same function, one that uses the CPU and one that uses the GPU, it may be ambiguous at runtime which image to pull. Instead, we want to allow the placement component to decide which image to deploy for a particular function.
- A *Function Deployment* is an instance of a *Function* with a concrete resource allocation and scaling policy configuration. A deployment consists of multiple *Function Container* instances and said configurations.
- A *Function Container* is the runtime configuration of a *Function Image*. It has a specific resource configuration that declares how many resources are allocated on a node when a particular replica of this *Function Container* is deployed. In our running example, a GPU-based "object-detector" might require less CPU but more VRAM than the CPU-based *Function Image*. A *Function Replica* is a concrete instantiation of a *Function Container*. It represents the actual running function (like a Docker container).
- Each *Function Replica* has a *Function Simulator* object that models the function's behavior for the simulation. Each *Function Replica* has a *Function Replica State* that signals the replica's current life cycle. The life cycles are inspired by Kubernetes and range from being scheduled for placement to running and shutdown.

[3]https://github.com/edgerun/faas

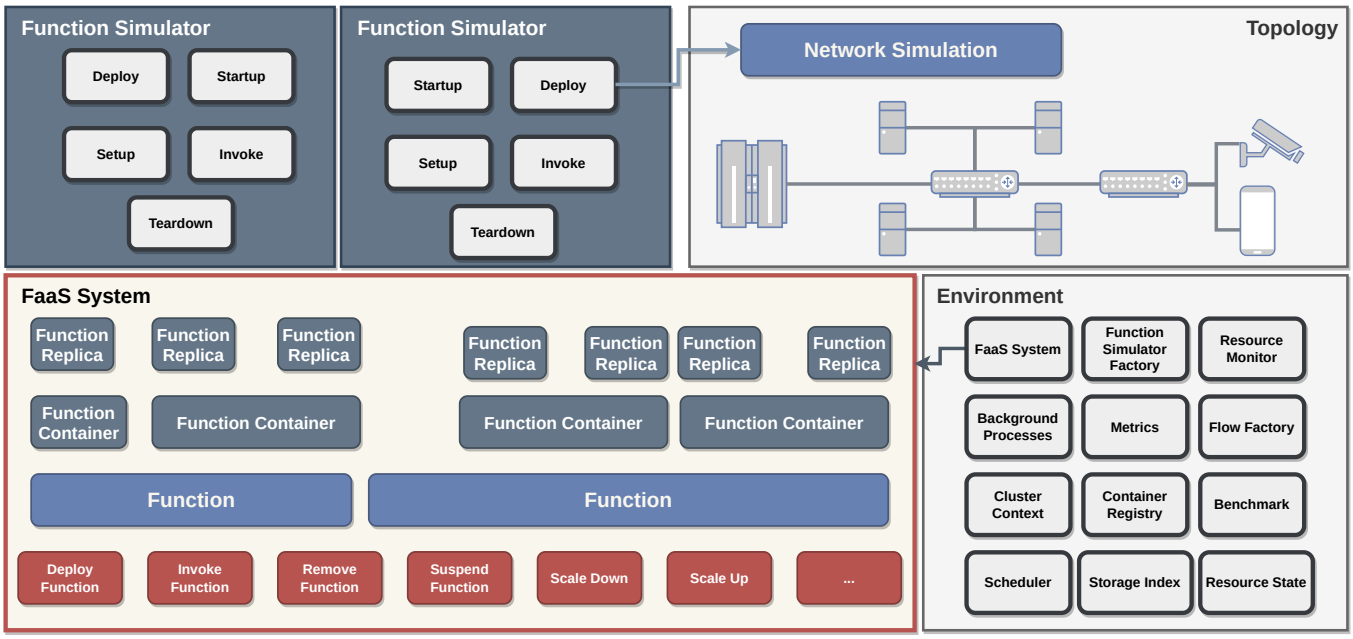


Figure 6 Main abstractions of *faas-sim*

- A machine capable of hosting function replicas is called *Function Node*. The *Node State* is a generic container for data needed during simulation time, for example, storing the number of concurrent invocations to a particular replica to calculate performance degradation. It also stores information that describes its CPU architecture, number of cores, memory size, and allocatable resources.

3.2 | *faas-sim* architecture

faas-sim's architecture allows the modification and extension to accommodate different serverless edge computing platform designs and execute various use cases. The architecture mainly consists of three abstractions: *FaaS System*, *Environment*, and *Function Simulator*. Figure 6 shows the internals of them, whereas *Environment* contains object references (e.g., to the *FaaS System*), while *FaaS System* and *Function Simulator* show methods to interact with them. The *FaaS System* acts as the central entity that exposes methods to interact with the platform. For example, it allows users to deploy functions, remove, invoke, or scale them up and down (also commonly referred to as *scale out* and *scale in*[44]). It manages a set of functions and the associated *Function Replica* and acts as the serverless edge computing platform's front end. The *Environment* contains references to essential objects used throughout the simulation. These objects contain the main logic of the simulation and can be configured and accessed via the *Environment*. For example, simulation users can replace the *Scheduler* (i.e., the placement component) by simply pointing it to their implementation. Each *Function Replica* has a *Function Simulator* containing the function's simulation model. Based on that, we structure the remaining section by first outlining the *FaaS System*, followed by the *Environment*. Both represent high-level components that encompass the simulation. Afterward, we detail how the *Function Simulator* works and our approaches to model resources, performance, and network as well as faults.

3.2.1 | FaaS System

The *FaaS System* abstraction is the high-level interface a simulation user interacts with. It is inspired by open-source FaaS platforms (i.e., OpenFaaS[25]) and container orchestration services (i.e., Kubernetes[45]). The API of the *FaaS System*[4] presents the frontend of a serverless edge computing platform and lets simulation users deploy, remove, scale, and invoke *Function Deployments*. The following shows an excerpt of available methods:

[4]The source code of this class is available under https://github.com/edgerun/faas. However, *faas-sim* provides an implementation called `DefaultFaaS System`

- `deploy`: makes the function invokable and deploys the minimum number of *Function Replica* instances on the cluster. The number of minimum running instances is configured via *Scaling Configuration*. *FaaS System* creates for each *Function Replica* a new *Function Simulator* using the *Function Simulator Factory*. Each *Function Replica* gets a *Function Node* assigned using the *Scheduler*. The *Cluster Context* is updated accordingly, and the *FaaS System* starts the *Function Replica* life cycle.
- `invoke`: the *Load Balancer* selects the replica and simulates the function invocation by calling the `invoke` method of the *Function Replica's Function Simulator*.
- `remove`: removes the function from the platform and shuts down all running *Function Replica* instances, calling the `tearown` method of each.
- `discover`: returns all running *Function Replica* instances that belong to the *Function Deployment*.
- `scale_down`: removes the specified number of running *Function Replica* instances, with respect to the minimum requirement. The current implementation first picks the most recently deployed *Function Replica* instances, which is Kubernetes' default behavior[44]. However, simulation users also can pass a list of *Function Replica* to remove. This operation is equivalent to the commonly used *scale in* operation[44].
- `scale_up`: deploys the specified number of *Function Replica* instances but has to respect the maximum number specified in the *ScalingConfiguration*. As with `scale_down`, the simulation user can also pass a list of *Function Replica* instances to schedule, and the operation is commonly referred to as *scale out*[44].
- `poll_available_replica`: repeatedly waits and checks for running *Function Replica* instances of the *Function Deployment*.

The API exposes additional lookup methods that we did not include above. The *FaaS System* manages functions using the *Environment* and is responsible for invoking all life cycle phases of the *Function Simulators*. Using the *Environment* allows simulation users to inject their *Load Balancer*, *Scheduler*, and other objects that the *FaaS System* uses.

3.2.2 | Environment

The *Environment* is the central instance that stores object references used throughout the simulation. Figure 6 highlights most of the objects currently in the *Environment*. This might change over time as the simulation is continuously updated to implement and provide more features out of the box. However, we explain some of them in detail in the following and show which features they enable.

- The *Cluster Context* keeps track of available *Function Container* images available and resources (i.e., CPU and memory) on each node. This enables the simulation only to download images unavailable on the node and the *Scheduler* to check if a node has enough resources.
- The *Function Simulator Factory* is a component that simulation users have to provide and determines which *Function Simulator* instance is assigned to a given *Function Replica* upon creation.
- The *Topology* contains all nodes and the network graph and can be created using the external library *Ether*, which also provides the network simulation. *Ether* implements a flow-based simulation that simulation users can transparently change by modifying the *Flow Factory*, which determines the concrete type of network simulation.
- The *Scheduler* has to implement a `schedule` method, which takes a *Function Replica* and decides on which node it should be placed. The default implementation of *FaaS System* uses a SimPy queue to process each requested *Function Replica* sequentially and use the *skippy-scheduler*[9]. However, simulation users can replace the *Scheduler* implementation or modify the *FaaS System* implementation to suit their needs. For example, simulation users can change the default monolithic scheduler architecture to decentralized.
- The *Resource State* keeps track of the resources across nodes and *Function Replicas*. This object acts as storage and offers methods such as `put` and `remove`. The simulation user is responsible for using those methods, *faas-sim* does not restrict

what kind of resources are stored. The *Resource Monitor* can measure resource usage repeatedly during the experiment by fetching the resource usage for all running function replicas. This approach simulates a pull-based monitoring strategy, in which a central controller is responsible for fetching the latest resource usage from all replicas, which mimics the behavior of real-world monitoring frameworks, such as Prometheus[5]. However, users can also modify the *Resource State* to log every *put* and *remove* action without using the monitor.

- The *Metrics* object is the central logging entity and stores everything in a table-like data structure for export at the end of the experiment (e.g., as CSV file). The class offers some methods tied to the simulation, which *faas-sim* automatically logs (i.e., the scheduling process), but simulation users can log arbitrary data. Simulation users can also implement the *Metrics* class to support other storage mechanisms (e.g., SQL databases) to improve performance and keep memory usage low.

- Simulation users populate the *Container Registry* before starting the simulation with their *Function Containers*. Each *Function Container* has a CPU architecture (i.e., `arm64v8`, `arm32v7`, `amd64`) and size of the image in bytes. The *Container Registry* is a node in the *Topology* which enables the simulation of downloading the *Function Container* onto the node.

- The *Background Processes* object is a list that simulation users can populate. SimPy has a built-in concept of processes running in the simulation's background. The list of processes is automatically started at the beginning of each simulation and allows users to run processes continuously. For example, the *Resource Monitor*, which periodically fetches the resource usage of each *Function Replica* and node, is a background process. Another critical component in the simulation of serverless edge computing is the *Autoscaler*. *faas-sim* includes a variety of *Autoscalers* that can be included in the list of *Background Processes*. Each *Autoscaler* implements a `run` method that repeatedly observes the platform state and invokes the *FaaS System's* `scale_up` and `scale_down` methods.

- The last concept we include in *faas-sim* is the *Storage Index*, which enables users to model object storages that are popular in serverless computing and are mainly used to store data that functions have processed. Therefore, *faas-sim* can realistically simulate the network transfer between *Function Replicas* and storage nodes.

Having access to all important aspects of the simulation (e.g., *FaaS System*) enables the implementation of *Function Simulators* that act as *Load Balancer*, *Autoscaler*, *Scheduler* and arbitrary components.

Before describing our evaluation, we want to define the *Simulation* and input parameters (i.e., *Topology* and *Benchmark*) in more detail.

3.2.3 | Trace-driven function simulators

The main goal of *faas-sim* is to determine the function execution time (FET) and resource usage of running workloads on particular nodes. *faas-sim* is trace-driven, which means that users need to provide measurements, or estimations, from real-world experiments to feed the simulation. We explain this in more detail in Section 3.2.4. In this section, we present the *Function Simulator* abstraction, which allows users to simulate various stages of a function that ultimately determine the FET and resource usage.

While we ship a set of functions with *faas-sim*, we also enable simulation users to profile new functions on their hardware using *Galileo* experiments[14]. In the following, we focus on our profiling approach but want to emphasize here the flexibility of our simulation. As we mentioned, the simulation does not make any assumptions about resource usage, and users must inject their traces. Further, *faas-sim* also does not have an integrated resource model, and simulation users must implement their own. However, *faas-sim* offers functionality to implement a resource model as we describe in Section 3.2.2. Figure 7 shows the *Function Replica* life cycle phases and denotes for each one the data we must store to enable the trace-driven simulation. For example, we calculate the resource usage of a single function invocation and use this in the simulation. We also use the duration for each phase to simulate it accurately. The phases, depicted in Figure 7, correspond to the phases that the *Function Simulator* offers to implement.

