Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum

Gabriele Russo Russo*, Tiziana Mannucci*, Valeria Cardellini* and Francesco Lo Presti*
* *University of Rome Tor Vergata, Rome 00133, Italy*
Email: {russo.russo, cardellini}@ing.uniroma2.it, tiziana.mannucci@alumni.uniroma2.eu, lopresti@info.uniroma2.it

2023 IEEE International Conference on Pervasive Computing and Communications (PerCom) | 978-1-6654-5378-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/PERCOM56429.2023.10099372

***Abstract*—As the Function-as-a-Service (FaaS) paradigm enjoys growing popularity within Cloud-based systems, there is increasing interest in moving serverless functions towards the Edge, to better support geo-distributed and pervasive applications. However, enjoying both the reduced latency of Edge and the scalability of FaaS requires new architectures and implementations to cope with typical Edge challenges (e.g., nodes with limited computational capacity). While first solutions have been proposed for Edge-based FaaS, including light function sandboxing techniques, we lack a platform with the ability to span both Edge and Cloud and adaptively exploit both.**

In this paper, we present Serverledge, a FaaS platform designed for the Edge-to-Cloud continuum. Serverledge adopts a decentralized architecture, where function invocation requests can be fully served within Edge nodes. To cope with load peaks, Serverledge also supports vertical (i.e., from Edge to Cloud) and horizontal (i.e., among Edge nodes) computation offloading. Our evaluation shows that Serverledge outperforms Apache OpenWhisk in an Edge-like scenario and has competitive performance with state-of-the-art frameworks optimized for the Edge, with the advantage of built-in support for vertical and horizontal offloading.

***Index Terms*—serverless, edge computing, offloading**

I. Introduction

Function-as-a-Service (FaaS) allows users to deploy units of computation, defined as *functions*, to be executed in response to events (e.g., HTTP triggers) in a serverless fashion [1], with the underlying platform taking care of most the operational issues, including resource provisioning and scaling. The fine-grained pricing models and the seamless scalability of FaaS have boosted its popularity for the last years, with all the major Cloud providers now offering FaaS platforms (e.g., AWS Lambda, Google Cloud Functions, Azure Functions).

While Cloud-based applications increasingly adopt the FaaS paradigm, it is natural to wonder whether and how FaaS can be exploited by pervasive applications deployed at the edge of the network [2], [3]. Domains like Industrial Internet-of-Things, smart healthcare, and augmented/virtual reality, usually require their inputs to be processed with very low latency [4] and, thus, their requirements can hardily be satisfied by transferring data to remote Cloud data centers and back.

In addition to public Cloud platforms, various open-source FaaS frameworks are available nowadays (e.g., Apache OpenWhisk and OpenFaaS), but, unfortunately, they are mostly designed to run in Cloud or clustered environments. Key limiting factors towards their seamless adoption at the Edge include (i) frequent use of centralized schedulers or gateway components, which introduce latency in geo-distributed settings, and (ii) memory-demanding function sandboxes, usually based on software containers.

The research community has started investigating solutions to better support FaaS at the Edge and novel frameworks have been recently presented that better suit Edge environments, often exploiting lightweight function sandboxing mechanisms instead of OS-level virtualization (e.g., Faasm [5] and Sledge [6], which rely on software-fault isolation). However, these solutions either work within single Edge nodes (e.g., [6], [7]), or scale over multiple nodes without considering geographical distribution (e.g., [5]). Other works (e.g., [8]–[10]) study architectures and algorithms for function placement and load distribution in decentralized FaaS systems, but relying on the existing Cloud-oriented frameworks for actual function execution, possibly incurring the issues mentioned above when running at the Edge.

In this paper we present Serverledge, a new FaaS system that aims to fill the gap between Edge and Cloud and provides a flexible and extensible framework for FaaS in geographically distributed environments. Serverledge adopts a decentralized architecture, with nodes organized into Edge zones and Cloud regions based on their location. Every Serverledge node, being it at the Edge or in the Cloud, is able to schedule and execute invocation requests with minimal or no interaction with remote nodes, keeping latency as low as possible. To cope with load peaks, Serverledge also supports vertical (i.e., from Edge to Cloud) and horizontal (i.e., among Edge nodes) computation offloading, allowing nodes to forward invocation requests that cannot be served locally. Serverledge supports functions written in multiple programming languages, currently relying on simple-yet-popular Docker containers for isolated function execution.

We design Serverledge with flexibility in mind, aiming to contribute an easy-to-extend prototype to the research community, for future investigations on FaaS at the Edge. Our key contributions can be summarized as follows:

- We design a decentralized FaaS framework for Edge-Cloud environments, where Edge nodes schedule and execute functions with minimal interaction with the rest of the system. To extend their serving capacity, nodes can offload requests to neighbor Edge nodes or to the Cloud.
- We implement Serverledge in Go, exploiting Docker containers for isolated function execution.

- We perform an experimental comparison to assess the relative performance of Serverledge against state-of-the-art FaaS platforms (namely, OpenWhisk, Faasm [5] and tinyFaaS [7]). We also evaluate the benefits of the offloading mechanisms integrated in Serverledge and demonstrate their flexibility by means of a proof-of-concept QoS-aware offloading policy.

The remainder of the paper is organized as follows. We review related work in Sec. II. An overview of our solution is given in Sec. III, before describing the design of Serverledge in Sec. IV and its implementation in Sec. V. We discuss the experimental evaluation in Sec. VI and conclude in Sec. VII.

II. Related Works

As surveyed in [11] and [12], the increasing popularity of FaaS has attracted significant interest from the research community. A recent surge of interest is related to running serverless functions at the Edge [2], [3], in particular to handle IoT workloads [13] bringing functions closer to devices and thus reducing latency and energy consumption. However, running serverless functions and applications at the Edge raises a different set of research challenges, mainly because of resource constraints and geographic distribution of Edge nodes. In this section, we narrow our attention to serverless proposals that explicitly cope with Edge challenges, first discussing FaaS frameworks for the Edge related to our proposal and then considering function placement and load distribution.

