Pervasive and Mobile Computing 100 (2024) 101915

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

A framework for offloading and migration of serverless functions in the Edge–Cloud Continuum

Gabriele Russo Russo *, Valeria Cardellini, Francesco Lo Presti

Department of Civil Engineering and Computer Science Engineering, Tor Vergata University of Rome, Italy

ARTICLE INFO

Keywords:
Serverless
Edge computing
Offloading
Migration

ABSTRACT

Function-as-a-Service (FaaS) has emerged as an evolution of traditional Cloud service models, allowing users to define and execute pieces of codes (i.e., functions) in a serverless manner, with the provider taking care of most operational issues. With the unending growth of resource availability in the Edge-to-Cloud Continuum, there is increasing interest in adopting FaaS near the Edge as well, to better support geo-distributed and pervasive applications. However, as the existing FaaS frameworks have mostly been designed with Cloud in mind, new architectures are necessary to cope with the additional challenges of the Continuum, such as higher heterogeneity, network latencies, limited computing capacity.

In this paper, we present an extended version of Serverledge, a FaaS framework designed to span Edge and Cloud computing landscapes. Serverledge relies on a decentralized architecture, where each FaaS node is able to autonomously schedule and execute functions. To take advantage of the computational capacity of the infrastructure, Serverledge nodes also rely on horizontal and vertical function offloading mechanisms. In this work we particularly focus on the design of mechanisms for function offloading and live function migration across nodes. We implement these mechanisms in Serverledge and evaluate their impact and performance considering different scenarios and functions.

1. Introduction

Serverless computing has enjoyed an ever increasing popularity since its appearance in the last decade and it is expected to reach a projected market value of $36.8 billion by 2028 [1]. Although we still lack a standard definition of serverless, fundamental and characterizing aspects have been recently identified in the fact that serverless "allows to develop, deploy, and run applications (or components thereof) in the cloud without allocating and managing virtualized servers and resources or being concerned about other operational aspects" [2]. While serverless computing has various faces and interpretations, the most popular and prominent incarnation is *Function-as-a-Service* (FaaS). FaaS originated as an evolution of traditional Cloud service models, and allows users to deploy units of computation, defined as *functions*, and execute them in response to events (e.g., HTTP triggers). Importantly, functions are executed in a serverless fashion, with the provider taking care of most the operational issues, from provisioning a (virtual) server, installing the required libraries, deploying the software, scaling. Fine-grained pricing models, where users are only charged for the resources actually used to execute functions, and seamless scalability have further boosted the popularity of FaaS, with all the major Cloud providers now embracing this emerging service model in their catalogues (e.g., with services like AWS Lambda, Google Cloud Functions, Azure Functions). Besides them, several open-source FaaS frameworks have appeared, including Apache *OpenWhisk*, *OpenFaaS*, *Knative*, *nuclio*.

* Corresponding author.
E-mail addresses: russo.russo@ing.uniroma2.it (G. Russo Russo), cardellini@ing.uniroma2.it (V. Cardellini), lopresti@info.uniroma2.it (F. Lo Presti).

https://doi.org/10.1016/j.pmcj.2024.101915
Received 31 August 2023; Received in revised form 13 February 2024; Accepted 6 March 2024
Available online 8 March 2024
1574-1192/© 2024 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

FaaS emerged as an evolution of traditional Cloud service models like Infrastructure-as-a-Service and Platform-as-a-Service and, as such, it is not surprising that the adoption of FaaS mostly regards Cloud-based applications and infrastructures. Nonetheless, researchers and practitioners have started wondering whether and how FaaS could be successfully adopted out of Cloud data centers, at the edge of the network and in the emerging Edge-to-Cloud continuum [3–5], to support geographically distributed and pervasive applications and services. Indeed, application domains like Industrial Internet-of-Things, smart healthcare, and augmented/virtual reality, have often strict latency requirements [6], and, hence, cannot tolerate the delays caused by Edge-to-Cloud communication and data transfers. In contrast, emerging Edge computing environments represent promising solutions for these applications, enabling near-user, low-latency computation. Therefore, it is natural to investigate how emerging computing paradigms for the Cloud can be beneficial for pervasive and ubiquitous applications as well.

As mentioned, there is a large number of open-source FaaS frameworks available nowadays, representing a promising toolbox to bring FaaS at the edge of the network. However, looking at the architecture of these frameworks, it is easy to observe that they are mostly designed to run in Cloud or clustered environments. In particular, 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 to investigate solutions to better support FaaS beyond the Cloud scenario. A few works (e.g., [7–9]) study architectures and algorithms for function placement and load distribution in decentralized FaaS systems, but still relying on the existing Cloud-oriented frameworks for actual function execution, possibly incurring in some of the aforementioned issues. Other works have proposed novel frameworks. For instance, some solutions exploit lightweight function sandboxing mechanisms instead of OS-level virtualization (e.g., Faasm [10] and Sledge [11]), to better suit resource-constrained deployments. However, these solutions either work within single Edge nodes (e.g., [11,12]), or scale over multiple nodes without considering geographical distribution (e.g., [10]).

Our group has recently presented *Serverledge* [13], an open-source[1] FaaS framework 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.

In this paper, we extend Serverledge with additional mechanisms to better support function scheduling and execution, focusing in particular on function offloading and migration. We study two different mechanisms for function offloading, based, respectively, on recursive request forwarding and HTTP redirection. We discuss the merits of both the approaches and show that none outperforms the other in every scenario. We then propose an algorithm to dynamically select the best mechanism to offload incoming requests at run-time.

Offloading allows Serverledge to transfer the execution of a function from one node to another, as soon as an invocation request is received (e.g., because the node is overloaded). In this work, we introduce an additional mechanism, namely function *live migration*, to allow nodes to transfer the execution of a function to another node *after* the function has started. For this purpose, we integrate live container migration in Serverledge, studying a suitable migration protocol both for synchronous and asynchronous function invocations.