The *Function Simulator* is the core of *faas-sim* and implements the behavior of a function. Deploying a *Function Replica* consists of the following life cycle phases. The *Function Replica* is *deployed* on the node (e.g., by pulling the container image if it is not present), and *starts* the *Function Replica*. Afterward, the replica can execute some *setup* code and then is ready for *invocation*. In the end, the replica is *tore down*. To simulate this life cycle on a node, a `FunctionSimulator` class must

[5] https://prometheus.io/

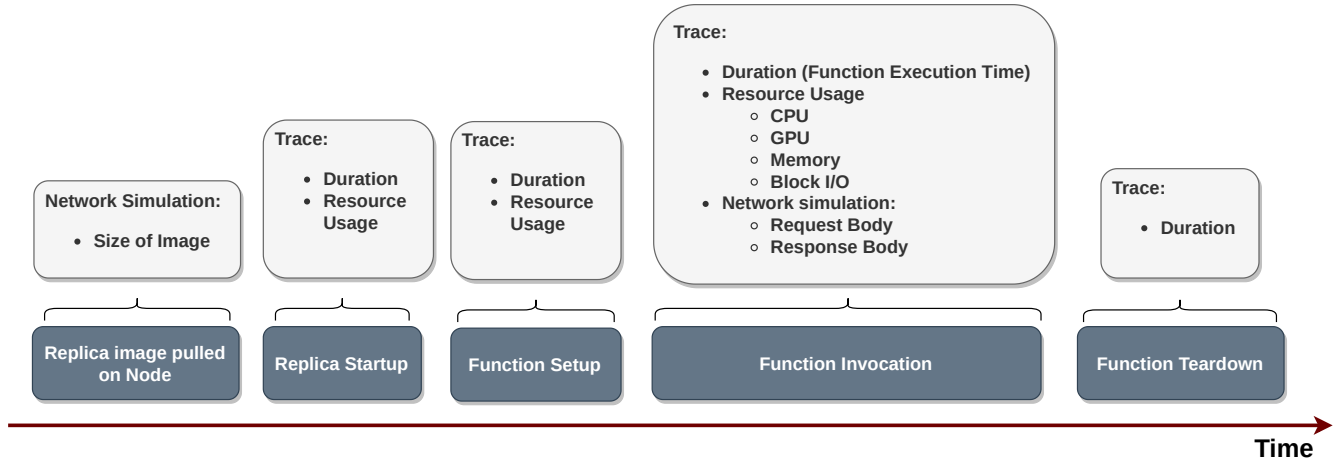


Figure 7 Trace-driven analysis process

be implemented with the following methods. Each method yields SimPy simulation events, which enables simulation users to implement arbitrary complex simulation models of functions.

- `deploy`: deploys the *Function Replica* on the node (e.g., by pulling the container)
- `startup`: starts the *Function Replica*
- `setup`: executes any setup code of the *Function Replica*
- `invoke`: simulates the invocation of the *Function Replica*
- `teardown`: cleans up and stops the running *Function Replica*

Figure 8 shows the life cycle phases and puts the real-world execution and function code next to the simulation code. It also depicts the different states the replica is in during the simulation, from starting to shutting down. The deployed function offers AI inference and initially loads an AI model into memory, which is kept in memory to speed up inference.

Before going into details, we want to highlight three things: (1) in some phases `docker` commands are shown. Typically, the underlying orchestration service executes these commands, and we include them only to simplify the explanation. (2) the simulation is not tied to Docker or container virtualization techniques and can model the behavior of other technologies. (3) this example highlights many aspects of the function execution model, which are all optional,i.e., simulation users can decide which of them they include.

In the *deploy* phase, the `docker pull` command downloads the function image from the container registry. The simulation model simulates a network download using *Ether*, which downloads the Docker image size from the container registry if it is not already on the node.

The `yield` statement enables SimPy to track time spent and includes the simulated download time from *Ether*. Afterward, the `docker run` command starts the function, which we simulate using the time taken from the traces. The `env.timeout` call lets the simulation wait. SimPy has no built-in concept of units, leaving this open to the simulation users. Our traces are in seconds; therefore, `startup_delay` is also in seconds. Next, the simulation includes the container's resources after the startup (e.g., CPU usage, memory). The delay in starting the function and the resource usage comes from the trace analysis the simulation users must provide. The *Function Replica* transitioned in the meantime to the *starting* state and began the *setup* phase. The *Function Replica* only consumes the base resources. As we describe in Section 3.2.2, *faas-sim* is not built around a specific resource model but offers a resource state object that acts as storage. For example, we can measure the function's idle CPU usage, store it, and afterward remove it. Further, the *Function Simulator* methods accept (among other arguments not shown for conciseness) an *Environment* (i.e., `env`) argument, which gives access to the resource state. In the *setup* life cycle phase, the container loads an AI model into memory. The simulation models the delay of loading the model and claims resources consumed by the AI model. After the *setup* phase, the replica is *running* and can be *invoked*. In the real world, the function invokes the AI model with the request (e.g., an image). However, in the simulation, the *Function Simulator* first claims resources that are used due to

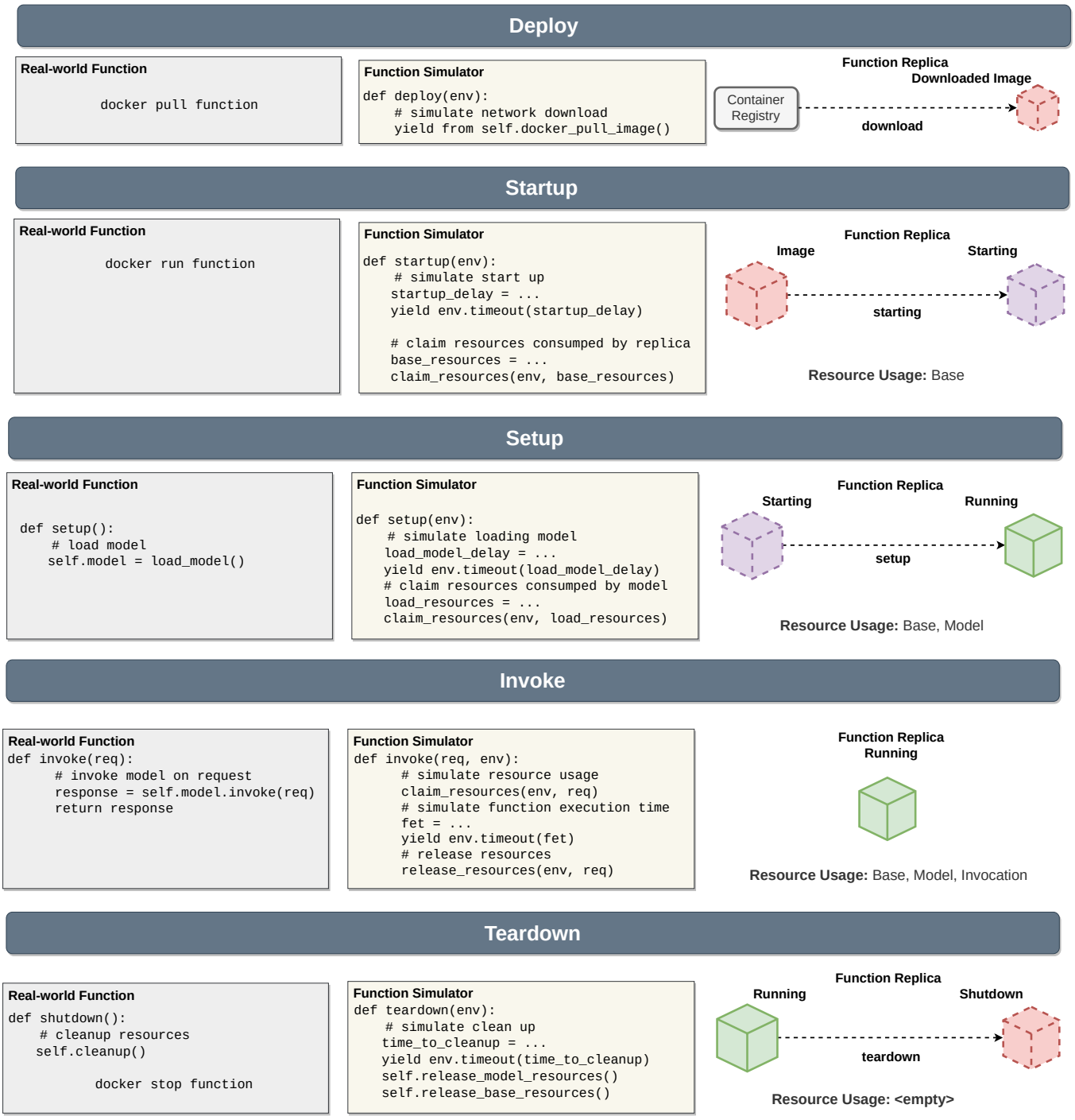


Figure 8 Life cycle phases of AI inference function based on OpenFaaS' HTTP *of-watchdog*.

the inference process and simulates the inference by waiting for the time we derived from the traces and afterward releases the resources. While this example is kept simplistic on purpose, *faas-sim* users can implement sophisticated simulations that model more complex aspects of a function. For example, the execution duration can depend on the input (e.g., small vs. large input), and simultaneously, simulated resource consumption can also rely on the input. To achieve this, a formula (either using profiled traces or ML models) is needed that can estimate the execution time. In contrast, the parameters of this formula can depend on the input size, resource usage based on the input, etc. This allows the modeling of functions that use different resources (e.g.,

I/O, CPU) based on the input and other factors. The resource modeling approach of *faas-sim* is explained in Section 3.2.5, and we later show how it can be used to implement complex multi-tenant scenarios in Section 3.2.6 and Section 4.1.1.

Eventually, we can stop the function replica. In the real world, we call `docker stop`, and the container receives a signal to shut down (i.e., leads to the invocation of `shutdown`). The *Function Simulator*'s `teardown` method is called, and it waits for a specific time, simulating the clean-up process, and then releases all resources.

The *Function Simulator* is flexible, and simulation users can adapt it to their needs. We show in Section 4.4 how they can mimic the behavior of OpenFaaS functions.

3.2.4 | Trace-driven performance modeling

As mentioned, *faas-sim* is trace-driven and relies on the results of basic profiling benchmarks that *faas-sim* then uses to model more complex behavior. Existing simulation systems for edge and cloud computing, such as CloudSim[46] and its numerous extensions[23,22], use a performance model based on discrete values of CPU instructions. The execution duration is calculated using the number of CPU instructions of a workload and the CPU's instruction rate in instructions per second (IPS)[47]. This approach has several limitations. Depending on the computer architecture, not every instruction takes up the same amount of CPU time, meaning that the IPS is not static for the CPU but depends on the mix of instructions. Further, the CPU speed varies significantly in the edge-cloud continuum[48], and functions do not only rely on the CPU speed but also on other resources, such as I/O. Additional factors determine task execution duration on a CPU, including L1 and L2 cache sizes or I/O access density. Ultimately, even these approaches have to estimate or profile the number of instructions of a function invocation and the CPU, similar to our profiling experiments. The effort of determining these data is equivalent to running a complete set of profiling experiments, meaning there is no benefit over a trace-driven model.

In Section 3.2.3, we introduced the *Function Simulator* and have shown which traces the simulation can use during the simulation. The *Invoke* step of our example shown in Figure 8 has a placeholder for determining the function execution time (FET), which we want to elaborate on. Generally, we simulate FET using an *oracle* that determines the FET based on the workload, the node executing the workload, and the current resource utilization. We model performance and performance degradation by fitting functions on the distribution of workload traces for a particular workload and compute device. During simulation time, the oracle samples from the fitted distributions to determine the FET, and the simulation system records how many requests are being executed in parallel.

While *faas-sim* is not tied to a specific performance model to determine FET and resource usage, we want to show and evaluate the model that *faas-sim* ships with. To that end, we first explain how we model resources of cluster nodes, and then how we use stochastic performance models to simulate performance degradation and multi-tenancy.