Table I compares Serverledge to related FaaS systems proposed in the literature, as well as two established open-source frameworks, namely OpenWhisk and OpenFaaS. The solutions closest to ours are Colony [14] and Faasm [5], as they support function execution offloading.

Colony is a framework for parallel FaaS in the Cloud-Edge continuum. Its goal is to let nodes process data on their resources while also offering their computing capacity to the rest of the infrastructure. Colony differs from most existing FaaS frameworks, as it transparently converts the logic of complex user-given functions into task-based workflows backing on task-based programming models through COMPSs [15]. The generated workflows are then executed over the infrastructure, possibly offloading tasks both horizontally and vertically. To the best of our knowledge, the source code of Colony has not been publicly released.

Faasm is an open-source research prototype that introduced *Faaslets*, an isolation abstraction for high-performance serverless computing. Faaslets isolate the memory of executed functions using software-fault isolation (SFI), as provided by WebAssembly, while allowing memory regions to be shared between functions in the same address space. Relying on Faaslets, Faasm significantly reduces the initialization time and memory footprint of function sandboxes, compared to container-based approaches. Moreover, Faasm has built-in support for function chaining and state management. Faasm runs using multiple worker nodes, which can schedule and offload requests horizontally to other workers. However, Faasm does not explicitly consider geographical distribution of the nodes.

Sledge [6], [16] and tinyFaaS [7] are other FaaS frameworks specifically designed for Edge environments, aiming to provide serverless execution with reduced resource consumption. The key difference between the solutions mentioned above, including Serverledge, and these two frameworks lies in the fact that Sledge and tinyFaaS target single-node deployment scenarios and, thus, they lack the ability to exploit Cloud resources.

As regards the sandboxing mechanism used for function execution, Sledge adopts an approach similar to Faasm, exploiting software-fault isolation and WebAssembly-based runtime environments. Sledge has recently been extended in [16] to orchestrate and schedule the execution of function compositions through QoS-aware policies. While Serverledge does not currently support function chains or compositions, we plan to introduce similar features in the future.

Similarly to our approach, tinyFaaS relies on traditional Docker containers for isolated function execution. However, to limit the overhead due to dynamic management of running and idle containers, tinyFaaS uses a "static" pool of containers for each function. Indeed, a configurable number of containers are spawned upon registration of a new function, without waiting for invocation requests. Furthermore, tinyFaaS, which only supports JavaScript functions, allows multiple requests to be served concurrently within the same container, avoiding the additional memory footprint of concurrent container instances.

Compared to the research prototypes described above, OpenFaaS and OpenWhisk are feature-rich open-source FaaS frameworks, which have been primarily designed for Cloud and clustered computing environments. In particular, the architecture of OpenFaaS and OpenWhisk does not suit well geographically distributed environments, as they include centralized scheduling and management components (e.g., the *Controller* in OpenWhisk, the *Gateway* in OpenFaaS). Both these frameworks rely on software containers for isolation.

Research efforts have been also devoted to propose solutions to place and manage serverless functions and applications in Edge environments, also considering their integration with serverless Cloud services. Scheduling of serverless functions across heterogeneous and possibly resource-constrained Edge servers has been considered in a number of works (e.g., [8], [17]–[19]). They investigate optimal function placement with the goal of minimizing the completion time of serverless applications under the trade-off between processing time and communication overhead. For example, Deng et al. [18] propose a proactive algorithm to split the data traffic between Edge nodes. Schedulix [8] comprises a greedy algorithm to determine both the order and placement of functions over a hybrid public-private cloud. Optimization problem formulations have been employed for resource provisioning and allocation in Edge and Cloud serverless environments, e.g., [19], [20]. For example, by means of Mixed Integer Programming, NEPTUNE [19] places latency-constrained functions on Edge nodes according to user locations, by avoiding their saturation and exploiting GPUs if available. Model-driven resource management algorithms based on queuing theory have been also proposed, for example in LaSS [21] to determine the

TABLE I: Comparison of FaaS frameworks. (Dist. = System Distribution, C = Cluster-level, G = Geographical; H = Horizontal Offloading, V = Vertical Offloading, Comp. = Composition)

	Dist.	Offload	Comp.	Runtime	State	Languages
Colony [14]	G	H+V	Yes	COMPSs [15]	–	C++, Python
Faasm [5]	C	H	Yes	Faaslet (SFI-based)	Yes	C++, Python, (any compiled to Wasm)
OpenWhisk	C	–	Yes	Containers	–	Go, Java, JS, Python, PHP + ...
OpenFaaS	C	–	(Yes)	Containers	–	Go, Java, JS, Python, PHP + ...
Sledge [6], [16]	–	–	Yes [16]	SFI-based	Yes [16]	C++, (any compiled to Wasm)
tinyFaaS [7]	–	–	–	Containers (static)	–	JS
Serverledge	G	H+V	–	Containers	–	Python, JS, (any through custom images)

placement of each function and to auto-scale the allocated resources in response to workload dynamics.

The above approaches rely on centralized decision-making components and may suffer from scalability in large-scale Edge-Cloud environments. Decentralized approaches (e.g., [20], [22]) aim to address this issue by distributing decisions for admission, scheduling, and provisioning. For example, AuctionWhisk [22] is an auction-inspired approach, evaluated using OpenWhisk, where application users bid on resources, while Edge nodes decide locally which functions to execute and which to offload in order to maximize revenue. Decentralized heuristic algorithms are proposed in [20] but their evaluation is only by means of simulation.

Offloading strategies among Edge nodes have been also investigated in the field of serverless edge computing. DFaaS [9] leverages an overlay network to balance load within a federated Edge FaaS platform, composed of OpenFaaS nodes. Cicconetti et al. [10] propose an Internet Protocol-inspired algorithm to offload invocation requests within a network of FaaS nodes. Both the solutions can be considered to extend our work and introduce new strategies for horizontal offloading. Vertical offloading to Cloud has been also studied, for example in [23], which allows users to specify latency and cost requirements and determines where to execute the task on the basis of prediction models.