Our key contributions can be summarized as follows:

- We present the design of Serverledge with new features and components introduced with respect to [13], namely, asynchronous function invocation, the ability of provisioning limited shares of CPU time to functions, the ability of buffering incoming requests, live function migration, and the use of Podman as an alternative function runtime alongside Docker.
- We present and compare two approaches for function offloading. As none of the mechanisms provides the best performance in every situation, we devise an algorithm to adaptively choose the most convenient mechanism at run-time.
- We design and implement a live function migration mechanism for Serverledge, that allows us to migrate running function instances in presence of both synchronous and asynchronous invocations. We demonstrate the functionality of function migration and its overhead in terms of latency.

We remark that, compared to our previous publication [13], in this work we introduce several new mechanisms, namely asynchronous function invocation, CPU capping for functions, request buffering, an alternative container runtime for function isolation (i.e., Podman), offloading based on HTTP redirection and live function migration.

The remainder of the paper is organized as follows. We review related work in Section 2. In Section 3 we present the architecture of Serverledge and the design of its key components. We focus on the design of offloading mechanisms in Section 4, where we also introduce our adaptive mechanism selection algorithm. In Section 5 we describe the design and implementation of live function migration in Serverledge. We present the experimental evaluation in Section 6 and conclude in Section 7.

[1] https://github.com/grussorusso/serverledge

Table 1
Comparison of FaaS frameworks. (Dist. = System distribution, C = Cluster-level, G = Geographical; Dec. Sched. = Decentralized scheduling; Offl. = Function execution offloading, H = Horizontal offloading, V = Vertical offloading, Migr. = Live function migration; Comp. = Function composition).

	Dist.	Dec. Sched.	Offl.	Migr.	Comp.	Runtime
Colony [20]	G	✓	H+V	–	✓	COMPSs [21]
faasm [10]	C	✓	H	–	✓	
funcX [22]	G	✗	–	–	✓	
OpenWhisk	C	✗	–	–	✓	Container
OpenFaaS	C	✗	–	–	✓[a]	Container
Sledge [11,23]	–	–	–	–	✓	
TEMPOS [24]	G	✗	–	–	✓	Process/WASM-based
tinyFaaS [12]	–	–	–	–	–	Container (static)
Serverledge	G	✓	H+V	✓	–	Containers

[a] Through an external project.

2. Related work

The interest of researchers in serverless computing and FaaS has enormously grown in the last years, as demonstrated by several surveys on the topic (e.g., [14–18]). A recent surge of interest is related to running serverless functions at the edge of the network [3–5], in particular to handle IoT workloads [16], bringing functions closer to devices and thus reducing latency and energy consumption. However, adopting FaaS out of traditional Cloud or clustered environments raises a different set of research challenges, mainly because of resource constraints and geographic distribution of Edge nodes [5,19]. In this section, we focus on proposals that explicitly cope with these challenges. First, we discuss FaaS frameworks for the Edge related to our proposal. Then, we review works dealing with function placement and load distribution in Edge–Cloud FaaS systems, focusing on solutions for function offloading and migration.

2.1. Frameworks

Table 1 compares Serverledge to related systems proposed in the literature for FaaS at the edge of the network, as well as to two established open-source frameworks for the Cloud, namely OpenWhisk and OpenFaaS. The solutions closest to ours are Colony [20] and Faasm [10], as they support function execution offloading.

Colony is a framework for parallel FaaS in the Cloud–Edge continuum. The goal of Colony is to let distributed nodes process data on their embedded compute resources while also offering their computing capacity to the rest of the infrastructure. A distinguishing feature of Colony with respect to the most popular FaaS frameworks is the ability of transparently converting the logic of complex user-given functions into task-based workflows backing on task-based programming models through COMPSs [21]. 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 are implemented using WebAssembly, and isolate the memory of executed functions using software-fault isolation (SFI), while allowing memory regions to be shared between functions in the same address space. WebAssembly (Wasm) is a portable, binary instruction format for memory-safe, sandboxed execution and as such has emerged as a promising approach for supporting serverless at the Edge [25,26]. Relying on Faaslets, Faasm significantly reduces the initialization time and memory footprint of function sandboxes, compared to container-based approaches. Moreover, thanks to the ability of sharing memory regions, Faasm supports efficient function chaining and communication. 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 [11,23] and tinyFaaS [12] 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 [23] 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.

Designed for scalable and high performance remote function execution, $f$uncX [22] is a distributed FaaS framework that decouples cloud-based management functionality from edge-hosted function execution, supporting multiple runtime environments. However, compared to the other frameworks described above, management and scheduling duties remain in the Cloud. Garbugli

et al. [24] propose a holistic platform, named TEMPOS, to manage serverless functions with differentiated QoS guarantees in the compute continuum. They exploit various mechanisms for QoS enforcement at system-level, including Linux real-time scheduling, differentiated MOM priorities, and Time-Sensitive Networking (TSN).

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.

2.2. Function scheduling and offloading

Besides novel FaaS implementations, researchers have also been working on strategies for resource allocation and scheduling for serverless functions in Edge-to-Cloud environments, including approaches to offloading and migration. The problem of scheduling function execution across heterogeneous and possibly resource-constrained Edge servers has been considered in a number of works (e.g., [7,27–29]). 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. [28] propose a proactive algorithm to split the incoming data traffic between Edge nodes. Schedulix [7] comprises a greedy algorithm to determine both the order and placement of functions over a hybrid public–private cloud, comprising AWS Lambda and OpenFaaS. Optimization problem formulations have been employed for resource provisioning and allocation in Edge and Cloud serverless environments, e.g., [29,30]. NEPTUNE [29] exploits Mixed Integer Programming to place latency-constrained functions on Edge nodes according to user locations. NEPTUNE also takes into account the availability of GPUs for accelerated function execution. Model-driven resource management algorithms based on queuing theory have been also proposed, for example in LaSS [31] to determine the placement of each function and to auto-scale the allocated resources in response to workload dynamics. In a previous work [32], we also studied the problem of scheduling invocation requests in a resource-constrained serverless cluster, and integrated well-known scheduling policies (i.e., shortest-job-first and priority-based FIFO) in the open-source OpenWhisk framework. Differently from this work, we did not support geographical distribution, offloading and migration.