3.2.5 | Resource modeling

Our resource model approach is based on traces gathered during workload and device profiling experiments. Similar to our performance model, which is centered around the execution time of one function invocation, the resource model allows us to describe the resource usage of one function invocation. Based on the traces, we calculate a resource vector representing the utilization of the following resources:

- u_{cpu}: the CPU utilization,
- u_{io}: the block I/O in bytes/sec,
- u_{net}: the network I/O in bytes/sec,
- u_{gpu}: the GPU utilization (if available), and
- u_{ram}: the RAM usage

Each resource represents the mean usage for one function invocation. We are using *telemd*[6], a lightweight telemetry daemon that continuously collects and publishes data. *Telemd* enables fine-grained profiling on the container level (i.e., CPU, network & block I/O, and memory). The GPU utilization is measured on the system level, and measures must be taken to avoid interference. Details can be found in[49]. The resource vector can be subsequently *claimed* during *Invoke* step of the function simulator. Adding

[6]https://github.com/edgerun/telemd/

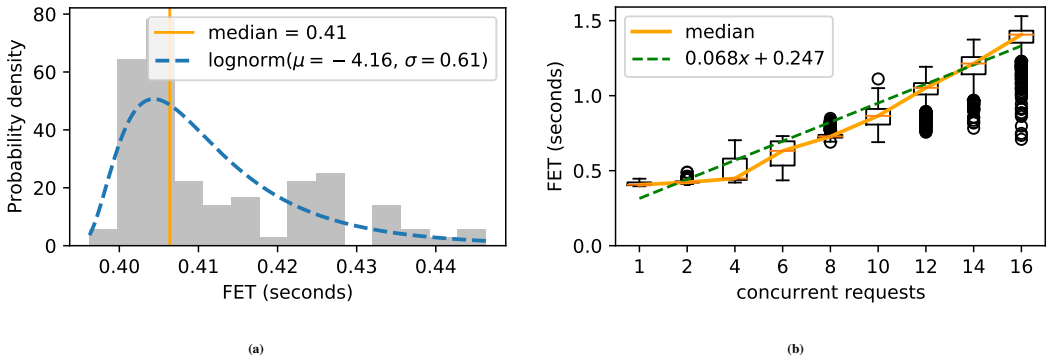


Figure 9 Example function execution times (FET) of running an SMT function. (a) Distribution with single request $n = 100$, (b) performance degradation with increasing number of concurrent requests

the resources during each function invocation facilitates estimating performance degradation in multi-tenancy scenarios, as shown in the following section.

3.2.6 | Stochastic performance models

Stochastic performance modeling embraces the fact that there is variance in the execution duration of a task, even if the conditions appear equivalent. When performance degrades because of concurrent execution of tasks and resource contention, not only does the function execution duration increase, but the variance also increases. Modeling this behavior is particularly relevant for systems that need to balance load between workers. There are two distinct scenarios: single-tenant performance degradation and multi-tenant performance degradation. In the single-tenant case, only one type of workload is executed on a node, and the performance degradation can be formulated simply as a function of the number of concurrently executing processes. In the multi-tenant case, multiple workloads may require heterogeneous amounts of resources, which is much harder to model.

Single-tenant performance degradation

To illustrate how *faas-sim* enables this, we present an example from *faas-sim* that simulates the execution of Satisfiability Modulo Theories (SMT) workloads.

Figure 9a shows the traces and fitted probability density of the function execution from an experiment in which we profiled a particular SMT solver on an edge device. It shows the particular parameters for the fitted log-normal distribution. Our performance modeling approach samples during simulation time from the log-normal distribution for each invocation. Figure 9b shows how the FET distribution changes with increasing concurrent requests due to resource contention. The dotted line shows a linear regression fitted over the medians of the distribution.

Based on these observations, we can generalize a simulation model as follows. During simulation time, we want to estimate the FET for the workload (serverless function) f on node n given the current number of concurrent requests r. The parameters are available in the simulation system as part of the environment (see Section 3.2.2). First, we fit a linear regression to estimate the median FET based on the function and the current number of served requests. Then, we add noise sampled from the log-normal distribution we fitted on the experiment with a single workload. We can then define the FET as a function $\hat{t}_{f,n}$ of r, where f is the SMT function, and n is the particular edge device of the profiling experiment. For the example given in Figure 9, the concrete parameters would be defined as:

$$\hat{t}_{\text{smt,edge}}(r) = 0.068 \cdot r + 0.247 + X \tag{1}$$

Where X is the log-normal distributed noise: $X \sim Lognormal(-4.16, 0.61)$. This yields a simple but reasonably accurate simulation model that can be plugged into a *faas-sim* Function Simulator. In previous work where we used the same stochastic modeling technique for simulating the execution of AI workflows, we have shown that this approach leads to very good results[50].

Multi-tenant performance degradation

Besides the single-tenancy case, which is based on fitting simple parameterized distributions, we also explored an ML-based approach to predict performance degradation occurring in multi-tenancy scenarios[49]. Our approach is based on the previously

described resource vectors we assume to have for each deployed function. We integrate this model as follows. At each function invocation, we sample the execution duration based on the stochastic performance model and collect the current resource usage afterward. The resource usage is used as input to invoke our ML model, which returns the factor that worsens the performance. For the final execution duration, we multiply the performance degradation factor by the initially sampled duration. The input is based on the function requests being executed at the moment and base resource usage as well as hardware accelerators (e.g., GPU). The single-column vector has a fixed length of 34. It contains the sum, mean, standard deviation, minimum, maximum, 25^{th}, 75^{th} percentile of CPU, GPU, network, and block I/O usage, mean memory usage, and the number of running containers. The difficulty of this approach lies in training a good model, which we describe in more detail and present results in Section 4.1.1.

3.2.7 | Network simulation

A network simulation is fundamental to simulating interactions between nodes and workloads within a serverless system. Network topology, capacity, and usage significantly impact the performance of serverless systems, so having a reasonable simulation model for networking is essential. Examples of interactions in our model where the impact of networking is high include:

- downloading a *Function Container* onto a cluster node.
- transferring data from storage nodes to a *Function Replica*.
- transferring data between functions (e.g., request and response data).

Our network simulation is based on the network model of Ether[13], which is more high-level than packet-level simulators such as ns-3[51] or OPNET[52]. These simulators precisely model the low-level interaction between network protocols and networking hardware, which differs from our goal. Instead, our simulator implements a high-level network model on top of topologies based around *flows*, representing data transfers between nodes through several links. This way, we trade off fidelity for simulation performance.

The conceptual model of Ether is simple and has three core concepts that are relevant to the network simulation: (a) **node**: a computing or storage devices within the network; (b) **link**: anything that facilitates a connection between nodes in the network (e.g., a network card, a WiFi access point, or a mobile network uplink), which has an associated *data rate capacity* (bandwidth); (c) **topology**: a graph that models nodes and links as vertices and connections as edges that can hold QoS attributes such as latency.

Simulating data transfer between nodes involves opening a *flow* through several connected networks *links*, i.e., a route. This is a very common model for simulating and reasoning over networks[53,54,55]. Our network simulation implements a simple shortest-path routing through the topology and fair link bandwidth allocation across flows. A flow can be considered a stream of data between two nodes that uses the bandwidth of links. Each link has a certain amount of bandwidth, and we implement fair bandwidth allocation across flows. When a data transfer between node n_1 and n_2 is simulated, a route is computed with a shortest path algorithm. The route contains all links along that path between n_1 and n_2. All existing flows that use links along the route are interrupted, and their bandwidth is reallocated according to the fair bandwidth allocation scheme. A flow will only allocate as much bandwidth on links as the lowest bandwidth a link across the route (in other words, the bottleneck of the topology) can provide. For calculating the data transfer duration of the flow, we use the following model:

$$\text{duration(flow)} = \text{round trip time} \cdot 1.5 + \left(\frac{\text{bytes to transfer}}{\text{goodput}} \right) \tag{2}$$

In other words, the time to establish the TCP connection (which is bound by the link latency and multiplied by 1.5 to reflect the TCP handshake procedure), plus the ideal time to transfer the given amount of bytes. The round trip time is calculated by summing up the latency values of all connections between links along the route. The goodput data rate, i.e., the application-level throughput of communication, is estimated from the allocatable bandwidth across the route in bytes per second, multiplied by a magic number of 0.97 that captures roughly the TCP protocol overhead.

$$\text{goodput(flow)} = 0.97 \cdot \min_{\text{link} \in \text{route}} (maxAllocatablePerFlow(\text{link})) \tag{3}$$

The function $maxAllocatablePerFlow$ implements bandwidth allocation based on a max-min fairness notion[53], and the outer call to min finds the bottleneck in the route. The details of our algorithm implementation can be found in our code repositories [7].

[7] https://github.com/edgerun/ether/blob/master/ether/core.py

The choice has no particular significance, but it is known that TCP congestion control converges to a balanced allocation[56]. When simulating edge computing systems, more research is needed to determine the accuracy of different bandwidth-sharing models, such as proportional fairness[54].

Limitations

Although the simulation model is straightforward, its accuracy in the basic scenarios we have investigated is comparable to that of packet-level simulators like ns-3, as shown in our evaluation of the network simulation accuracy in Section 4.3. However, the simplicity of the model comes with some limitations. TCP congestion control using additive-increase/multiplicative-decrease (AIMD) is known to converge to an equal allocation of a contested link between flows[56], our max-min fairness allocation does not model the process of converging to that allocation, which takes time and may affect the result of individual flow simulations. Also, TCP is known to have degrading performance behavior when there are many parallel flows[57], which still needs to be explicitly handled in the current model. Moreover, the goodput of a flow can be negatively affected by TCP packet loss, which is not simulated and likely harder to implement since we do not simulate on the packet level. It is currently unclear to us when and how such behavior would impact the conclusions drawn from simulation results, given that network conditions are often only one aspect of the systems that *faas-sim* targets. For example, in the evaluation of Skippy[9], an optimizing container scheduler for geo-distributed scenarios, the network simulation played a significant role in identifying network bottlenecks. However, aspects such as the workload execution time or computational resource contention had a similar or higher impact, such that the additional fidelity of modeling TCP loss rate, which Morris[57] showed can range from 1%-5% with 40-140 active TCP flows, would have no significant impact on the overall result of the evaluation. To understand this more and to be able to generalize, we invite the community to contribute baseline scenarios that can provide more evidence for whether this type of fine-grained simulation is needed for edge system evaluations.

3.2.8 | Fault modeling

Faas-sim supports out of the box two common classes of faults in distributed systems, specifically in the edge-cloud continuum: network degradation and replica failure. The default network simulation configuration throws an error when the bandwidth of a link is under a certain threshold. This simulates the exhaustion of network connections that can quickly saturate in resource-constrained environments (e.g., a Raspberry Pi 3 has a 100 Mbit/s bandwidth). Moreover, users can implement methods that randomly shut down functions or make nodes unavailable. The flexible *Function Simulators* can provide more fine-grained faults, such as resource exhaustion during invocation. For example, a function invocation might consume too much memory and cause the function and possibly others on the same node to shut down abruptly.

3.3 | User-defined simulation scenarios

A simulation encapsulates the configuration and the runtime state of a simulation. It requires two inputs: a *Topology* and a *Benchmark* which can be built using *request generators*.

3.3.1 | Topology

The simulation takes an *Ether*[13] topology as an input parameter. This enables users to generate network topologies in code and synthesize plausible infrastructures. As we described in Section 3.2.7, *Ether*'s fundamental concepts, which are tightly integrated into *faas-sim*, are nodes, links, cells, and the topology. A node represents a compute or storage device, and users can set the CPU and memory resources and label it to expose any additional capabilities (e.g., hardware accelerators). Links represent connections between nodes, such as access points, routers, and switches. Each link can be configured to have a specific data rate capacity. Cells represent a group of nodes, links, or other cells and help users to compose larger computing clusters. Topologies represent these components' graph representations and connect the nodes, cells, and links.

faas-sim provides several *Ether* topologies for common edge computing scenarios out of the box. These include: (a) an Industrial IoT scenario with distributed premises and shared cloud infrastructure, (b) an urban sensing scenario based on data from the Array of Things project[58] that connects distributed IoT nodes through mobile Internet, (c) a multi-region cloud scenario with several cloud data centers connected via an Internet backbone. Users can use *Ether* to generate topologies for their scenarios, but the pre-defined scenarios can serve as a baseline. These topologies have been used to successfully evaluate edge computing resource management algorithms, as shown in Section 4.

3.3.2 | Benchmark

The *Benchmark* is a container for simulation parameters and the workload configuration to model a particular experiment scenario. Such scenarios could be: (1) generate N sequential requests for function f; (2) for t number of hours in simulation time, generate requests for functions f_1, f_2, and f_3 based on random workload patterns; or (3) for t number of days in simulation time, create u number of workload generators (representing users in the scenario generating requests) that generate requests for function f in a sine-based workload pattern. *faas-sim* provides several such scenarios as examples but will ultimately have to be provided by the user depending on their use case and evaluation scenarios.

Specifically, the *Benchmark* is a SimPy process implemented in Python that prepares the evaluation environment by preparing the different workload types and then generates requests based on some logic. Simulation users must implement two methods: `setup` and `run`. The `setup` method has access to the *Environment*, and users can initialize objects, such as the container registry (i.e., register available images). The `run` method is invoked to bootstrap the actual simulation scenario and start the workloads. Simulation users can also deploy *Function Deployments* at this stage and use our request generators to produce workload on the deployed functions.