We observe that most of these proposals have been evaluated either by means of simulation or prototypes implemented on top of Cloud-native platforms such as OpenFaaS and OpenWhisk. Our proposed solution Serverledge represents a platform natively designed for the Edge that could be used by future works to investigate serverless function scheduling and load distribution in the Cloud-Edge continuum.

III. Overview of Serverledge

Serverledge is a decentralized FaaS platform designed for Edge-Cloud computing environments. Serverledge allows users to define functions through high-level programming languages and automatically allocates resources for their execution upon invocation. Following the approach adopted by most the existing FaaS platforms, including OpenWhisk and OpenFaaS, we execute functions within software containers, which are spawned as needed and initialized with the code and libraries required by each function.

Figure 1 illustrates the high-level architecture of a Serverledge installation, which. consists of one or more *nodes*,

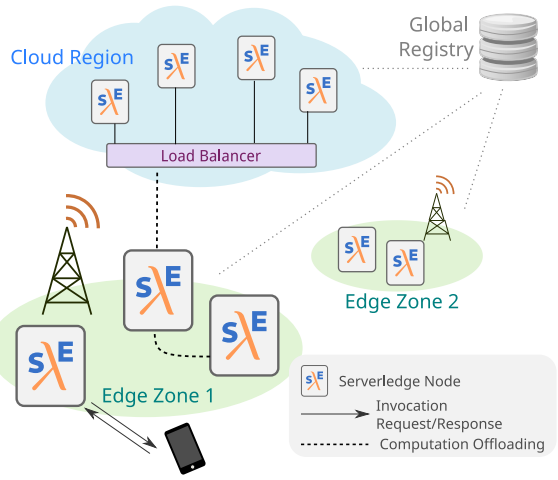


Fig. 1: Serverledge Overview.

deployed either in Cloud data centers or at the edge of the network, and a *global registry*. The latter provides distributed nodes with the required data about the system, including membership information about each deployed node. Within the registry, nodes are organized into different *cloud regions* (e.g., data centers) and *edge zones* based on their location. Cloud regions typically represent geo-distributed data centers, while Edge zones may be associated with, e.g., single towns or cities. Each Cloud region may further comprise a *load balancer* to distribute incoming requests to the nodes deployed in the region. Note that, while the global registry represents a single logical entity in the architecture, it may be associated with multiple replicas for scalability and fault tolerance.

The core idea underpinning the design of Serverledge is that there are not single or privileged entry points for function invocation. Indeed, users can send invocation requests to any node (e.g., one in their proximity). Compared to popular FaaS platforms designed for the Cloud, scheduling functionalities are not centralized and, thus, every node is able to schedule the execution of incoming requests. This is particularly important for Edge-generated requests, which are not forced to reach a centralized gateway in the Cloud for scheduling.

Serverledge adopts a per-request container scaling behavior, where new containers are only spawned when needed. In particular, when an invocation request enters the system, if enough resources (i.e., CPU and memory) are available, a new container is spawned and initialized to execute the function.

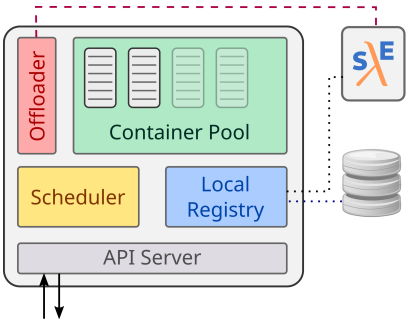


Fig. 2: Architecture of a Serverledge node.

When this happens, the request has to wait for the container to be fully initialized before being served and it is said to experience a "cold start". Following a common approach to reduce cold start frequency, containers are not immediately destroyed after function completion and are kept in a *warm pool* until a fixed timeout expires (e.g., 10-15 minutes in Cloud-based FaaS offerings). If one or more warm containers are available, these can be re-used to serve new requests for the same function avoiding a cold start.

Because of the limited resource capacity of Edge nodes, it is likely that a single node (and perhaps a whole Edge zone) cannot sustain the incoming load. Therefore, Serverledge allows nodes to *offload* invocation requests to other nodes, when needed. In particular, we support both *vertical* and *horizontal* offloading. The former refers to execution requests being forwarded from Edge to Cloud nodes, whereas the latter indicates request offloading among Edge nodes. According to the node organization described above and in the aim of keeping latencies under control, we assume that each Edge zone is associated with a single Cloud region for offloading. Similarly, horizontal offloading is enabled by default only within a single Edge zone.

IV. Architecture

A Serverledge node comprises a few key components, as depicted in Fig. 2: API server, Scheduler, Local Registry, Offloader and Container Pool. By interacting with each other and possibly with the Global Registry and other nodes, these components support the execution and scheduling functionalities we have introduced in the previous section. In the following, we will describe the design of each component and the interactions between different components.

A. Node API

Each node provides a set of key functionalities through an HTTP API, served by the *API server* component. The API is meant to be primarily used by client applications (e.g., to create and invoke their own serverless functions), but it is also accessible to other Serverledge nodes (e.g., for offloading, as illustrated in the following). In particular, each node supports the following key operations:

- `/create`: to register a new serverless function in the system, providing its source code and the required information for its execution (e.g., the amount of memory to reserve for its container instances).
- `/invoke`: to invoke an existing function, possibly specifying one or more input parameters and QoS requirements for the submitted request (e.g., QoS class, maximum response time).
- `/list`: to get a list of the registered functions.
- `/delete`: to de-register an existing function.
- `/status`: to obtain information about a node, including, e.g., the amount of available computational resources and the current state of its container pool.