Ascigil et al. [30] consider the more general problem of resource allocation for serverless functions running in an Edge–Cloud environment and propose centralized and decentralized optimization approaches. The scenario they consider is similar to the one targeted in this work, with multiple functions and groups of users. However, they assume that containers for function execution are statically provisioned in the infrastructure, rather than being created and terminated dynamically, and thus do not cope with cold starts. Instead, we define a system model based on the behavior of the most popular FaaS frameworks, including the issues related to dynamic container management.

Some works focus on exploiting function offloading, where a FaaS node, after receiving an invocation request, decides to forward the request to another node instead of serving it. As mentioned, we distinguish between *vertical* offloading, where requests are offloaded to nodes at higher levels of the infrastructure (e.g., from Edge to Cloud), and *horizontal* offloading, where requests are offloaded to nodes at the same infrastructure level (e.g., within the same Edge zone).

As explained above, few FaaS frameworks (e.g., Colony and Serverledge) have built-in support for both horizontal and vertical offloading. Conversely, popular open-source frameworks for the Cloud (e.g., OpenWhisk, OpenFaaS) have no native support for offloading, but researchers have proposed federated systems on top of them with offloading abilities. For example, horizontal offloading is exploited in DFaaS [8], which relies on an overlay network to federate a set of OpenFaaS nodes at the Edge. By means of horizontal offloading, DFaaS manages to balance load among the federated nodes. A similar scenario is studied by Cicconetti et al. [9], who propose an Internet Protocol-inspired algorithm to offload invocation requests within a network of FaaS nodes, and in P2PFaaS [33], a framework for load balancing and scheduling in a peer-to-peer FaaS system.

A few works exploit vertical offloading, usually to forward requests from the edge of the network towards the Cloud. For instance, Das et al. [34] consider the problem of scheduling the execution of serverless pipelines either at the Edge or in the Cloud. The proposed policy allows users to specify latency and cost requirements and determines where to execute the task based on prediction models of the task duration. Deep reinforcement learning (DRL) is used in [35], focusing on a scenario where functions can be offloaded from IoT devices to Edge nodes. They use DRL to minimize the long-term system *latency cost*, a metric computed in terms of function response time and deadline. A game-theoretic approach is presented in [36], which considers the interaction between self-interested wireless devices that can reserve communication and computing resources for latency-sensitive applications, and a FaaS Edge operator that allocates resources for function execution. The authors consider both the case of perfect and imperfect information. Vertical offloading is also exploited in AuctionWhisk [37], an auction-inspired approach integrated in OpenWhisk, which targets a FaaS system running in a Fog computing scenario. The proposed approach relies on auctions where users bid on resources, while FaaS nodes decide locally which functions to execute and which to offload towards the Cloud in order to maximize revenue.

UnFaaSener [38] considers a different offload scenario. Instead of simply offloading requests from a FaaS node to another, UnFaaSener studies how to offload function execution from a serverless platform (e.g., Google Cloud Functions) to a traditional VM, to take advantage of underutilized servers in the user's infrastructure.

The problems of decentralized task scheduling and offloading have been widely studied beyond the specific FaaS application domain, e.g., in Tasklets [39,40], a task-based distributed computing framework. For a comprehensive discussion of task offloading in Edge and Cloud systems, we refer interested readers to [41,42].

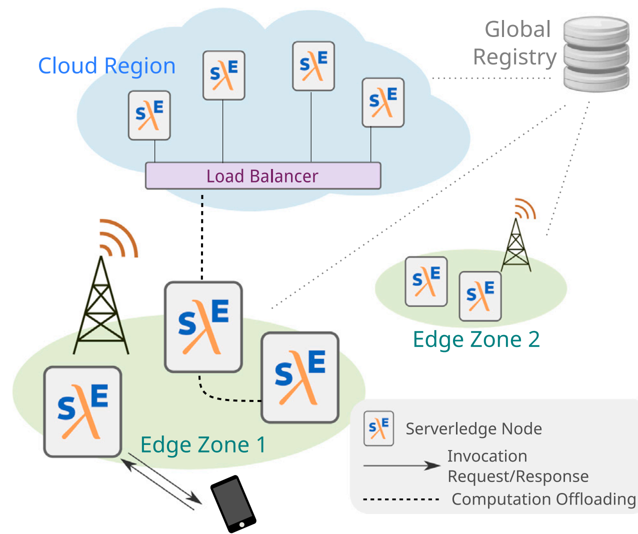


Fig. 1. Serverledge overview.

2.3. Live function migration

To the best of our knowledge, none of the existing FaaS frameworks currently offers live function migration as a built-in feature. Nonetheless, researchers have started to investigates approaches to function migration. Kahrula et al. [43] consider a container checkpointing mechanism, similar to our solution, that can be used both for fault tolerance and migration. Their work specifically targets serverless functions for IoT at the Edge.

Soltani et al. [44] consider function migration to address a limitation of Cloud-based FaaS offerings, which usually limit the maximum execution time of functions. Therefore, they propose a solution that migrates functions close to the execution timeout to a different Cloud provider. However, their solution is not integrated in a general-purpose FaaS framework.

Pelle et al. [45] propose a network-assisted solution to migrate serverless applications at the edge of the network. However, differently from our approach, they do not migrate single function instances during execution, but rather provision the same function to multiple infrastructure nodes and quickly switch between them at run-time.

3. Design of Serverledge

Serverledge is a decentralized FaaS platform, designed for Edge–Cloud computing environments. According to the FaaS paradigm, 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 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.

Fig. 1 illustrates the high-level architecture of a Serverledge installation, which. consists of one or more *nodes*, deployed either in Cloud data centers or at the edge of the network, and a *global registry*. The latter provides the 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.[2] Cloud regions typically represent geo-distributed data centers, while Edge zones may be associated with, e.g., single towns or cities. Each region (especially Cloud regions) may also 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 no 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

[2] Since we do not currently target mobile devices, we assume the location of each node (at the granularity of Cloud region/Edge zone) to be statically configured at deployment time.