3.3.3 | Request generators

faas-sim allows programmable workload generators to produce realistic invocation patterns. These generators are located in the *request-generator* project on GitHub[8]. They are composed of an arrival process and a workload pattern. The arrival process determines the distribution from which the inter-arrival times, i.e., Δt between requests, are drawn. The *request-generator* offers as of now: *constant/static* and *expovariate* processes. The workload pattern determines the average target requests per second (*rps*) pattern at a given time. Several workload patterns are supported, such as *constant*, *sine*, and *random walk*. Simulation users can combine the arrival process and workload patterns freely. These tools allow the modeling and creation of dynamic workload patterns and can simulate realistic ones. Moreover, *faas-sim* also supports files that contain a list of inter-arrival times. This enables users to export workloads and replay them, increasing the reproducibility of simulations and allowing users to model workloads without our tool. Specifically, simulation users can take existing datasets[59] or synthesize their own[60] and pre-process them into a list of inter-arrival times our simulator supports.

Figure 10 depicts possible combinations of arrival processes and workload patterns.

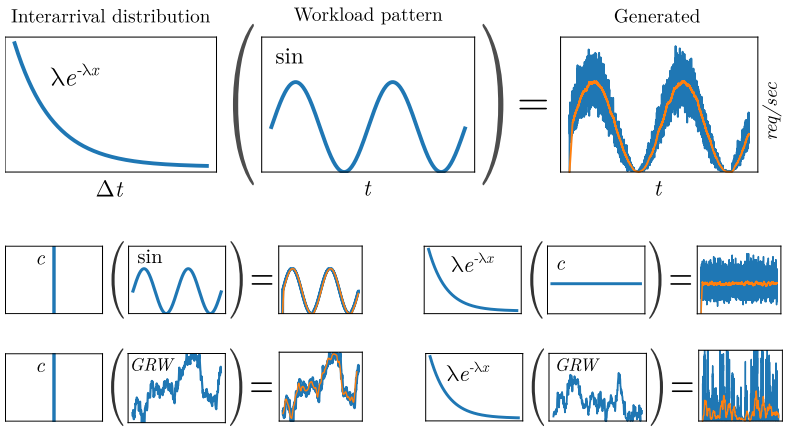
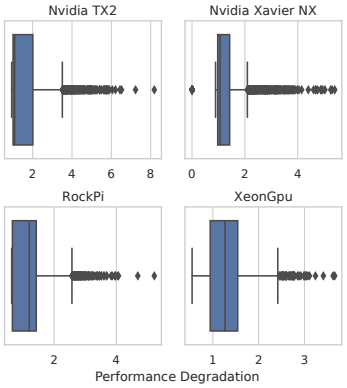


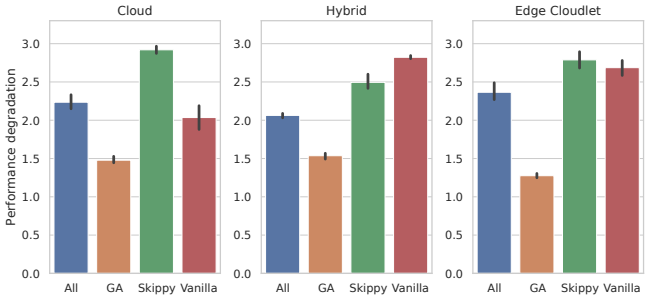
Figure 10 Generating workloads by combining inter-arrival distributions with workload patterns, taken from[61].

The first row illustrates the combination of the *expovariate* interarrival distribution and the *sinusoidal* workload pattern. The orange line should match the workload pattern and display a moving average of *rps*. The left column shows how the *constant* distribution replicates the workload pattern, and the right column shows how the *constant* workload pattern with a *expovariate* distribution models a static workload pattern with randomized intervals. To model highly fluctuating workloads, the bottom right combines the Gaussian *random walk* workload pattern with the *expovariate* distribution.

[8]https://github.com/edgerun/request-generator/



(a) Performance Degradation on edge devices



(b) Performance Degradation on edge devices

Figure 11 Performance degradation in real-world experiments and in simulation.

4 | EVALUATION

The evaluation consists of five parts: We showcase a selection of published works that have used *faas-sim* and highlight their results. This shows the ability of *faas-sim* to be used for the use cases shown in Section 2 and addresses the challenges introduced in Section 1. Moreover, based on our resource and performance modeling approaches, we highlight functions profiled included in *faas-sim* that can bootstrap scenarios without additional experiments in Section 4.2. The high heterogeneity in terms of performance duration of the traces emphasizes the necessity for stochastic models, as presented in Section 3.2.6. Section 4.3 presents our network simulation experiments comparing results from *Ether* with a testbed and ns-3[62]. Section 4.4 shows the flexibility of our *Function Simulator* approach. Lastly, Section 4.5 presents experimental results that the resource usage of *faas-sim* for different scenarios.

4.1 | Simulation use cases

This section showcases selected works using *faas-sim* for the use cases outlined in Section 2.2. We highlight their approaches and results and reflect on the introduced challenges. These works range from ML-based multi-tenancy performance models, serverless adaptations (including the evaluation aspects), simulation of different infrastructure topologies, and simulation-driven adaptations. This use-case-based evaluation also shows that *faas-sim* tackles the simulation challenges.

4.1.1 | Application performance estimation

In[49], we introduced a novel performance degradation prediction approach that estimates the performance degradation factor based on the current resource usage. This approach models the impact multi-tenancy has on performance. The ML-based approach was developed by executing various functions (e.g., AI inference, CPU heavy) on various devices (e.g., Raspberry Pi 4 to an Intel NUC).

Figure 11a shows the results of these experiments as boxplots where the performance degradation is the factor of increase in contrast to the mean execution duration. Due to hardware limitations, not all functions were executed on all devices (e.g., on the Rock Pi only a few can run). Besides the performance degradation, we also measured the resource usage to create for each function its resource vector. The resource vectors are, in turn, used to pre-process the ML model's input. The input is a vector describing the current resource usage based on the function requests being executed at the moment and base resource usage as well as hardware accelerators (e.g., GPU). The single-column vector has a fixed length of 34. It contains the sum, mean, standard deviation, minimum, maximum, 25^{th}, 75^{th} percentile of CPU, GPU, network, and block I/O usage, mean memory usage, and the number of running containers. During the simulation, we determine which functions are currently being executed on each node and use the associated resource vector we presented in Section 3.2.5 to form the input. We used TPOT[63], an AutoML tool, to find a suitable ML pipeline, and validation results have shown good performance (i.e., the mean absolute error ranged from 0.02 to 0.09).

The final pipeline had the following configuration, and our approach fits each unique device with its own ML model:

```
ExtraTreesRegressor(CombineDFs(DecisionTreeRegressor(Binarizer(AdaBoostRegressor(
input_matrix, learning_rate=0.001, loss=square, n_estimators=100), threshold=1.0), max_depth=10,
   min_samples_leaf=17, min_samples_split=9), PCA(input_matrix, iterated_power=4, svd_solver=randomized)),
   bootstrap=False, max_features=0.8500000000000001, min_samples_leaf=1, min_samples_split=4, n_estimators=100)
```

The model's accuracy shows that our resource model can be used to integrate sophisticated approaches to estimate the impact of multi-tenancy. Thus, it allows users to get reasonable estimates for their application performance through simulations and shows that *faas-sim* can support high configurability simulations by extending the core resource model. A limitation of this work is that it has to be investigated how well it generalizes for new, unseen functions as the training and evaluation focused on a limited set of functions. Moreover, we trained an individual model for each device, thus increasing the effort to include new devices as they would need to be profiled accordingly.

4.1.2 | Evaluating serverless adaptation approaches

Evaluating serverless adaptation approaches is challenging on real-world testbeds, given the complexity of experiments necessary to draw generalized conclusions about the performance of such approaches. Simulation tools like *faas-sim* that can easily generate variations of computing infrastructure and workload patterns allow much more meaningful evaluations. The simulation should be able to mimic the spatio-temporal context of real-world applications. That is the distance between the request origin and the execution location and the varying request patterns over time. *faas-sim* has been used to evaluate different adaptation approaches, ranging from optimization-driven approaches that avoid performance interference and decrease performance degradation due to multi-tenancy[49] to the evaluation of load balancer placement[64]. Scheduling of data-intensive applications[9]. Specifically, the results of[49] used the performance degradation caused by multi-tenancy as key performance indicator, as shown in Figure 11b. The figure shows different function adaptation approaches (i.e., All, GA, Skippy, and Vanilla) representing different scheduler configurations. Using the performance degradation model, simulation users can estimate the efficacy of their adaptation strategy to avoid this in multi-tenancy situations. In[64], we have evaluated different load balancer placements, thus focusing on the spatio-temporal context caused by varying request patterns arriving at the load balancer instances. We implemented a scheduling and scaling strategy to spawn load balancer instances dynamically and evaluated the function execution time with different configurations.

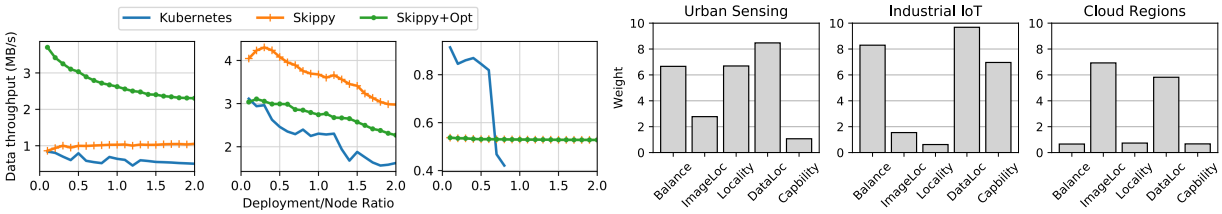
To summarize, *faas-sim* can evaluate different serverless adaptation strategies and, in combination with *Ether*'s topologies, considers spatio-temporal context. This is normally very difficult given the many parameters of the infrastructure and the environment that would have to be tuned to draw generalizable conclusions from experiments on real-world testbeds. Therefore, updating the models is advisable when adding new devices or functions.

4.1.3 | Resource planning

Resource planning helps platform designers and providers estimate their system's ability to handle different applications on a given infrastructure. It lets them plan before buying expensive hardware by simulating scenarios with varying hardware configurations. *faas-sim* has been used to conduct simulations on various infrastructure configurations that ranged from the cloud, industrial IoT, and Smart City infrastructures[9], and on a range of infrastructures with different compositions of available hardware ranging from small factor computers to cloud-like servers[49]. In both works, *faas-sim* was able to estimate the impact different hardware configurations had on the platform, and through the programmable scenario definition, various topologies were automatically created. Specifically in[49], the authors extended the simulation and Ether by generating random topologies based on probabilities (i.e., users can specify how many nodes have an Intel Xeon or Intel i5 CPU equipped, etc.). Moreover, in[9], we presented an initial evaluation of the Skippy scheduling system that is also used by default in *faas-sim* (see Section 3). Skippy introduces several soft constraints (scheduling functions) that are designed to help the scheduler make better container placement decisions in edge computing scenarios. Using *faas-sim*, we were able to show that the placements that Skippy makes lead to more scalable application deployments across different scenarios, indicated by the increased data throughput of the deployed application in Figure 12a.