B. Registry

As mentioned above, Serverledge uses a registry to store information about the nodes in the system and the registered functions, including their code. At system-level, the Global Registry keeps this information and makes it available to the nodes as needed. As such, updates to the existing functions (e.g., creation of a new function) must be communicated to the Global Registry, in order to make them visible to the whole system. In addition to accessing the Global Registry, each node is equipped with a *Local Registry*, which has a twofold role. First of all, it acts as a local cache for information retrieved from the Global Registry. By doing so, it provides local components (e.g., the scheduler) with low-latency access to most the data they use, avoiding unnecessary reads from the Global Registry. A time-to-live is associated with each cache entry to guarantee that information is periodically refreshed from the Global Registry.

Besides caching, the Local Registry is responsible for storing information that we aim to collect and manage on the node, without propagating it at global level. Specifically, each node deployed at the Edge runs periodic monitoring tasks to gather information about its neighbor nodes, located within the same Edge zone (e.g., the same town). In particular, these nodes run the well-known *Vivaldi* [24] algorithm to build and update a virtual coordinate space, which allows nodes to estimate the network distance among each other. The Vivaldi algorithm require nodes to periodically exchange their own virtual coordinates. We exploit such message exchange to spread additional information about each node, including the current amount of available resources (i.e., CPU and memory) and a synthetic snapshot of the container pool in terms of existing warm containers. Such monitoring allows nodes to identify non-overloaded close neighbors, in terms of network distance, which can be regarded as ideal candidates for computation offloading, when needed.

C. Function Scheduling and Execution

When a function invocation request is received, the node retrieves the necessary information about the function (i.e., required runtime environment, memory and CPU demand, source code) from the Local Registry, which – in turn – will interact with the Global Registry if the required data have not been cached. Then, the request is passed to the *Scheduler*

component, which must provision the required resources for execution, if possible, or drop the request.

In principle, the scheduling process boils down to identifying a suitable container for function execution, either retrieving it from the pool of warm containers or creating a new one. If a new container is needed, we create it from a suitable base image as specified by the function (e.g., functions written in Python require the Python interpreter). Once the container is started, we finalize the initialization by copying the source code package of the function into the container. The invocation request has to wait for the whole initialization procedure before the actual execution starts (i.e., the well-known cold start issue).

As mentioned, containers are not terminated after function execution and are instead kept in a pool of warm containers for future re-use. Containers remain in the warm pool until any of two events occurs and, specifically: (i) the container has been idle for a period longer than a threshold $T^{expired}$, or (ii) we need to create a container for a different function and must reclaim memory from the warm pool to do so. While more advanced techniques have been explored in the literature for cold start reduction and mitigation (e.g., [25], [26]), they are out of the scope of this paper and will be considered for future extensions.

As depicted in Fig. 3, besides picking or creating containers for function execution, the Scheduler can make other decisions for incoming requests. First, the scheduler can decide to offload requests to another node, which will take care of actual function execution (more details in the next section). Furthermore, requests can be dropped by the Scheduler and, thus, not executed at all (e.g., during heavy-load periods).

D. Execution Offloading

Offloading the execution of functions allows nodes to cope with high load periods, by moving a share of their own workload to peers. Besides node congestion, offloading may be useful in general to optimize the provided service level, e.g., letting particular requests being served remotely on specialized hardware for higher performance.

Serverledge supports both vertical (i.e., from the Edge to the Cloud) and horizontal (i.e., within an Edge zone) offloading. On the one hand, vertical offloading typically allows nodes to significantly increase their accessible computing capacity, as Cloud regions likely offer more and/or more powerful nodes. Conversely, horizontal offloading involves single nodes in the neighborhood, which do not necessarily offer better performance than the originally targeted node. On the other hand, the network delay between Edge and Cloud may impose non-negligible overhead on offloaded requests, especially if their computation demand is limited. In this regard, horizontal offloading is attractive, as target nodes are selected based on proximity metrics and can be reached with reduced delays.

While horizontal and vertical offloading appear as distinct levers to the Scheduler, which should carefully pick one or the other depending on the circumstances, the offloading mechanisms are not significantly different on a system design perspective. When the Scheduler makes an offloading decision for a request, a target node is selected relying on the Local Registry, which provides information on the neighbor Edge nodes and the available Cloud regions (if any). Then, the request is forwarded to the selected node. For this purpose, the local node acts as a reverse proxy, submitting the invocation request to the API of the remote node. The local node waits for the computation result travelling back from the remote node and sends it back to the invoking client as soon as possible.

Offloaded requests incur the same scheduling process on the remote node, although we may and actually distinguish them from regular, client-generated ones. For instance, a default constraint we integrated in Serverledge is that offloaded requests cannot be further offloaded by the remote node. While our design would support such recursive offloading, a longer forwarding chain may easily undermine the ability of the Scheduler to estimate and control incurred latency. We will show in the following that the proposed offloading mechanisms significantly boost the capacity of single nodes, with minimal overheads.

V. Implementation

Serverledge has been implemented in Go.[1] In this section, we briefly review the main implementation aspects for each component of the architecture.

API server. Each Serverledge node exposes an HTTP API, which is implemented on top of *Echo*[2], a high performance web framework for Go.

Registry. For the Global Registry, we exploit *etcd*[3], a distributed, reliable key-value store. As regards the Local Registry, we implement it by means of two sub-components: (i) a cache for the function metadata retrieved from the Global Registry, and (ii) a neighborhood monitoring service. The latter uses UDP messages for communication among Edge nodes and latency estimation through network coordinates.

Scheduler. The Scheduler is a multi-threaded component that receives invocation requests from the API server and manages their execution. The Scheduler relies on a `Policy` interface to support flexible policy definition. The `Policy` interface comprises three operations, namely `Init()`, `OnArrival()`, `OnCompletion()`. The first operation is used at startup to initialize any necessary data structure, whilst the remaining ones are, respectively, used upon arrival or completion of a request.

Container Pool. The Container Pool comprises a few data structures that store and provide access to the existing running and warm containers for each function. The Container Pool has been implemented avoiding coupling with a specific containerization platform and provides interfaces for the integration of multiple platforms. However, in this work we focused on a single platform, Docker, given its popularity and ease of use. In particular, we exploited the Docker SDK for Go to interact with the Docker daemon on each node.