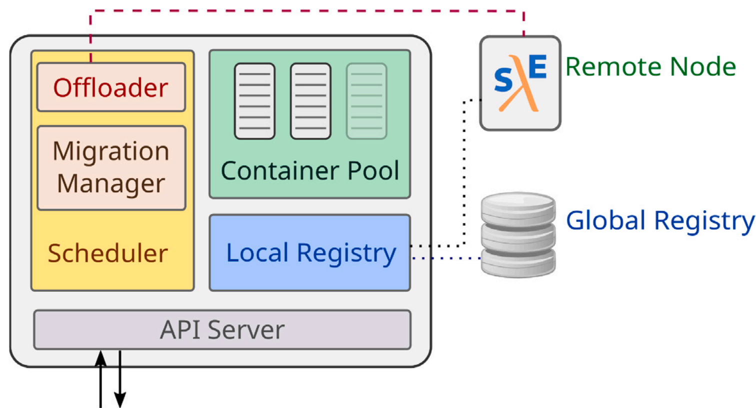


Fig. 2. Architecture of a Serverledge node.

and initialized to execute the function. 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 min 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.

Due to 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.

Furthermore, nodes can also reduce their current load by migrating a function instance that is already being executed. We will provide more information about function live migration in Section 5.

A Serverledge node comprises a few key components, as depicted in Fig. 2: API server, Scheduler, Local Registry, 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.

3.1. API server

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). The information is stored in the Global Registry and, thus, the new function is available at every Serverledge node.
- `/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. The information is retrieved from the Global Registry and possibly cached locally.
- `/delete`: to de-register an existing function. The operation is applied to the Global Registry, causing the function to be deleted system-wise.
- `/status`: to obtain information about a node, including, e.g., the amount of available computational resources and the current state of its container pool.
- `/async`: to obtain the result of an asynchronous function invocation (see details in Section 3.5).

3.2. 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* [46] algorithm to build and update a virtual coordinate space, which allows nodes to estimate the network distance among each other. The Vivaldi algorithm requires 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.

3.3. Function scheduling

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) 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 decides whether, where and how the request is served. The Scheduler is also supported by two submodules, *Offloader* and *Migration Manager*, which are responsible, respectively, for managing offloaded requests and migrated function instances.

As illustrated in Fig. 3, the Scheduler has the following options for every incoming request:

- executing the function locally;
- pushing the request to a local *queue*, for future execution;
- offloading the function to another node, either in the Edge zone or in the Cloud;
- discarding the request and returning it to the client.

The algorithm used to make such a decision for every incoming request represents the *scheduling policy* in use. Serverledge provides an easy-to-extend interface enabling the definition of custom policies (e.g., to differentiate multiple classes of users) and is not bound to a specific algorithm.

Focusing on local execution, 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 depending on the runtime environment required by the function and specified at creation time (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, besides local execution within a warm/cold container, the Scheduler can make other decisions for incoming requests, including adding to the queue, offloading and discarding. Compared to the work presented in [13], we introduced the ability of buffering incoming requests in a local *queue*. The queue provides the Scheduler with an additional freedom degree. By leveraging the queue, the Scheduler can cope with overloading periods either offloading requests or buffering them. Clearly, local buffering introduces additional delay for the request, but it can be useful for specific classes of users (e.g., users who can tolerate higher response times, but do not want their requests to be dropped) or at times when network conditions make offloading less attractive.

3.4. Container management

As explained, and following the approach adopted by most the existing FaaS implementations, we rely on containers to isolate the execution of different functions and different instances of the same function, as they all share the same computing resources provided by the node. The node relies on a *Container Pool* component to manage and track all the containers allocated in the node.

The creation of new containers – as explained above – is triggered by the Scheduler when a request must be executed. New containers are created according to the specifications of the function they will execute. Besides the base image to use, key function attributes used in this phase are the memory and CPU demand of the function. These attributes determine the maximum amount of memory and the quota of CPU time to reserve for the container. These resource limits have a twofold role. On the one hand, they allow the Scheduler to determine whether a new container can be spawned at any time, based on the amount of computing resources consumed by the existing ones. On the other hand, by properly sizing the CPU allocation of the containers, Serverledge has an additional mechanism to differentiate users and functions in the system, similarly to what is done by commercial FaaS platforms, which apply different prices depending on resource allocation.

After executing a function, containers are not terminated and are instead moved to the 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

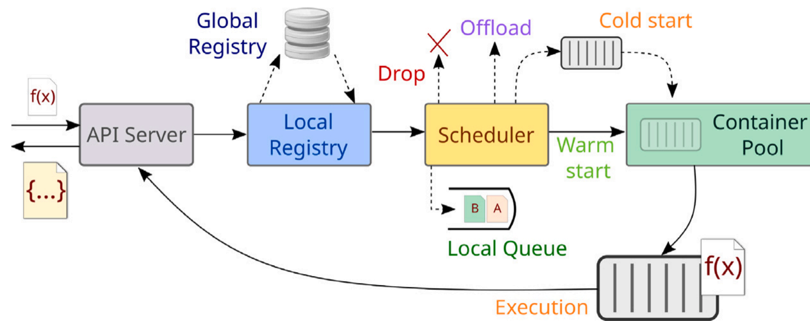


Fig. 3. Illustration of function scheduling. If local execution is not possible (either in a warm or newly created container), the Scheduler can drop the request, offload it to a remote node or push it to a local queue for future execution.

longer than a threshold $T^{expired}$ (whose default value is 10 min), or (ii) we need to create a container for a different function and must reclaim memory from the warm pool to do so.

Regarding the software toolbox used to create and manage software containers in Serverledge, we support both Docker[3] and Podman,[4] allowing users to select the solution to use before starting Serverledge. Compared to Docker, Podman offers some advantages, mostly related to its daemonless design.