4.1.4 | Co-simulation driven adaptations

In[9], we tackled the challenges schedulers in edge-cloud container systems face when deploying data-intensive workloads. Specifically, AI workloads were used as an example that must pull large AI models and training data from remote storage services over the network. These storage services are scattered across the system's infrastructure, and we have shown that pushing function



(a) Data throughput of different approaches and across different infrastructures

(b) Importance of Soft Constraints

Device	Arch	CPU	RAM	Accelerator	Storage
Xeon (GPU)	x86	4 x Core Xeon E-2224 @ 3.44 GHz	8 GB	Turing GPU - 6 GB	SSD
Intel NUC	x86	4 x Intel i5 @ 2.2 GHz	16 GB	N/A	NVME
RPi 3	arm32	4 x Cortex-A53 @ 1.4 GHz	1 GB	N/A	SD Card
RPi 4	arm32	4 x Cortex-A72 @ 1.5 GHz	1 GB	N/A	SD Card
RockPi	aarch64	2 x Cortex-A72, 4 x Cortex-A53	2 GB	N/A	SD Card
Coral DevBoard	aarch64	4 x Cortex-A53	1 GB	Google Edge TPU	eMMC
Jetson TX2	aarch64	4 x Cortex-A57 @ 2 Ghz	8 GB	256-core Pascal GPU	eMMC
Jetson Nano	aarch64	4 x Cortex-A57 @ 1.43 GHz	4 GB	128-core Maxwell GPU	SD Card
Jetson NX	aarch64	6 x Nvidia Carmel @ 1.9 GHz	8 GB	384-core Volta GPU 48 tensor cores	SD Card

Table 1 Device type specifications

Function	Jetson NX	Jetson Nano	Jetson TX2	Xeon (GPU)	Intel NUC	RockPi 4	RPi 4
Fio	13.77	19.64	4.31	1.14	1.12	21.99	27.81
Mobilenet Inf. TFlite	0.33	0.45	0.33	0.28	0.28	0.52	1.28
Python Pi	0.83	0.75	0.55	71.59	0.25	0.92	23.59
Resnet50 Inf. CPU	0.51	0.93	0.74	0.17	0.16	1.38	2.91
Resnet50 Inf. GPU	0.39	0.73	0.39	0.13	×	×	×
Resnet50 Preprocessing	6.08	7.95	6.39	2.66	2.53	7.65	19.5
Resnet50 Train CPU	×	×	×	×	197.45	×	×
Resnet50 Train GPU	142.0	847.17	228.12	32.13	×	×	×
Speech Inf. GPU	1.65	4.54	3.31	0.75	×	×	×
Speech Inf. TFlite	2.68	3.89	3.44	1.08	1.06	3.82	6.75
TF GPU	1.17	0.62	1.89	0.37	×	×	×

Table 2 Function traces included in *faas-sim*. Shows the mean FET in seconds over 100 requests.

execution closer to the data can alleviate network pressure and result in faster execution. We introduced weighted scheduling soft constraints for which we ran simulations to find optimal weights and could apply the weights obtained from the simulation in their scheduler. A good assignment of weights depends on the desired optimization goal and the cluster topology. We used *faas-sim* within an optimization algorithm to find a weight assignment that makes good trade-offs. Figure 12b shows the weight of the different soft constraints used in the scheduler to optimize average FET. Without the simulator, we would have had to execute the same experiments on a dedicated testing infrastructure but were able, due to *faas-sim*'s flexibility, enabled through the dynamic topology creation by Ether, to speed up the optimization process. This shows the ability of *faas-sim* to enable simulation-driven adaptations and that the domain model translates to real-world systems by using the simulation outcomes in the scheduler.

4.2 | *faas-sim* traces

We presented in Section 3.2.5 and Section 3.2.6 *faas-sim*'s resource and performance model, respectively. In the following, we want to highlight traces included in *faas-sim* and ready to use. The traces are split into performance (i.e., mean FET per invocation) and resources (i.e., resource usage per invocation). Table 1 shows our testbed, which contains a variety of typical edge devices. It includes modern embedded AI devices (i.e., Nvidia's Jetson series[9]), to resource-constrained Single Board

[9]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

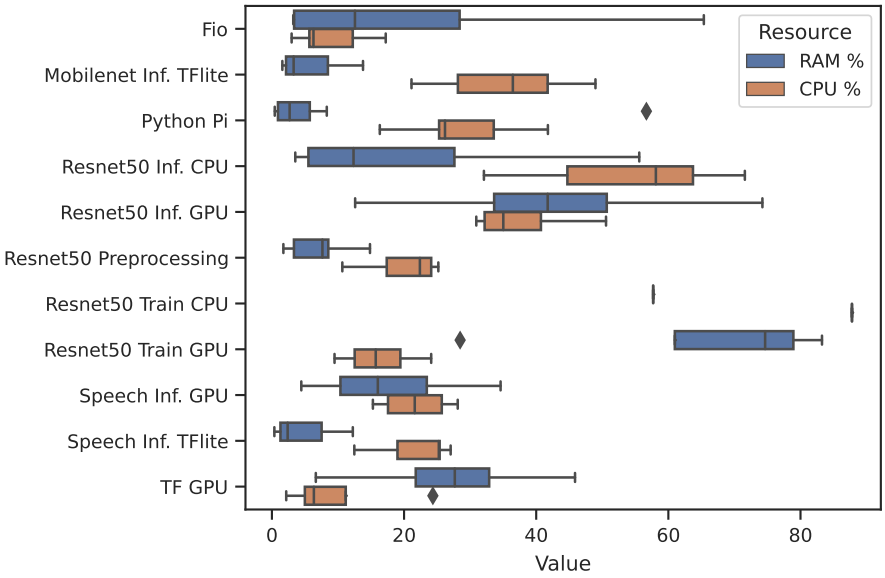


Figure 13 Mean CPU and RAM usage in % per function invocation across all devices.

Computers (i.e., Raspberry Pi, Rock Pi) and a PC with an Nvidia GPU (i.e., Xeon (GPU)). We collect traces from the following functions.

- AI-based object classification inference and training (i.e., Resnet50[65], Mobilenet[66])
- AI-based speech-to-text inference (i.e., DeepSpeech[67])
- Matrix multiplication using Tensorflow[68]
- A Python-based function that calculates Pi
- I/O heavy workload (i.e., Fio[10])

Tensorflow offers two implementations; one that targets inference on resource-constrained devices (i.e., TFlite[11]) and the regular TF2 distribution[12]. We perform inference and training using TF2 on GPUs and inference using TFLite. Table 2 shows the average FET in seconds over 100 requests for each device. Note that × means the function has not been profiled on that particular device. This can have two reasons: the execution mode is not supported (i.e., because the device does not have a hardware accelerator), or the execution failed (e.g., due to low resources). The average CPU and RAM usage per request across all devices are shown in Figure 19. This figure shows that resource usage vastly differs across devices. Note that the *Resnet50 Train CPU* has only been executed on the Intel NUC.

These traces enable our performance and resource modeling approaches.

4.3 | Network simulation evaluation

As we described in Section 3.2.7, *Ether*, the network model underlying *faas-sim*, uses a flow-based model to simulate data transfer between nodes, as opposed to a packet-level simulation like ns-3. This trades off fidelity (the ability to simulate network QoS like packet loss) against performance (running simulations faster).

To that end, we demonstrate our networking simulation a) produces similar results to ns-3 when simulating node-to-node transfer of data that is characteristic to data-intensive serverless edge computing workloads[9] (at least in simple scenarios), and b) accurately replicates real-world geo-distributed network scenarios such as the cloud testbed used in the evaluation of EMMA[27].

[10]https://fio.readthedocs.io/en/latest/fio_doc.html
[11]https://www.tensorflow.org/lite
[12]https://www.tensorflow.org/

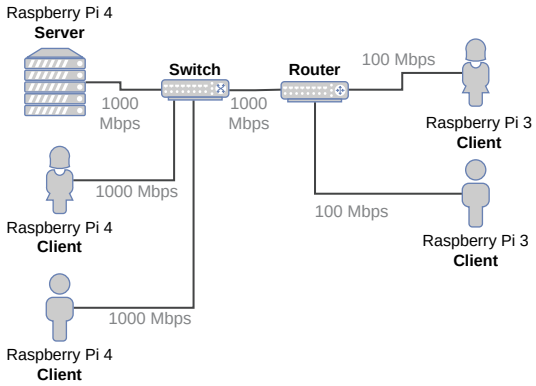


Figure 14 Testbed used for basic node-to-node data transfer experiments.

We should note that these experiments do not yet provide conclusive evidence that *any* edge computing networking scenario can be accurately modeled with *Ether* and *faas-sim* or that the reduced fidelity provides good enough results for all scenarios. As we argued in Section 3.2.7, and as we show in the evaluation, the extra fidelity may not be necessary to draw meaningful conclusions for the systems that *faas-sim* targets. That said, more evidence is needed to generalize this statement. Specifically, more experiments are needed to show how simulation accuracy behaves in scenarios where many flows compete for scarce networking resources for prolonged periods of time.

4.3.1 | Comparison to packet-level simulation

We first want to understand whether our flow-based network simulator can accurately replicate basic scenarios of node-to-node data transfer, which is typical in serverless edge computing systems like the ones presented in[69,9]. This includes transmitting individual images for ML inference, video files over Internet uplinks, or pulling container images from worker nodes.

Methodology

To that end, we run these basic scenarios on a real-world testbed with an emulated network topology. Then, we replicate the scenarios first on the baseline packet-level simulator ns-3 and compare it to running the experiments on our simulator.

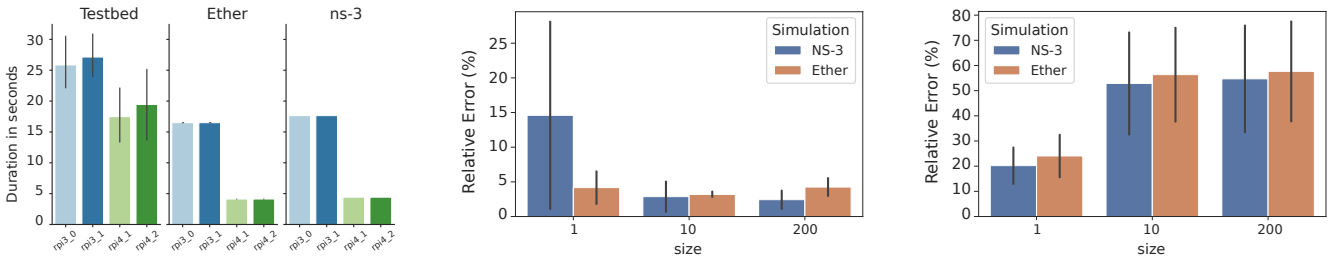
Figure 14 shows the testbed setup and also indicates which devices act as servers and clients. Our testbed consists of three Raspberry Pi 4, two Raspberry Pi 3, an unmanaged network switch, and one EdgeRouter X. We model the network topology in ns-3 and *Ether* and execute the experiments in each simulator and on the testbed. Transferring data via HTTP is one of the main use cases in our simulation, as described in Section 3.2.7, so our experiments build on these technologies. Specifically, the scenarios consist of an Apache HTTP server and multiple clients downloading files using the HTTP client *curl*. The file size varies between experiment runs and is based on real-world observations:

- **1MB**: the median size of a graphical image from a desktop web page (as of February 2023)[70].
- **10MB**: 90th percentile size of a video on the Internet[70].
- **200MB**: represents a *Docker* container image that contains a serverless function workload.

We perform two types of experiments we label *seq* and *par*. The *seq* experiment tests one client downloading a file from one server (i.e., one flow over one link). The *par* experiment tests the network behavior of multiple clients downloading the file in parallel from the same server (i.e., multiple flows over one link). We perform each experiment run with one or multiple clients downloading one file from the same server and repeat this procedure 400 times. However, due to ns-3's deterministic behavior, we only executed each experiment once.

Results

Figure 15b and Figure 15c show the relative error between the average download duration measured on the testbed and from the simulations. Specifically, we collect the duration of each download from each client, and the figures aggregate the errors across all four clients (i.e., two Raspberry Pi 3 and two Raspberry Pi 4). Figure 15b shows that the relative error of the *Seq* experiments



(a) *Par* network simulation experiments with 200 Megabyte file

(b) Relative to the ground truth in the *Seq* experiments.

(c) Relative to the ground truth in the *Par* experiments.

Figure 15 Results of basic data transfer scenarios on testbed, ns-3, and *Ether*.

is relatively low. ns-3's variance with the 1 Megabyte file is high because it could simulate the Raspberry Pi 3 client accurately (i.e., 1.06% error), while the simulation error of the Raspberry Pi 4 was 28%. In contrast, the experiments with 10 Megabyte file size are reversed, and the Raspberry Pi 4 simulation was more accurate. However, the error of *Ether* stays under 7% across file sizes.

Figure 15c shows the relative error of the *Par* experiments which shows that the relative error can be high with increasing file sizes. However, both simulators show similar errors, whereas NS-3 is more accurate than *Ether*. We think this high error rate in the parallel experiments is mainly caused due to the testbed's components (i.e., Raspberry Pis) and the fast network congestion. Both simulators do not model or consider the performance or hardware and, therefore, can not accurately reflect the low performance of the resource-constrained devices. Figure 15a shows that both simulators produce similar and steady results, but the durations measured on the testbed vary greatly.

4.3.2 | Replicating real-world experiments

Now that we have seen that basic networking scenarios produce good results, we want to demonstrate that our simulator can replicate more complex real-world scenarios representative of edge computing systems. To that end, we replicated an experiment we have previously performed on a real-world testbed when evaluating the elastic MQTT middleware system EMMA[27]. EMMA is tailored to geo-distributed scenarios and aims to reduce end-to-end latencies in pub/sub systems by leveraging edge resources for brokering messages. It uses a custom protocol to monitor the network distance between clients and brokers, essential to making EMMA automatically re-configure client-broker connections to optimize latencies. We later extended EMMA to auto-scale brokers based on edge resources based on demand in proximity[71]. Because both this monitoring protocol and pub/sub messaging in IoT are network-bound workloads, we consider EMMA a good candidate for evaluating our network simulation.