[1] `https://github.com/grussorusso/serverledge`
[2] https://echo.labstack.com/
[3] https://etcd.io/

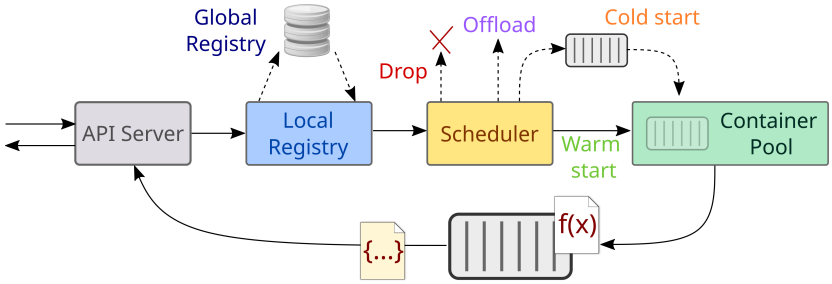


Fig. 3: Illustration of function scheduling. If local execution is not possible, the Scheduler can drop the request or offload it.

Offloading. The offloading mechanisms have been introduced with minimal impact on the node architecture and design. Indeed, the same API used by regular clients is exploited for offloading requests, with the originally targeted node acting as a reverse proxy. Nonetheless, the implementation of this mechanism required same care. While we relied on the HTTP client provided by Go for making new requests towards, e.g., Cloud nodes, we found it essential to tune the client configuration to achieve good performance. Specifically, as the default configuration does not allow to keep more than two idle TCP connections per single host, TCP connection reuse was not properly exploited in the offloading scenario. Letting the client keep a much larger number of idle connections (e.g., 2,000) significantly reduced the offloading overheads.

VI. Evaluation

To evaluate Serverledge, we first compare it against state-of-the-art FaaS platforms. Then, we evaluate the offloading mechanisms integrated in Serverledge.

A. Experimental Setup

Infrastructure. We run the experiments on AWS EC2 virtual machines. To mimic an Edge-Cloud scenario, we deploy Edge and Cloud nodes in different AWS *regions*, i.e., respectively, `eu-central-1` in Germany and `eu-west-1` in Ireland. To further differentiate Edge and Cloud nodes, we consider different types of EC2 instances for them. Edge nodes run in `c4.large` instances with 2 vCPUs and 3.75 GB of memory, whilst Cloud nodes run in `c4.xlarge` instances, with 4 vCPUs and 7.5 GB of memory. A `c4.2xlarge` instance is used as a client and located in the same region of Edge nodes. Unless differently specified, for the Global Registry of Serverledge, we deploy a single instance of etcd in one of the Cloud nodes.

Workloads. We use *Locust*, a Python-based load testing tool, for load generation. Locust allows us to emulate the behavior of N_U users concurrently issuing requests to Serverledge, with configurable think times or maximum rates. The duration of each experiment is set to 900s.

We use the following functions in the experiments:

- `Sieve`: implementation of the *Sieve of Eratosthenes*, which computes the list of primes up to a given bound (i.e., 10,000 in the experiments). Implemented in JavaScript by authors of [7].
- `Fib`: recursive computation of the n-th element of the Fibonacci sequence. Implemented in Python and Go.
- `Classification`: simple binary image classifier based on a convolutional neural network, implemented in Python on top of Keras and Tensorflow.
- `Validator`: validator of user-submitted JSON objects, according to a given schema. Implemented in Python.

Serverledge Configuration. We set the warm container expiration timeout to 600s. The Local Registry monitoring interval is set to 30s, with its cache time-to-live set to 60s.

B. Performance Comparison

We consider three state-of-the-art platforms, namely OpenWhisk, Faasm and tinyFaaS, which have been introduced in Sec. II, to compare Serverledge performance. For this comparison, we deploy each platform using a single Edge node, with an additional node used for load generation. Note that OpenWhisk is usually deployed in Cloud-based clusters through Kubernetes, as it relies on several distributed components. Because OpenWhisk also supports a "lean" deployment mode for Edge scenarios, we will consider this setup for the experiments. Both tinyFaaS and Faasm are deployed using Docker containers, following the official instructions. For this experiment, we also deploy the etcd-based Global Registry in the Edge node, to have a fair comparison with the other systems, deployed in a single node.

To assess the maximum throughput sustained by each platform, we let $N_U = 20$ parallel users generate as many requests they can (i.e., with no think time between consecutive requests)[4]. All the platforms execute the `Sieve` function. Unfortunately, we were not able to run Faasm at high throughput, with the system keeping an excessive number of open files and crashing.[5] Therefore, we perform the comparison against Faasm using a different, reduced workload, which is presented later in this section.

1) Results with OpenWhisk and tinyFaaS: The results of this comparison are reported in Table II. Figure 4 shows the throughput over time, while Fig. 5 compares response time distributions (with whiskers from the 5^{th} to the 95^{th} percentile). We first note that OpenWhisk, which is not designed

[4]We verified that the workload saturates system capacity by doubling the number of users without observing evident throughput increases.

[5]The issue we encountered is likely the same reported here: https://github.com/faasm/faasm/issues/504. The issue is "open" at the time of writing.

TABLE II: Comparison of Serverledge, tinyFaaS and OpenWhisk.