3.5. Function execution

Synchronous invocation is the default behavior in Serverledge and the one assumed in Fig. 3. When a function is invoked, the invoking client remains connected to Serverledge waiting for the invocation result to be returned.

Asynchronous function invocation allows clients to invoke functions without being blocked while they are executed. Serverledge takes care of function execution and the client retrieves the execution result (if available) at its convenience. Specifically, when a client sends an asynchronous invocation request, Serverledge immediately replies with an *invocation ID*, a unique identifier of the request. The ID is sent to the client and the connection is closed. Within Serverledge, the request is then forwarded to the Scheduler and executed, with no difference compared to synchronous invocations. However, when the function is executed, instead of sending the results back to the client, they are saved in the Global Registry (and cached in the Local Registry). Later, the client will use the `/async` command to specify the invocation ID and poll for the execution result. The client might also contact a different node to retrieve the result, as it is stored in the Global Registry.

Function execution happens within containers, where the source code of the function has been previously copied. Actually, the container does not directly run the source code of the function when started. Instead, it runs a simple HTTP server, named *Executor* server, whose goal is basically wrapping the execution of the function code. The Executor is kept in execution all the time the container is active, either executing a function or warm, and listens for HTTP requests from the Serverledge node (i.e., it does not listen for connections from outside the node).

When the Scheduler has to start the execution of a function, after obtaining a container from the Container Pool (either warm or newly created), it sends an `/invoke` request to the Executor server in the container, transmitting the invocation parameters specified by the client too. The Executor will send back the execution results produced by the function as an HTTP response.

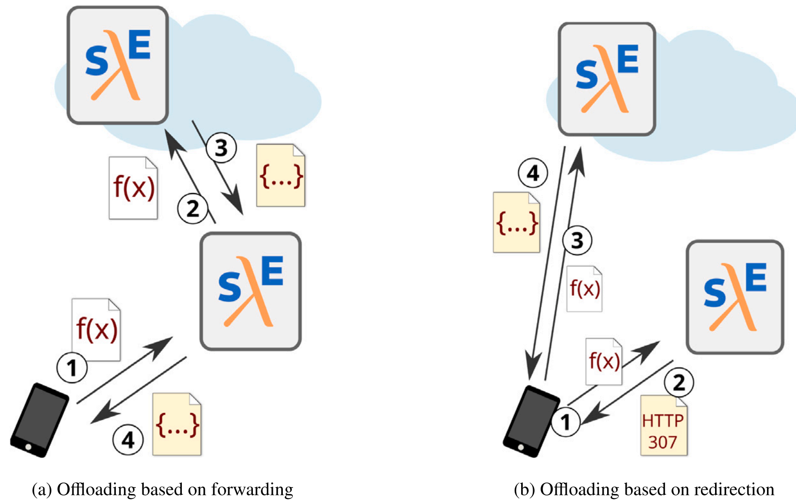
Function invocation can be either synchronous or asynchronous.

4. Function 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 be 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. On the contrary, horizontal offloading involves single nodes in the neighborhood, which do not necessarily offer better performance than the original 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.

[3] https://www.docker.com/
[4] https://podman.io/



(a) Offloading based on forwarding

(b) Offloading based on redirection

Fig. 4. Overview of the offloading mechanisms in Serverledge.

As explained above, the decision of offloading an incoming request is made by the Scheduler component, based, e.g., on the current resource utilization of the node. 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 from a system design perspective. When the Scheduler makes an offloading decision for a request, a remote node for offloading is also selected relying on information in the Local Registry, which includes information on the neighbor Edge nodes and the available Cloud regions (if any).

In this section, we describe two different mechanisms for function offloading that we integrated in Serverledge. The first one relies on the idea of request forwarding, whereas the second exploits HTTP redirection. After presenting the two mechanisms, we will introduce an algorithm to dynamically select the best mechanism to use at run-time. Hereafter, we will use the expression "local node" to refer to the first node receiving the request from the client; and the expression "remote node" to indicate the node to which the request is offloaded.

4.1. Offloading via forwarding

A simple approach to function offloading consists of *forwarding* the incoming request to the remote node and is illustrated in Fig. 4(a). In this scenario, the local node acts as a reverse proxy, receiving the request from the client, and issuing a new invocation request to the API of the remote node. The local node keeps the connection with the client open and waits for the computation result traveling back from the remote node. When the execution result is ready, the local node sends it back to the invoking client as soon as possible.

In principle, offloaded requests incur the same scheduling process on the remote node, although we may 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. Although 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.

An important advantage of this approach is that offloading is completely transparent with respect to the client, which does not observe anything different compared to normal execution (except, possibly, for additional latency). As such, the request might also be offloaded to a Serverledge node that is not directly exposed to and reachable by external clients. A key drawback of this approach is the fact that the local node has to keep the connections to the clients open, even though their invocation requests have been offloaded, incurring some additional overhead.

4.2. Redirect-based offloading

We consider an alternative offloading mechanism with the aim of limiting the involvement of the local node in the handling of offloaded requests as much as possible. Specifically, we exploit the redirection mechanisms provided by the HTTP protocol, as illustrated in Fig. 4(b). When the Scheduler decides to offload a request to another node, the local node immediately replies to the client with a `307 - Temporary Redirect` HTTP response, using the *Location* header to indicate the remote node to contact for a new invocation request. The client, as soon as it receives the response, sends a new invocation request to the remote node. The new request will be viewed and scheduled by the remote node regardless of the offloading process.

This approach relieves the local node that is offloading a request from maintaining an active connection with the client. Moreover, redirecting the client to the remote node can be beneficial in terms of response time if the remote node is closer to the client than

to the local node (e.g., a closer Edge node). However, this approach also has some drawbacks, mostly related to the involvement of the client in the offloading process. Indeed, Serverledge has no guarantees of offloaded requests being actually transmitted to the chosen remote nodes, following a redirection, and, in general, the offloading process is not transparent with respect to clients. Moreover, compared to the forwarding approach, the redirection may result in higher response times when offloading a request to a node that is distant from the client (e.g., the Cloud), since it includes a full round-trip time (RTT) between the client and the remote node.