The specific experiment we replicate is described in detail in[27]. The left image in Figure 16 shows the evaluation testbed. It involves the deployment of clients and brokers in three different cloud regions, and the experiment creates both clients and brokers in different regions at specific times during the experiment. We measured the latencies of different MQTT topics that spanned different regions, shown in the center image of Figure 16. The various peaks and dips in latency come from clients and brokers leaving and entering the network. We then replicated this topology using *Ether*, and plugged in the measured Internet latencies as distributions into the simulator. Within a cloud region, we assumed high-bandwidth 10 GB/s LAN links. Then, we modeled the experiment as a *faas-sim* benchmark, which executed roughly the same events, and simulated clients produced similar workloads. The image on the right in Figure 16 shows the results from the same experiment that we modeled in *Ether* and *faas-sim*. A visual interpretation of the data shows that the general behavior of the system was modeled correctly. The x-axis is slightly misaligned because the experiment time was measured differently, and the latencies in the simulated results are more coarse-grained averages which is why the graphs are smoother. As discussed in[27], some latency peaks result from Java VM warmup and buffering, which the simulation cannot capture. This is valuable for researchers, as platform-specific performance aspects should be considered explicitly. Overall, however, we showed that the simulation provided good enough results compared to real-world experiments, so we could move forward in evaluating our auto-scaling approach in[71] using simulation data. Detailed results and the remaining simulation scenarios of scaling brokers can be found in[71].

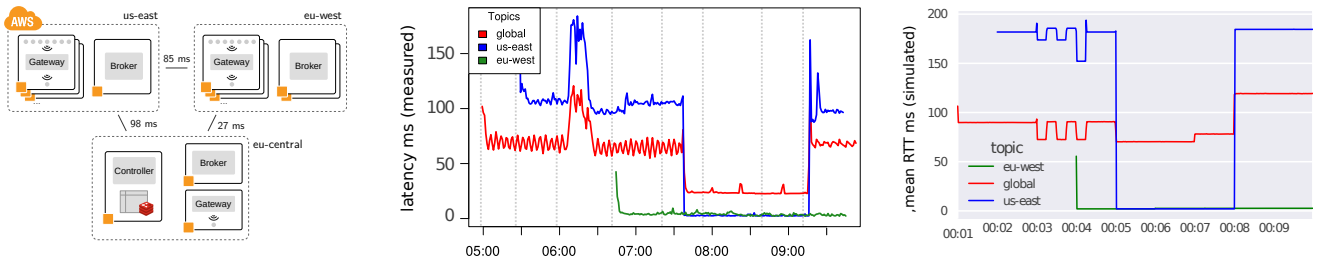


Figure 16 Real-world multi-region cloud testbed (left), evaluation results of EMMA on the testbed (center), and results from experiment replication in *Ether* (right).

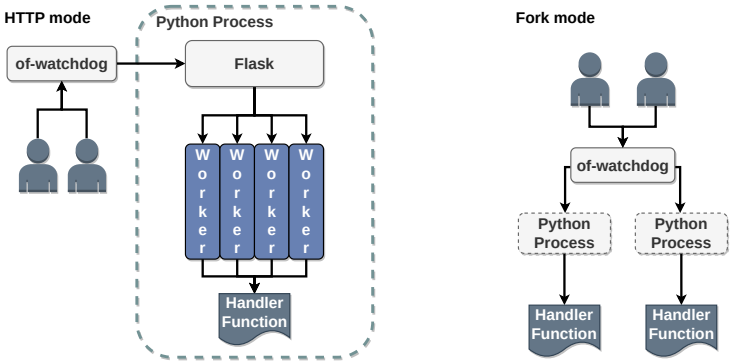


Figure 17 *of-watchdog* execution modes

4.4 | Flexible platform design

In the following, we show *Function Simulators* implemented in *faas-sim*. Specifically, they model OpenFaaS' reverse proxy (*of-watchdog*[13]), which acts as the base to implement functions in OpenFaaS. To implement a function in OpenFaaS, users implement a `handle` function that is called through a Go-based reverse proxy (i.e., *of-watchdog*). The execution mode of the reverse proxy determines how it invokes the `handle` function. We focus on two execution modes and assume the function is implemented in Python. Figure 17 shows the *HTTP* and *Fork* execution modes.

The *HTTP* execution mode starts a Flask HTTP server and re-directs each call to this server. The HTTP server internally uses a queueing mechanism with a pre-defined set of workers that call the *Handler Function*[72]. The *Fork* execution mode forks for each incoming request a new Python process which executes the *Handler Function*. The *HTTP* execution mode can cache expensive resources shared among the workers. The *Fork* mode is computationally more expensive and incurs higher latency because it starts a new Python process for each request. The following shows how these two modes are implemented in *faas-sim* as *Function Simulators*. The classes are `HTTPWatchdog` and `ForkingWatchdog` and extend the `Watchdog` class. The three (simplified) classes are shown in Figure 18. The `HTTPWatchdog` initializes during *setup* a SimPy queue and allows up to a user-defined number of concurrent requests. The *invocation* waits for a free worker, claims resources, executes the function, releases the resources, and frees the queue. Simulation users must implement the methods that model the function execution and resource usage. The `ForkingWatchdog` invokes the user's supplied methods. It is up to the simulation users to simulate the creation of the Python process accurately and also consider that function invocations may fail if there are too many because of resource exhaustion. However, this example shows the flexibility of *faas-sim*'s core simulation component, the *Function Simulator*, and its ability to mimic real-world behavior realistically.

[13] https://github.com/openfaas/of-watchdog

HTTPWatchdog

```
def setup(env):
    # create worker queue, with length of 4
    self.queue = simpy.Resource(env, capacity=4)
def invoke(env):
    # wait till worker is free
    token = self.queue.request()
    yield token

    # claim resources for function invocation
    yield from self.claim_resources(env)

    # execute request, method is abstract and
    # users must implement it
    yield from self.execute(env, req)


    # release resources
    yield from self.release_resources(env)

    # free worker
    self.queue.release(token)
```

Watchdog

```
def teardown(env, replica):
    while (self.inflight_requests() != 0):
        yield env.timeout(self.teardown_wait_interval)
    yield env.timeout(0)
```

ForkingWatchdog

```
def invoke(env):
   # claim resources for function invocation
    yield from self.claim_resources(env)

    # execute request, method is abstract and
    # users must implement it
    yield from self.execute(env, req)


    # release resources
    yield from self.release_resources(env)
```

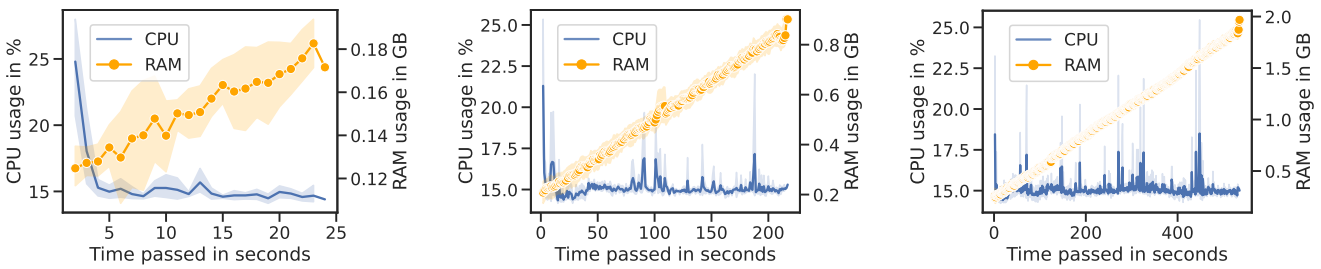
Figure 18 Classes implementing OpenFaaS' watchdogs.

4.5 | *faas-sim* resource usage

Lastly, we want to demonstrate that *faas-sim* is sufficiently performant to simulate long-running scenarios even on developer machines. Moreover, understanding the resource usage of *faas-sim* helps when using *faas-sim* as a co-simulator in a system to make runtime decisions.

To that end, we execute a series of experiments to evaluate the resource usage of the simulator in terms of CPU, memory, and execution run time. Specifically, we model a Smart City topology with 15 edge clusters and one cloud cluster. Each computing cluster has one client (i.e., Intel NUC) and multiple devices (i.e., Xeon, Jetson TX2, Nano, and NX). This allows us to observe the resource usage over time. Each client sends up to 100 requests per second with a constant workload, and we deploy 30 function replicas of the aforementioned *Resnet50 Inference CPU* function. However, the number of requests each client sends differs *100*, *1000*, and *2500*. The total requests sent across the system are 1500, 15000, and 37500, respectively. We simulate these scenarios using the traces (i.e., FET and resource usage) shown before and deactivate the *Resource Monitor* for these experiments as our system does not use the resource metrics gathered during the simulation. The simulations are started natively (i.e., using `Python`), and the test host has 32GB RAM and an i7 7700K@4.2 GHz with four cores (and eight threads).

Each scenario is executed five times, and Figure 19 shows the CPU and memory usage of the host. Before conducting the experiments, we measured the baseline consumption of the host and only show the isolated memory usage of the simulation. The CPU results contain other processes, but due to Python's single-threaded execution model, the simulation constantly uses 100% of one CPU core (i.e., 13% relative to all available threads).



(a) 100 requests per client. **(b)** 1000 requests per client. **(c)** 2500 requests per client.

Figure 19 CPU and RAM usage over three different experiments

Memory usage increases over time and can reach up to 2GB during the simulation, which lasts over 8 minutes. While this seems problematic, we show in the following that the resource usage is constant with few adjustments. Figure 20 shows the scenario in which each client sends 2500 requests. Contrary to the other experiment runs, we deactivated the logging framework (i.e., *Metrics*) and saw that the simulation's memory usage is much lower and grows slower. We also provide an optional logger

implementation that continuously writes the internal data structures to disk and removes the old ones, thus, avoiding the memory increase observed in the results.

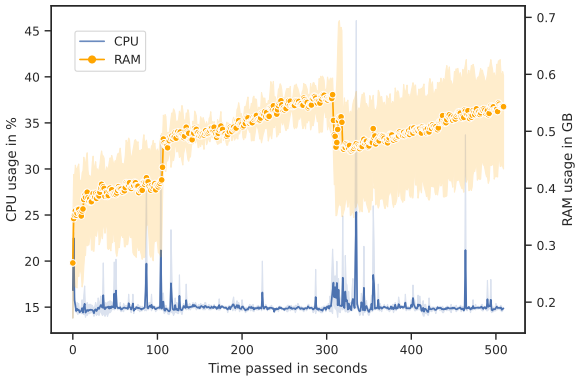


Figure 20 CPU and RAM usage with 2500 requests per client but no logging.

5 | RELATED WORK

5.1 | Specialized high fidelity simulators

Some simulators were implemented exclusively for evaluating a new adaptation technique[73]. Similarly, particular simulations aim at near-identical outcomes as the ground truth for specific system components. For example, Eismann et al. introduce *Sizeless* for predicting memory-related configurations to cut client costs and improve provider resource efficiency[74]. Later, Lin et al. propose formally modeling FaaS applications to aid the implementation of an accurate cost model[73].

The downside is that specialized simulators are costly and more dependent on their intended platforms and applications. Worse, in work where the simulator is not the contribution and is only a necessary side product, it may not generalize well outside its intended scope. Consequently, we provide default implementations that simulation users must populate with realistic traces to cover various applications and platforms. Since the components are interchangeable, it is possible to incorporate arbitrary methods for performance and resource modeling.

Importantly, this design decision allows *faas-sim* to retain relevancy as the state-of-the-art evolves and promotes a streamlined research directive.

5.2 | General edge-cloud simulations

Sphere[75] is an extension of the SCORE[76] simulator. Sphere aims to enable the implementation of topology- and orchestration models for edge computing and fast-paced simulation. They evaluate different platform architectures and cover many aspects of the edge-cloud continuum. However, a drawback they state to solve in future work is that their code needs to be more easily extendible. In contrast, *faas-sim* simulation users are encouraged to extend and modify the simulation. Wang et al.[77] present their open-source simulation platform SimEdgeIntel. The authors want to offer a simulation tool with a low entry barrier to implement new resource management strategies for Edge Intelligence due to the cross-platform and cross-language support. Their system model focuses on device-to-device (D2D) communication and considers network handovers in the simulation. Alwasel et al.[19] present IotSim-Osmosis, an extension of CloudSim to model and simulate IoT applications across the edge-cloud continuum. They envision a multi-layered architecture comprising IoT, edge, cloud, and software-defined wide area networks (SD-WAN). Their simulation supports a variety of communication protocols. The tool also supports custom policies to control components (i.e., task scheduling and routing). IotSim-Edge[18] also extends CloudSim and provides entities to model different edge-cloud aspects, such as mobility, energy consumption and IoT protocols (e.g., CoAP) and network protocols (e.g., 4G). The simulation scenarios heavily focus on IoT applications (e.g., stream processing) and offer a GUI to create scenarios. IotSim-SDWAN[78]

focuses on providing accurate network simulations for edge-cloud systems. Specifically, they enable software-defined wide area networks (SDWAN) modeling and introduce a system model to replicate edge-cloud networks realistically.