	Thr. *(req/s)*	**Response Time** *(ms)* Avg	Min	Max	P25	P50	P75	P90	P95	P99
OpenWhisk	55.47	322.25	43.20	4801.10	250	290	340	430	510	740
tinyFaaS	1357.84	13.06	1.89	240.04	7	12	17	24	29	39
Serverledge	805.06	22.03	2.39	3049.65	16	21	28	34	39	50
Serverledge (with offloading)	1827.28	10.17	2.44	1466.29	7	9	12	15	18	24
OpenWhisk (reduced workload)	53.56	332.58	30.64	4765.79	260	290	360	450	520	810
tinyFaaS (reduced workload)	89.44	3.42	1.88	209.37	3	3	4	5	6	7
Serverledge (reduced workload)	89.38	3.73	2.56	1767.85	3	3	4	4	5	8

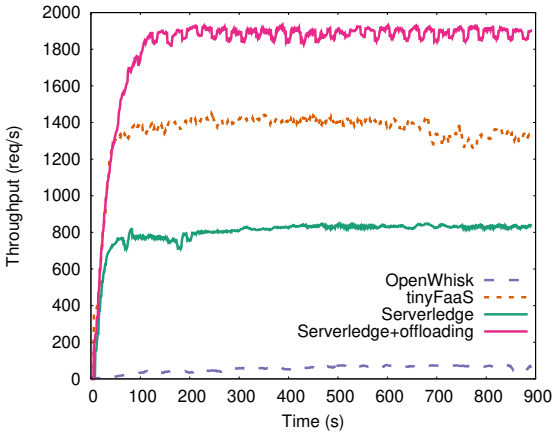


Fig. 4: Throughput of OpenWhisk, tinyFaaS and Serverledge running in a single node. tinyFaaS has the highest throughput among them, but Serverledge can exploit additional nodes to offload and serve more requests.

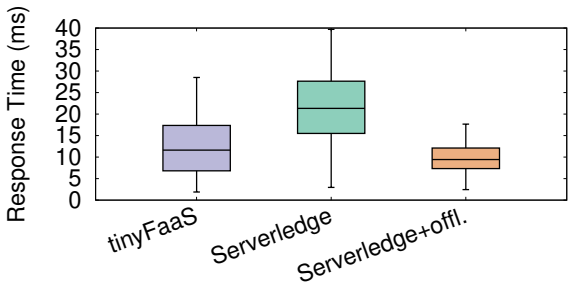


Fig. 5: Response time of tinyFaaS and Serverledge (with and without offlading).

for resource-limited deployments, shows poor performance in the considered scenario, with an average throughput equal to 55.5 req/s. Similarly, OpenWhisk has the worst performance in terms of response time, with the median response time being 290ms, compared to 21ms achieved by Serverledge.

The platform showing the best performance is tinyFaaS, whose measured throughput is 1358 req/s and median response time equal to 12 ms. Serverledge processes 805 req/s with a median response time equal to 21 ms. While not exciting, these results were expected. Indeed, tinyFaaS adopts a simplified container management approach that significantly reduces the overheads associated with function execution. Specifically, it statically allocates a pool of containers for each function when the system is started, with each container allowed to serve multiple requests concurrently. By doing so, tinyFaaS avoids cold starts and the overheads due to container management. This is evident looking at the maximum response time achieved using tinyFaaS (i.e., 240 ms) compared to that measured in Serverledge (i.e., 3,049 ms, which corresponds to a cold start). While the design of tinyFaaS appears optimal, it may hardily scale in a more general scenario, as pre-spawning containers for every existing function, regardless of its invocation patterns, would likely require more memory than provided by the node.

Furthermore, tinyFaaS is not able to scale its execution across multiple nodes. To demonstrate the different direction pursued by Serverledge, we also consider the case where an additional node is available in the same region to offload requests. By doing so, we are able to process more than 1800 req/s, reducing also the median response time to 9 ms.

As a final comparison, we consider what happens under a reduced workload, with each user issuing requests at a maximum rate of 5 req/s. OpenWhisk has the worst results even in this case, completing about 54 req/s. Serverledge and tinyFaaS shows almost identical throughput, with minimal differences in the response time distribution (see, Table II).

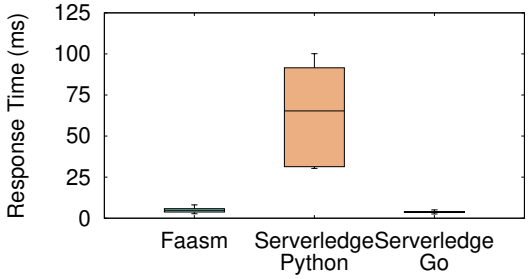
2) Results with Faasm: Because of the aforementioned issue, we consider a reduced throughput scenario to compare Serverledge and Faasm, where we focus on response time evaluation. Specifically, we consider $N_U = 5$ users, issuing no more than 5 req/s. We use the `Fib` function for the experiments, considering the cases where it is invoked with argument $n = 20$ and, then, $n = 25$. As regards Faasm, we rely on the recursive Fibonacci implementation comprised in the official repository. The latter consists of C++ code, compiled to WebAssembly for execution. Since we are not able to run an identical function implementation, we implement the same algorithm both in Python and Go for execution in Serverledge.

The results of these experiments are reported in Table III, with Fig. 6 showing the measured response time. Clearly, as the rate of incoming requests is low, both Faasm and Serverledge can sustain the incoming load. Looking at the response time, we note that Faasm serves 99% of the requests in no more than 13 ms, with both the considered inputs. Serverledge with the Python runtime has worse performance, especially when computing `Fib(25)`, with a median response time equal to 31 ms. The response time of Serverledge is dramatically reduced when running the compiled Go imple-

TABLE III: Comparison of Serverledge and Faasm.

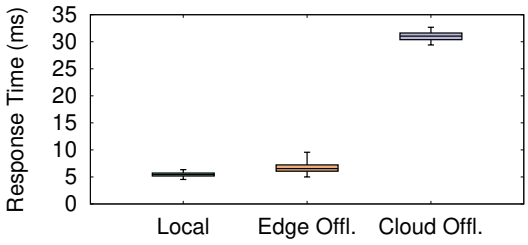
		Response Time *(ms)*							
	Thr. *(req/s)*	Avg	Min	Max	P25	P50	P75	P90	P95	P99
Faasm - Fib(20)	14.51	4.67	1.93	219.70	3	4	5	6	7	12
Serverledge - Fib(20) *Python impl.*	14.49	8.60	5.68	1146.65	7	7	10	11	11	13
Serverledge - Fib(20) *Go impl.*	14.50	3.76	2.46	480.40	3	3	4	5	5	7
Faasm - Fib(25)	14.51	5.14	2.69	223.00	4	5	6	7	8	13
Serverledge - Fib(25) *Python impl.*	14.49	63.22	29.92	1325.18	31	65	92	94	96	120
Serverledge - Fib(25) *Go impl.*	14.50	4.04	2.60	476.92	4	4	4	4	5	7