4.3. Adaptive mechanism selection

As discussed above, both the offloading mechanisms have advantages and disadvantages, not only in terms of performance, but also in terms of client transparency. As such, none can be regarded as the best mechanism to use. Therefore, we aim to devise an algorithm to automatically select the most suitable mechanism to use at run-time in Serverledge.

We focus on performance aspects, with the aim of optimizing the response time observed by the client. In this regard, the network delay is the key factor impacting the response time of offloaded requests, as the two proposed mechanisms cause requests to follow different paths. The idea underpinning our adaptive mechanism selection is to monitor the network delay at run time and use the offloading mechanism that takes the shortest path (in terms of latency).

Let $RTT(a,b)$ denote the measured network round-trip time between nodes a and b and let C, L and R denote, respectively, the client node, the local node (that is, the original target of the invocation request) and the remote node. Let also R_F and R_R denote the expected response time of a request offloaded, respectively, using forwarding or redirection. We readily have:

$$R_F > RTT(C,L) + RTT(L,R) \tag{1}$$

$$R_R > RTT(C,L) + RTT(C,R) \tag{2}$$

According to the expressions above, when offloading a request, we use the following rules to select the offloading mechanism to use:

- if $RTT(C,R) > RTT(L,R)$, we use request forwarding;
- if $RTT(C,R) \leq RTT(L,R)$, we use redirect-based offloading;
- if we cannot currently estimate the RTT based on monitoring information, we opt for a default mechanism (i.e., forwarding for requests offloaded to the Cloud, and redirection for requests offloaded to the Edge).

The proposed adaptive algorithm is very easy to implement and adds minimal overhead to request scheduling. However, it requires information about network delay. Serverledge nodes use Vivaldi algorithm to estimate network distance to neighboring Edge nodes and we exploit this feature in case of Edge offloading. For Cloud offloading, we let nodes periodically measure the network delay through ICMP ping messages.

Unfortunately, our approach also requires information about network delay measurements of the client. For this reason, we extend the CLI-based default client of Serverledge to collect this information during operation. To make measurements available to the node making offloading decisions, we include them in the invocation request message. Clearly, we cannot be sure that all the clients will provide the required information and, thus, the adaptive mechanism selection will be only enabled when possible.

5. Live function migration

Function execution offloading allows incoming requests to be forwarded/redirected to a different Serverledge node before being served. There might be situations where a node has already started executing a function and would like to offload the remaining part of the computation to another node. For instance, this might happen if the node starts executing a function that is expected to complete within a few milliseconds and instead it happens to be a long-running task; or the node might have accepted a large number of low-priority requests and suddenly receives a high-priority invocation, for which resources must be reclaimed. In these situations, the ability of *migrating* the function instances to other nodes allows the node to free up some resources to be better re-allocated.

It is worth observing that the process described above could still be regarded as an example of computational offloading, applied to a portion of the function. However, we prefer to indicate it with a different expression (i.e., live function migration) because the mechanism is profoundly different with respect to the offloading ones introduced above. The key difference regards the entity that is moved across nodes. While in the offloading scenario described in the previous section we can simply move a request message, in this case we need to migrate a whole function instance during execution. Therefore, to preserve the state of the computation, we need to migrate the live container where the function is being executed.

Similarly to virtual machine live migration, the process of migrating a container consists of the following key steps: the container is paused on the source node; a snapshot of the container memory and execution state is created; the snapshot is transferred to the destination node; the container is restored from the snapshot in the destination node. Moving the snapshot of a container is clearly much more expensive in terms of time compared to forwarding a request message, as we will also demonstrate in the experiments.

Mechanisms for live container migration have been proposed in recent years (e.g., *CRIU*), providing the essential technology on which our approach is built. However, although essential, such mechanisms are not sufficient to support live function migration in

a system like Serverledge. First, most the existing techniques do not preserve active network connections. While this issue might be negligible for some applications (e.g., stateless web services, which can simply restart listening for new requests), in our context we can lose the ability of communicating with the running function instance from the host. Moreover, migrating functions poses new challenges related to the interaction with the invoking client. Specifically, we must ensure that the client eventually receives the execution results, regardless of whether the function has been migrated or not. Therefore, we need to design specific migration protocols.

In the remainder of this section, we describe how live function migration is integrated in Serverledge, both for synchronous and asynchronous function invocations.

5.1. Migration of asynchronous requests

We first consider the case where a function has been invoked asynchronously and has to be migrated, as it is simpler to perform. The migration process can be triggered by the Scheduler for any running function. We do not currently consider the case where the migration of a function instance is requested by a client, although the mechanism we present could be adopted in that scenario as well.

As soon as the migration process starts, the Migration Manager notifies the container that it will be migrated soon. For this purpose, we extend the Executor server that runs within every container, adding a `/prepareMigration` command. The Migration Manager uses this command to send an HTTP request to the Executor, where the IP address of the migration destination node is specified. The Executor stores this information that will be necessary later. Then, the container is paused, and a snapshot of the container memory and execution state is created. At this point, the Migration Manager sends a `/migrate` request to the remote node, attaching the snapshot of the container and the ID of the asynchronous request. When receiving this migration request, the node is also informed about the synchronous/asynchronous nature of the invocation.

Upon receiving the snapshot, the remote node restores the container and resumes its execution. As soon as the function code terminates, the Executor server within the container will try to communicate the computation result back to the Serverledge node. However, following the migration, the connection between the (initial) Serverledge node and the Executor server will be lost, and the attempt of the Executor of communicating the results will fail. We intercept this failure and retrieve the IP address received in the `/prepareMigration` to directly send the execution results to the new Serverledge node. Finally, the node will store the execution results in the Registry, as usually done for all asynchronous invocations. The client will be able to poll for the results through any Serverledge node. The protocol described above is illustrated in Fig. 5.