The simulators above aim to simulate different application scenarios and their concepts. However, *faas-sim* can integrate concepts of them, such as focusing on IoT protocols. Therefore, the focus of this work is dedicated serverless simulations.

5.3 | Serverless simulations

To compare existing serverless simulators, Table 3 summarizes whether and to what extent they support the six use cases identified in Section 2.2. A ✓, ∼, and × indicate a full, partial, or no coverage, respectively. We consider a feature partially covered when the work only provides rudimentary support (e.g., no support for fine-grained customization) or when it seems technically possible to cover the feature through manual extension.

SimLess by Ristov et al.[21] introduces abstractions for performing numerous experiments in various conditions without requiring extensive parameter configurations. Although SimLess considers heterogeneous environments and supports performance modeling for client programmers, it does not provide an interface to implement execution models for function invocation. Additionally, it does not support resource planning and serverless adaptations to facilitate research on serverless platforms.

Mastenbroek et al.[15] re-design and extend *OpenDC*[79] to support serverless computing. The low-level nature of OpenDC naturally provides interchangeable interfaces but incurs higher complexity. SimLess and OpenDC contrast each other. While SimLess focuses on client programmers and trades reduced configuration complexity with less flexibility, OpenDC focuses on platform providers with powerful modeling tools for highly configurable simulations of data centers. Additionally, the former focuses on client programmers, and the latter focuses on platform providers with powerful modeling tools for highly configurable simulation of data centers. FaaS-sim provides a middle ground between SimLess and OpenDC to support client programmers and platform providers by providing interchangeable high-level abstractions.

Jeon. et al., introduce *DFaasCloud*[17] as an extension of CloudSim[46]. We consider their work a proof-of-concept since their serverless support is rudimentary, and simulating complex workloads is infeasible. Moreover, the official repository did not see any contributions after its initial release. *SimFaaS* by Mahmoudi and Khazae[20] aids client programmers and platform providers estimate costs. Nevertheless, other than extensively covering cost prediction, it has several limitations. First, it explicitly only supports the scale-pre-request pattern, i.e., simulations with resource-based or concurrency value scaling are impossible. Second, focus on existing public cloud providers. In contrast, *faas-sim* is not limited to existing cloud platform providers by supporting simulations for hybrid edge-cloud environments and allowing simulation users to implement arbitrary serverless function adaptation techniques.

Lastly, we considered the work by Bhardwaj et al.[80]. and Manner et al.[81]. The former introduces *KubeKlone*[80], a simulation-based framework based on uqSim[82] focusing on a simulator to train AI-based operations for microservice deployments. The latter introduces a simulation tool for client programmers to find suitable configurations by quickly permuting various parameters. Nevertheless, they do not associate any open-source repository with their work.

Table 3 Comparing simulators supporting serverless abstractions

	SimLess	OpenDC 2.0	DFaaSCloud	SimFaaS	*faas-sim*
Application Performance Estimation	✓	✓	✓	✓	✓
Performance Modeling	∼	✓	∼	∼	✓
Resource Planning	×	✓	✓	×	✓
First-class Serverless Adaptations	×	∼	∼	×	✓
Co-Simulation Driven Adaptations	×	∼	×	×	✓
Edge-Cloud Continuum	×	×	✓	×	✓
Simulation Configurability	Medium	High	Low	Low	Low-High
Topology Generation	×	UI	JSON, UI	×	Code[13]

To summarize, the main differences between *faas-sim* and the simulators presented in Section 5.2 and Section 5.3 are:

- *Simulation focus*: our system model resembles serverless computing and allows users to evaluate serverless edge platforms. The previous works do not always replicate a specific platform architecture (e.g., general edge-cloud simulations).
- *System models*: *faas-sim* aims to provide a playground for researchers and practitioners, allowing them to introduce new models based on our foundation and does not expect a specific resource model.
- *Application simulation*: to the best of our knowledge, our *Function Simulator* approach is a novel way of modeling applications and gives simulation users great flexibility. Others only allow users to specify the number of instructions required (e.g., CloudSim-based simulations) or only need the application duration (e.g., SimFaaS [20]).
- *Serverless Function Adaptations*, including request routing, scaling, and scheduling of function instances, are only modeled as first-class citizens in *faas-sim*. Inevitably, this also impacts the ability of the simulators to be used in co-simulation driven adaptations. Specifically, SimFaaS and SimLess do not allow simulation users to specify the function adaptation implementations, DFaaSCloud allows custom scheduler adaptations but does not outline customization of the scaling component, OpenDC 2.0 appears to only allow users to select the function scheduler and request routing implementations. However, not the autoscaler implementation [83].
- *Co-simulation driven adaptations* we consider OpenDC 2.0 and DFaaSCloud only partially fulfilling this criterion as they allow us to inject custom adaptations to some extent. Contrary, *faas-sim* not only enables users to customize the three major adaptations, but it also uses the *skippy-scheduler* internally that works the same way the default Kubernetes scheduler does, thus, allowing to easily optimize scheduling approaches and apply them in the real world.
- *Simulation Configurability* is based on the parameters that simulation users can tweak. For example, SimFaaS and DFaaSCloud offer only high-level parameters that describe the general characteristics of the deployed functions (e.g., arrival rate, execution duration, etc.). SimLess takes a different approach by observing the execution of functions on public cloud platforms and using the measurements to feed the simulation. OpenDC 2.0 has a high level of configurability as users design the data centers from the ground up and can tweak any aspect, which holds for *faas-sim* as well as the topology creation considers hardware characteristics but leaves it up to the simulation users to incorporate these details in the simulation. Therefore, *faas-sim* also allows performing simulations without much configuration similar to DFaaSCloud and SimFaaS. Moreover, OpenDC 2.0 tries to model data centers on a very fine-grained level while *faas-sim* tries to model serverless edge platforms.
- *faas-sim* is the only one with first-class citizen support for generating topologies based on code. OpenDC 2.0 gives users an interface where they can manually create the layout of a data center, while DFaaSCloud offers an existing GUI and accepts JSON as input. SimFaaS and SimLess both do not allow any topology modeling.

6 | CONCLUSION

Serverless edge computing is an emerging platform paradigm that extends the promising serverless computing paradigm. These paradigms abstract the infrastructure from developers and promise cost-efficient deployment through autonomous management. However, serverless computing platforms are cloud-centric and have yet to adapt to the emerging edge-cloud continuum. Combining all resources and enabling function execution across the continuum requires novel platform ideas that move away from the cloud-centric design and distributed and decentralized control and data plane. We identified two main tasks each serverless edge computing platform has to implement: platform architecture and serverless function adaptations. The edge-cloud continuum deepens the complexity of implementing them and differentiates them from cloud-centric platforms. To this end, we presented *faas-sim*, an open-source trace-driven discrete-event Python simulator that can support researchers and practitioners in developing and evaluating serverless edge computing systems. Our evaluation demonstrated that *faas-sim* is usable in a wide range of scenarios out of the box while allowing users to extend and modify it to their use case. We have shown that our networking simulation provides similar results to state-of-the-art packet-level simulators such as ns-3 and in an evaluation using a cloud testbed setup. However, these results do not necessarily generalize to more complex scenarios or that any scenario can be accurately modeled. More experiments are needed to show the accuracy of the network simulation in complex scenarios. Our trace-driven resource modeling approach for simulating workloads provides a flexible and accurate framework for modeling various workloads and heterogeneous computing platforms. While we have evaluated it using the herein presented functions,

our single-tenancy and multi-tenancy approaches should be re-evaluated when adding new devices and functions or including new simulation aspects. *faas-sim* has been used successfully in several publications to evaluate different systems aspects, from scheduling performance comparisons to simulating large-scale distributed edge computing infrastructure to using *faas-sim* as co-simulator to optimize system parameters at runtime. *faas-sim* is part of the *Edgerun* project, an ecosystem of tools to help advance the research of edge-cloud system challenges. The simulator users can use these tools to automatically integrate their testbeds and profiling experiments to gather trace data for their simulation scenarios. In future work, we want to explore embedding *faas-sim* into the control loop of *operating* serverless systems. With a high-performant trace-driven simulator, we can incrementally build a digital twin of the real infrastructure and use trace data from real workload execution to refine the simulation model at runtime. The simulator can then be used as an engine to evaluate placement or scaling decisions stochastically.

Acknowledgements

The authors want to thank their following colleagues and students who have contributed to this publication by running experiments for their theses, some of which were presented here, and contributing code to *faas-sim* and *Ether*: Andreas Bruckner, Cynthia Marcelino, Theresa Müller, Jacob Palecek, Paul Prüller, Alexander Rashed, Alexander Woda. We also thank Alexander Knoll who has helped with setting up various testbeds that were presented throughout the paper. Moreover, we want to thank the reviewers that provided valuable input that helped us improve the paper.

References

1. Aslanpour MS, Toosi AN, Cicconetti C, et al. Serverless Edge Computing: Vision and Challenges. In: ACSW '21. Association for Computing Machinery. ; 2021; New York, NY, USA

2. Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* 2019. doi: 10.48550/ARXIV.1902.03383

3. Raith P, Nastic S, Dustdar S. Serverless Edge Computing—Where We Are and What Lies Ahead. *IEEE Internet Computing* 2023; 27(3): 50–64.

4. Xie R, Tang Q, Qiao S, Zhu H, Yu FR, Huang T. When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues. *IEEE Wireless Communications* 2021; 28(5): 126-133. doi: 10.1109/MWC.001.2000466

5. Pfandzelter T, Bermbach D. tinyfaas: A lightweight faas platform for edge environments. In: 2020 IEEE International Conference on Fog Computing (ICFC). IEEE. ; 2020: 17–24.

6. Tamiru MA, Pierre G, Tordsson J, Elmroth E. mck8s: An orchestration platform for geo-distributed multi-cluster environments. In: 2021 International Conference on Computer Communications and Networks (ICCCN). IEEE; 2021: 1-10

7. Lähderanta T, Leppänen T, Ruha L, et al. Edge computing server placement with capacitated location allocation. *Journal of Parallel and Distributed Computing* 2021; 153: 130-149. doi: https://doi.org/10.1016/j.jpdc.2021.03.007

8. Wolski R, Krintz C, Bakir F, George G, Lin WT. Cspot: Portable, multi-scale functions-as-a-service for iot. In: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing. ; 2019: 236–249.

9. Rausch T, Rashed A, Dustdar S. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems* 2021; 114: 259-271. doi: https://doi.org/10.1016/j.future.2020.07.017

10. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking Cloud Serving Systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. Association for Computing Machinery; 2010; New York, NY, USA: 143–154

11. Duplyakin D, Ricci R, Maricq A, et al. The Design and Operation of CloudLab. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). USENIX Association; 2019; Renton, WA: 1–14.

12. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-Scale Cluster Management at Google with Borg. In: EuroSys '15. Association for Computing Machinery; 2015; New York, NY, USA

13. Rausch T, Lachner C, Frangoudis PA, Raith P, Dustdar S. Synthesizing Plausible Infrastructure Configurations for Evaluating Edge Computing Systems. In: 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). USENIX Association; 2020.

14. Raith P, Rausch T, Prüller P, Furutanpey A, Dustdar S. An End-to-End Framework for Benchmarking Edge-Cloud Cluster Management Techniques. In: 2022 IEEE International Conference on Cloud Engineering (IC2E). IEEE. ; 2022.

15. Mastenbroek F, Andreadis G, Jounaid S, et al. OpenDC 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE. ; 2021: 455–464.

16. Raith P, Rausch T, Dustdar S, Rossi F, Cardellini V, Ranjan R. Mobility-Aware Serverless Function Adaptations Across the Edge-Cloud Continuum. In: 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC). IEEE; 2022: 123-132

17. Jeon H, Cho C, Shin S, Yoon S. A CloudSim-extension for simulating distributed functions-as-a-service. In: 2019 20th International Conference on parallel and distributed computing, applications and technologies (PDCAT). IEEE. ; 2019: 386–391.