(a) `Fib(20)`

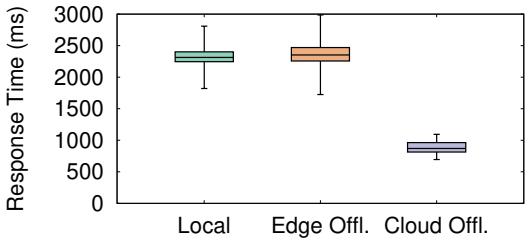


(b) `Fib(25)`

Fig. 6: Response time of Faasm and Serverledge running the `Fib` function with different inputs.



(a) `Validator` function.



(b) `Classifier` function.

Fig. 7: Response time of the `Validator` and `Classifier` functions in the Edge-Cloud scenario. For `Validator`, Cloud offloading has evident impact on response time due to network delay, as expected. Conversely, for the resource-demanding `Classifier`, the speedup provided by Cloud nodes clearly outweighs the impact of network delay.

mentation of the function. In this case, Serverledge shows response times comparable to those measured with Faasm, and even better on average.

We remark that the benefits of Faasm are still evident looking at the maximum response time in Table III. Indeed, exploiting lightweight function runtimes, cold starts in Faasm have reduced impact on response time compared to Serverledge. We also plan to consider alternatives to Docker containers for function execution as future work.

C. Offloading Evaluation

In this set of experiments, we focus on the offloading mechanisms integrated in Serverledge. As such, we now consider both Edge and Cloud nodes, deployed in different geographical locations. Specifically, we deploy two nodes at the Edge and two nodes in the Cloud. The Cloud region also comprises a round-robin load balancer, which is the target for vertically offloaded requests from the Edge. An additional node is located at the Edge for load generation, with all requests directed to a single Edge node. As such, the other Edge node and the Cloud nodes are available for offloading. The measured median latency between the Cloud and Edge region is about 25 ms.

We run the experiments running the `Classifier` function, which uses a TensorFlow model to classify input images, and the `Validator` function, which validates input JSON data. Among them, `Classifier` is more computationally demanding than `Validator` and we will show that this has a major impact on the offloading performance.

As a first experiment, we want to compare local function execution, vertical offloading to the Cloud and horizontal offloading to the Edge. For this purpose, we consider three configurations in which only one of these scheduling options is enabled at a time (e.g., the node can only offload requests to the Edge). Being mainly interested in evaluating the impact of different offloading strategies on response time, we consider medium-intensity workloads for each function and, specifically: 5 users issuing at most 0.5 req/s for `Classifier`, 20 users with the same rate limitation for `Validator`.

Figure 7a shows the response time for the `Validator`

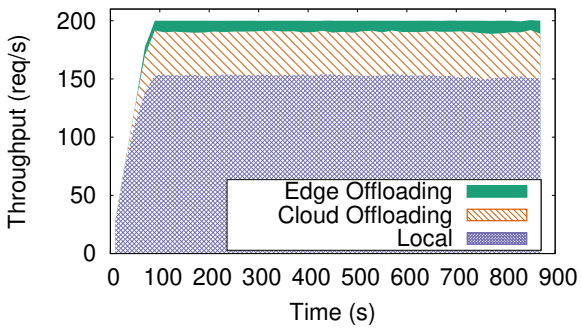


Fig. 8: Throughput of the system in the Edge-Cloud scenario.

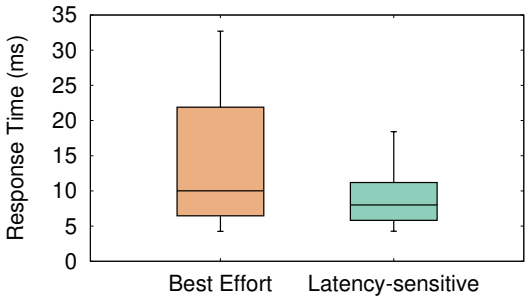


Fig. 9: Response time of different QoS classes in the Edge-Cloud scenario.

function. As expected, local function execution leads to the least response times (no more than 9 ms for 99% of the requests), while requests offloaded to the Edge experience moderately higher response times (up to 14 ms for 99% of the requests). As such, we can observe that the offloading mechanism itself introduces limited overhead, in the order of few milliseconds. Requests offloaded to the Cloud show much higher response times, exceeding 30 ms. This is not surprising, as the extra latency corresponds to the observed network delay between Edge and Cloud in our experimental setting.

We then repeat this experiment with the `Classifier` function, which is computationally heavier. As shown in Fig. 7b, the results in this case are quite different. In fact, locally executed and Edge-offloaded requests are the those experiencing the highest response time, with a median value larger than 2 s. Conversely, requests offloaded to the Cloud report a median response time lower than 1 s. While these results seem to conflict those discussed above, we must keep in mind that the Cloud region provides a total of 8 vCPUs, compared to the 2 vCPUs of each Edge node. As this function has a higher CPU demand, the benefits provided by more CPU resources outweigh the additional network latency (about 25 ms).

On the one hand, these experiments demonstrate the functionality of our offloading mechanisms, which cause limited overhead. On the other hand, we observe that vertical and horizontal offloading must be properly exploited depending on the considered scenario, including the function to execute. As such, a scheduling and offloading policy should be adaptive and take into account multiple factors to make optimal use of the available local and remote resources.

Although we plan to investigate the definition of proper offloading policies as future work, we present a final experiment to demonstrate how Serverledge supports the definition of QoS-aware policies, and how offloading mechanisms can be exploited along with local execution.