5.2. Migration of synchronous requests

The process of migrating synchronous requests is similar to the one presented above for asynchronous requests, but it requires a few modifications in the end. Indeed, in synchronous invocation requests, the client remains connected to the node, waiting for the execution results. Therefore, we need a way to ensure that, as soon as the execution of the function completes on the remote node, we send back the results to the invoking client, which is still connected to the initial node. For this purpose, when dealing with synchronous requests, we make the `/migrate` request sent by the initial node to the destination one a synchronous one as well. This means that the migration source node remains connected to the remote one waiting for the execution results. When the migrated request is completed and the remote node collects its execution results, it communicates them to the migration source node. The initial node will be able to reply to the invoking client with the results, terminating the invocation request. The resulting protocol is illustrated in Fig. 6.

5.3. Live container migration

The migration protocol presented above relies on the ability of migrating live containers via checkpointing. For this purpose, we rely on *CRIU*[5] (Checkpoint/Restore In Userspace), a software library to freeze running containers and checkpoint their state to disk in Linux. We exploit the integration of CRIU and Podman to migrate Podman containers hosting functions. To do so, it suffices for the Migration Manager to execute a single shell command to create a checkpoint and a single command to restore it. The created checkpoint contains a dump of the container memory and all the relevant information for its migration. The checkpoint does not contain the content of the container disk that can be retrieved from the image used to start the container. As such, it is required that the same image used to start the container in the first node is also available in the destination node. Since Serverledge base images are stored in a public repository, this does not represents an issue for us.

6. Evaluation

In this section, we present a set of experiments aimed at evaluating the performance of Serverledge, focusing on the novel contributions related to offloading and migration of this work. The section is organized as follows. We first describe the experimental setup. Then, we give an overview of the comparison of Serverledge against state-of-the-art FaaS frameworks, presented in [13]. Then, we focus on the evaluation of the novel mechanisms presented in this work, namely offloading mechanisms, live function migration, CPU capping, and request queueing.

[5] https://criu.org/Main_Page

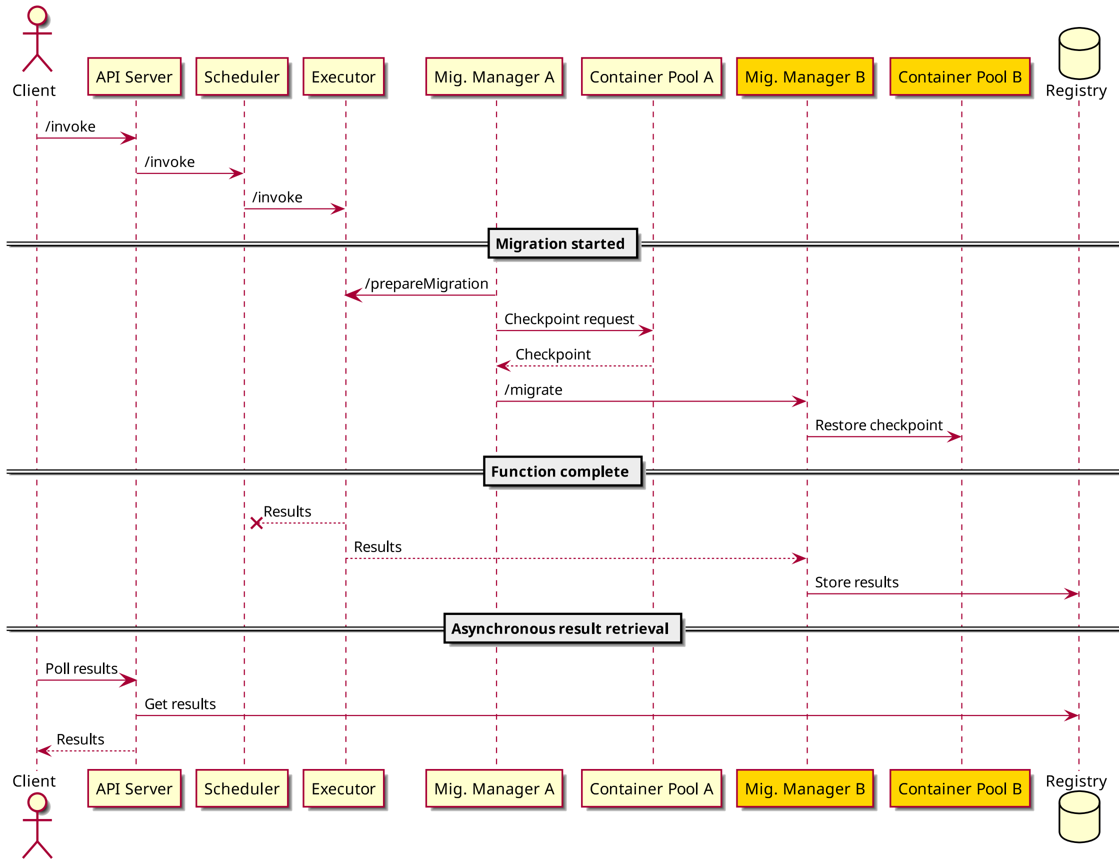


Fig. 5. Migration of asynchronous requests.

6.1. Experimental setup

For the experiments, we exploit virtual machines running in a bare metal server as well as AWS EC2 virtual machines. Except for experiments dealing with specific configurations, 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.

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.

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 the authors of [12].
- `Fibonacci` (Fib, for short): recursive computation of the nth element of the Fibonacci sequence. Implemented in Python and Go.
- `ML-training`: a long-running Python function that trains a neural network.
- `Sleep`: a simple Python function that sleeps for a fixed amount of time before returning.