18. Jha DN, Alwasel K, Alshoshan A, et al. IoTSim-Edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments. *Software: Practice and Experience* 2020; 50(6): 844-867. doi: https://doi.org/10.1002/spe.2787

19. Alwasel K, Jha DN, Habeeb F, et al. IoTSim-Osmosis: A framework for modeling and simulating IoT applications over an edge-cloud continuum. *Journal of Systems Architecture* 2021; 116: 101956. doi: https://doi.org/10.1016/j.sysarc.2020.101956

20. Mahmoudi N, Khazaei H. SimFaaS: A Performance Simulator for Serverless Computing Platforms. In: Helfert M, Ferguson D, Pahl C., eds. *Proceedings of the 11th International Conference on Cloud Computing and Services Science, CLOSER 2021, Online Streaming, April 28-30, 2021*SCITEPRESS; 2021: 23–33

21. Ristov S, Hautz M, Hollaus C, Prodan R. SimLess: Simulate Serverless Workflows and Their Twins and Siblings in Federated FaaS. In: Proceedings of the 13th Symposium on Cloud Computing. Association for Computing Machinery; 2022; New York, NY, USA: 323–339

22. Sonmez C, Ozgovde A, Ersoy C. EdgeCloudSim: An Environment for Performance Evaluation of Edge Computing Systems. In: 2017 Second International Conference on Fog and Mobile Edge Computing. IEEE. ; 2017: 39–44.

23. Gupta H, Vahid Dastjerdi A, Ghosh SK, Buyya R. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 2017; 47(9): 1275-1296. doi: https://doi.org/10.1002/spe.2509

24. Matloff N. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2008; 2(2009): 1–33.

25. OpenFaaS . OpenFaaS. 2016. https://www.openfaas.com/. Accessed February 28, 2023.

26. Burckhardt S, Gillum C, Justo D, Kallas K, McMahon C, Meiklejohn CS. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 2021; 5(OOPSLA). doi: 10.1145/3485510

27. Rausch T, Nastic S, Dustdar S. Emma: Distributed qos-aware mqtt middleware for edge computing applications. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE. ; 2018: 191–197.

28. Yao X, Chen N, Yuan X, Ou P. Performance optimization of serverless edge computing function offloading based on deep reinforcement learning. *Future Generation Computer Systems* 2023; 139: 74-86. doi: https://doi.org/10.1016/j.future.2022.09.009

29. Rodriguez MA, Buyya R. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience* 2019; 49(5): 698-719. doi: https://doi.org/10.1002/spe.2660

30. Deng S, Zhao H, Fang W, Yin J, Dustdar S, Zomaya AY. Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence. *IEEE Internet of Things Journal* 2020; 7(8): 7457-7469. doi: 10.1109/JIOT.2020.2984887

31. Raith P, Dustdar S. Edge Intelligence as a Service. In: 2021 IEEE International Conference on Services Computing (SCC). IEEE. ; 2021: 252–262.

32. Knative . Knative. 2018. https://knative.dev/docs/. Accessed February 28, 2023.

33. Fission . Fission. 2016. https://fission.io/. Accessed February 28, 2023.

34. IBM . OpenWhisk. 2016. https://openwhisk.apache.org/ Accessed February 28, 2023.

35. Amazon . AWS Lambda. 2014. https://aws.amazon.com/lambda/. Accessed February 28, 2023.

36. Baresi L, Hu DYX, Quattrocchi G, Terracciano L. NEPTUNE: Network- and GPU-Aware Management of Serverless Functions at the Edge. In: Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems. Association for Computing Machinery; 2022; New York, NY, USA: 144–155

37. Crankshaw D, Sela GE, Mo X, et al. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In: Proceedings of the 11th ACM Symposium on Cloud Computing. Association for Computing Machinery; 2020; New York, NY, USA: 477–491

38. Shafiei H, Khonsari A, Mousavi P. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Surveys* 2022; 54(11s). doi: 10.1145/3510611

39. Rossi F, Nardelli M, Cardellini V. Horizontal and vertical scaling of container-based applications using reinforcement learning. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE. ; 2019: 329–338.

40. Mayer R, Graser L, Gupta H, Saurez E, Ramachandran U. EmuFog: Extensible and Scalable Emulation of Large-scale Fog Computing Infrastructures. In: 2017 IEEE Fog World Congress. IEEE. ; 2017: 1–6.

41. Zhong Z, Xu M, Rodriguez MA, Xu C, Buyya R. Machine Learning-Based Orchestration of Containers: A Taxonomy and Future Directions. *ACM Computing Surveys* 2022; 54(10s). doi: 10.1145/3510415

42. Mampage A, Karunasekera S, Buyya R. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Computing Surveys* 2022; 54(11s). doi: 10.1145/3510412

43. Tuli S, Poojara SR, Srirama SN, Casale G, Jennings NR. COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog Computing Environments. *IEEE Transactions on Parallel and Distributed Systems* 2022; 33(1): 101-116. doi: 10.1109/TPDS.2021.3087349

44. Rossi F, Cardellini V, Lo Presti F, Nardelli M. Geo-distributed efficient deployment of containers with Kubernetes. *Computer Communications* 2020; 159: 161-174. doi: 10.1016/j.comcom.2020.04.061

45. Kubernetes . Kubernetes. 2014. https://kubernetes.io/. Accessed February 28, 2023.

46. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience* 2011; 41(1): 23-50. doi: 10.1002/spe.995

47. Markus A, Kertesz A. A survey and taxonomy of simulation environments modelling fog computing. *Simulation Modelling Practice and Theory* 2020; 101: 102042. doi: 10.1016/j.simpat.2019.102042

48. Rausch T, Avasalcai C, Dustdar S. Portable energy-aware cluster-based edge computers. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC). IEEE. ; 2018: 260–272.

49. Raith PA. Container scheduling on heterogeneous clusters using machine learning-based workload characterization. 2021. Masterthesis. TU Wien.

50. Rausch T, Hummer W, Muthusamy V. PipeSim: Trace-driven simulation of large-scale AI operations platforms. *arXiv preprint arXiv:2006.12587* 2020. doi: 10.48550/ARXIV.2006.12587

51. Henderson TR, Lacage M, Riley GF, Dowell C, Kopena J. Network simulations with the ns-3 simulator. In: SIGCOMM'08 demonstration. Association for Computing Machinery; 2008: 527.

52. Chang X. Network Simulations with OPNET. In: Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1. Association for Computing Machinery; 1999; New York, NY, USA: 307–314

53. Bertsekas D, Gallager R. *Data Networks*. Englewood Cliffs, NJ, USA: Prentice Hall. second ed. 1996.

54. Massoulié L, Roberts J. Bandwidth sharing: objectives and algorithms. In: IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320). IEEE. ; 1999: 1395-1403 vol.3

55. Kelly F. Charging and rate control for elastic traffic. *European Transactions on Telecommunications* 1997; 8(1): 33-37. doi: 10.1002/ett.4460080106

56. Chiu DM, Jain R. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems* 1989; 17(1): 1-14. doi: 10.1016/0169-7552(89)90019-6

57. Morris R. TCP behavior with many flows. In: Proceedings 1997 International Conference on Network Protocols. IEEE. ; 1997: 205-211

58. Catlett CE, Beckman PH, Sankaran R, Galvin KK. Array of Things: A Scientific Research Instrument in the Public Way: Platform Design and Early Lessons Learned. In: SCOPE '17. Association for Computing Machinery; 2017; New York, NY, USA: 26–33

59. 2018 Yellow Taxi Trip Data. 2018. Available at https://data.cityofnewyork.us/Transportation/2018-Yellow-Taxi-Trip-Data/t29m-gskq.

60. Kolosov O, Yadgar G, Maheshwari S, Soljanin E. Benchmarking in The Dark: On the Absence of Comprehensive Edge Datasets. In: 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). USENIX Association; 2020.

61. Rausch T. *A Distributed Compute Fabric for Edge Intelligence*. PhD thesis. TU Wien, 2021.

62. Riley GF, Henderson TR. The ns-3 Network Simulator. *Modeling and Tools for Network Simulation* 2010: 15–34. doi: 10.1007/978-3-642-12331-3_2

63. Olson RS, Moore JH. TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. *Automated Machine Learning: Methods, Systems, Challenges* 2019: 151–160. doi: 10.1007/978-3-030-05318-5_8

64. Palecek J. Improving Serverless Edge Computing for Network Bound Workloads. 2022. Master Thesis. TU Wien.

65. He K, Zhang X, Ren S, Sun J. Deep Residual Learning for Image Recognition. 2015. arXiv https://arxiv.org/abs/1512.03385.doi: 10.48550/ARXIV.1512.03385

66. Howard AG, Zhu M, Chen B, et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* 2017. doi: 10.48550/ARXIV.1704.04861

67. Hannun A, Case C, Casper J, et al. Deep Speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* 2014. doi: 10.48550/ARXIV.1412.5567

68. Abadi M, Barham P, Chen J, et al. TensorFlow: A System for Large-Scale Machine Learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association; 2016; Savannah, GA: 265–283.

69. Rausch T, Hummer W, Muthusamy V, Rashed A, Dustdar S. Towards a Serverless Platform for Edge AI. In: 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19). USENIX Association; 2019; Renton, WA.

70. HTTP Archive: State of the Web. https://httparchive.org/reports/state-of-the-web. Accessed February 27, 2023.

71. Bruckner A. Self-adaptive distributed MQTT middleware for edge computing applications. 2022. Master Thesis. TU Wien.

72. Balla D, Maliosz M, Simon C. Open Source FaaS Performance Aspects. In: 2020 43rd International Conference on Telecommunications and Signal Processing (TSP). IEEE. ; 2020: 358-364

73. Lin C, Mahmoudi N, Fan C, Khazaei H. Fine-Grained Performance and Cost Modeling and Optimization for FaaS Applications. *IEEE Transactions on Parallel and Distributed Systems* 2023; 34(1): 180–194. doi: 10.1109/TPDS.2022.3214783

74. Eismann S, Bui L, Grohmann J, Abad C, Herbst N, Kounev S. Sizeless: Predicting the Optimal Size of Serverless Functions. In: Proceedings of the 22nd International Middleware Conference. Association for Computing Machinery; 2021; New York, NY, USA: 248–259

75. Fernández-Cerero D, Fernández-Montes A, Javier Ortega F, Jakóbik A, Widlak A. Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption. *Simulation Modelling Practice and Theory* 2020; 101: 101966. Modeling and Simulation of Fog Computingdoi: https://doi.org/10.1016/j.simpat.2019.101966

76. Fernández-Cerero D, Fernández-Montes A, Jakóbik A, Kołodziej J, Toro M. SCORE: Simulator for cloud optimization of resources and energy consumption. *Simulation Modelling Practice and Theory* 2018; 82: 160-173. doi: https://doi.org/10.1016/j.simpat.2018.01.004

77. Wang C, Li R, Li W, Qiu C, Wang X. SimEdgeIntel: A open-source simulation platform for resource management in edge intelligence. *Journal of Systems Architecture* 2021; 115: 102016. doi: https://doi.org/10.1016/j.sysarc.2021.102016

78. Alwasel K, Jha DN, Hernandez E, et al. IoTSim-SDWAN: A simulation framework for interconnecting distributed datacenters over Software-Defined Wide Area Network (SD-WAN). *Journal of Parallel and Distributed Computing* 2020; 143: 17-35. doi: 10.1016/j.jpdc.2020.04.006

79. Iosup A, Andreadis G, Van Beek V, et al. The OpenDC vision: Towards collaborative datacenter simulation and exploration for everybody. In: 2017 16th International Symposium on Parallel and Distributed Computing (ISPDC). IEEE. ; 2017: 85–94.

80. Bhardwaj A, Benson TA. KubeKlone: A Digital Twin for Simulating Edge and Cloud Microservices. In: Asia-Pacific Workshop on Networking (APNet 2022). Association for Computing Machinery. ; 2022: 7.

81. Manner J, Endreß M, Böhm S, Wirtz G. Optimizing Cloud Function Configuration via Local Simulations. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). IEEE; 2021: 168-178

82. Zhang Y, Gan Y, Delimitrou C. uqSim: Scalable and Validated Simulation of Cloud Microservices. *arXiv preprint arXiv:1911.02122* 2019.

83. Sallo DH, Kecskemeti G. Towards Generating Realistic Trace for Simulating Functions-as-a-Service. In: Chaves R, B. Heras D, Ilic A, et al., eds. *Euro-Par 2021: Parallel Processing Workshops*. vol 13098. Springer International Publishing; 2022; Cham: 428–439

How to cite this article: P. Raith, T. Rausch, A. Furutanpey, and S. Dustdar (2023), *faas-sim*: A Trace-Driven Simulation Framework for Serverless Edge Computing Platforms, *Q.J.R. Meteorol. Soc.*, *2017;00:1–6*.