We focus on the `Validator` function and consider two different service classes, associated with each incoming request: (i) *latency-sensitive* requests, which should be served as quickly as possible; (ii) *best effort* requests, which do not have any particular requirement on the response time. We randomly associate each request with a service class, such that each request has 20% probability of being latency-sensitive. We aim to generate a load higher than what a single Edge node can sustain and consider 40 users issuing at most 5 req/s.

We define a proof-of-concept policy for this experiment, which works as follows. The scheduler tries to process each request locally on the node. If this is not possible, because there is not enough memory on the node, the request is offloaded. Specifically, latency-sensitive requests are offloaded to the Edge, to avoid the additional network delay. Conversely, best effort requests are offloaded to the Cloud, to preserve Edge resources.

Figure 8 shows the system throughput in this experiment, highlighting the share of requests that are served locally and those offloaded. We can note that the vast majority of the requests get served locally. As the node capacity does not allow to process all the requests, some of them are offloaded to the Cloud (i.e., the best effort ones, according to the policy in use) or to the Edge (i.e., the latency-sensitive ones). The response time for the two classes of requests is shown in Fig. 9. We can note that latency-sensitive requests benefit from the scheduling policy in use with significantly reduced response times, as desired.

VII. Conclusion

We presented Serverledge, a FaaS platform that blends together decentralized control, to suit geographically distributed infrastructures, and the ability to offload computation to exploit Cloud resource richness. Our evaluation shows that Serverledge outperforms existing platforms designed for clustered environments and has competitive performance compared to state-of-the-art frameworks designed for the Edge, while also supporting computation offloading.

As future work, we plan to extend Serverledge to support function compositions and state management, enabling the execution of complex applications. Furthermore, we will consider the integration of lighter function sandboxing techniques, following the approach adopted, e.g., by [5], [6].

Acknowledgments

This work is partially supported by the Italian National Research Centre in High Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR within the NextGenerationEU program.

REFERENCES

[1] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, pp. 76–84, 2021. [Online]. Available: https://doi.org/10.1145/3406011

[2] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *Proc. of 2021 Australasian Computer Science Week Multiconference, ACSW '21*. ACM, 2021.

[3] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, "When serverless computing meets edge computing: Architecture, challenges, and open issues," *IEEE Wirel. Commun.*, vol. 28, no. 5, pp. 126–133, 2021.

[4] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017. [Online]. Available: https://doi.org/10.1109/MC.2017.9

[5] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. of 2020 USENIX Annual Technical Conference, ATC '20*. USENIX Association, 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/system/files/atc20-shillaker.pdf

[6] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a serverless-first, light-weight wasm runtime for the edge," in *Proc. of 21st Int'l Middleware Conf., Middleware '20*. ACM, 2020, pp. 265–279.

[7] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *Proc. of 2020 IEEE Int'l Conference on Fog Computing, ICFC '20*. IEEE, 2020, pp. 17–24.

[8] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *Proc. of IEEE 13th Int'l Conf. on Cloud Computing, CLOUD '20*. IEEE, 2020, pp. 609–618.

[9] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo, "Dfaas: Decentralized function-as-a-service for federated edge computing," in *Proc. of 10th IEEE Int'l Conference on Cloud Networking, CloudNet '21*. IEEE, 2021, pp. 1–4.

[10] C. Cicconetti, M. Conti, and A. Passarella, "A decentralized framework for serverless edge computing in the internet of things," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 2, pp. 2166–2180, 2021.

[11] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, 2021, just Accepted.

[12] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy, and future directions," *ACM Comput. Surv.*, 2022, just Accepted.

[13] G. S. Cassel, V. Rodrigues, R. da Rosa Righi, M. Rosecler Bez, N. A. C., and C. André da Costa, "Serverless computing for internet of things: A systematic literature review," *Future Gener. Comput. Syst.*, vol. 128, pp. 299–316, 2022.

[14] F. Lordan, D. Lezzi, and R. M. Badia, "Colony: Parallel functions as a service on the cloud-edge continuum," in *Proc. of 27th Int'l Conf. on Parallel and Distributed Computing, Euro-Par '21*, ser. LNCS, vol. 12820. Springer, 2021, pp. 269–284.

[15] R. M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 3-4, pp. 32–36, 2015.

[16] X. Lyu, L. Cherkasova, R. Aitken, G. Parmer, and T. Wood, "Towards efficient processing of latency-sensitive serverless dags at the edge," in *Proc. of 5th ACM Int'l Workshop on Edge Systems, Analytics and Networking, EdgeSys '22*. ACM, 2022, p. 49–54.

[17] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proc. of 2019 IEEE/ACM 27th Int'l Symp. on Quality of Service, IWQoS '19*. IEEE, 2019, pp. 1–10.

[18] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, and A. Y. Zomaya, "Dependent function embedding for distributed serverless edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2346–2357, 2022.

[19] L. Baresi, D. Hu, G. Quattrocchi, and L. Terracciano, "Neptune: Network- and gpu-aware management of serverless functions at the edge," in *Proc. of Int'l Symp. on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '22*. IEEE, 2022.

[20] O. Ascigil, A. Tasiopoulos, T. K. Phan, V. Sourlas, I. Psaras, and G. Pavlou, "Resource provisioning and allocation in function-as-a-service edge-clouds," *IEEE Trans. Serv. Comput.*, 2021.

[21] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proc. of 30th Int'l Symp. on High-Performance Parallel and Distributed Computing, HPDC '21*. ACM, 2021, p. 239–251.

[22] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, "Auctionwhisk: Using an auction-inspired approach for function placement in serverless fog platforms," *Softw. Pract. Exp.*, vol. 52, no. 5, pp. 1143–1169, 2022.

[23] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proc. of IEEE/ACM CCGrid '20*. IEEE, 2020, pp. 41–50.

[24] R. Cox, F. Dabek, M. F. Kaashoek, J. Li, and R. T. Morris, "Practical, distributed network coordinates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 113–118, 2004.

[25] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proc. of IEEE/ACM CCGrid '21*. IEEE, 2021, pp. 797–803.

[26] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. of ASPLOS '20*. ACM, 2020, pp. 467–481.