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

6.2. Comparison against alternative frameworks

We consider three state-of-the-art platforms, namely OpenWhisk, Faasm and tinyFaaS, which have been introduced in Section 2, to compare Serverledge performance. For this comparison, we deploy each framework to a single Edge node, using an additional

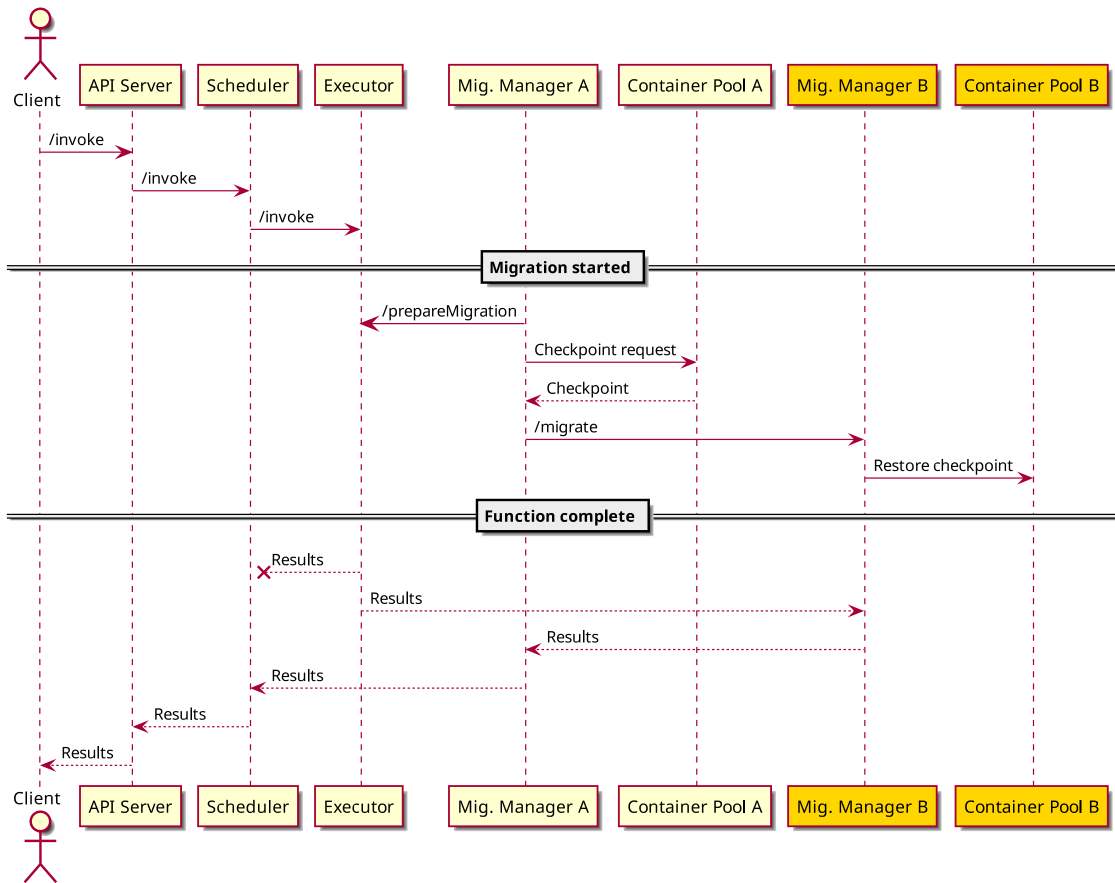


Fig. 6. Migration of synchronous requests.

node used for load generation. As regards OpenWhisk, which is usually deployed in Cloud-based clusters through Kubernetes, we rely on its "lean" deployment mode for Edge scenarios.[6] 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)[7]. 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.[8] Therefore, we perform the comparison against Faasm using a different, reduced workload, which is presented later in this section.

6.2.1. Results with OpenWhisk and tinyFaaS

The results of this comparison are reported in Table 2. Fig. 7 shows the throughput over time, while Fig. 8 compares response time distributions (with whiskers from the 5th to the 95th percentile). We first note that OpenWhisk, which is not designed 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 290 ms, compared to 21 ms 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

[6] https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b
[7] We verified that the workload saturates system capacity by doubling the number of users without observing evident throughput increases.
[8] 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 2
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

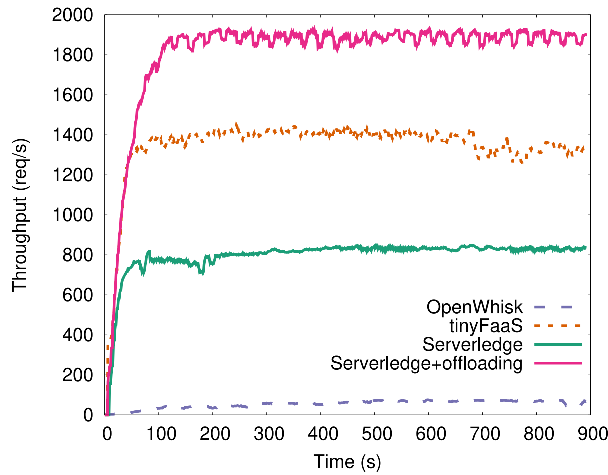


Fig. 7. Throughput of OpenWhisk, tinyFaaS and Serverledge running in a single node. tinyFaaS has the highest throughput among them, but Serverledge can exploit additional nodes to offload and serve more requests.

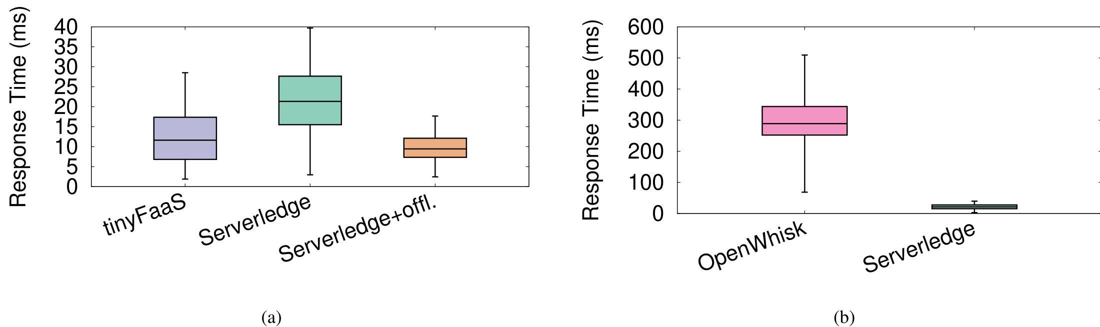
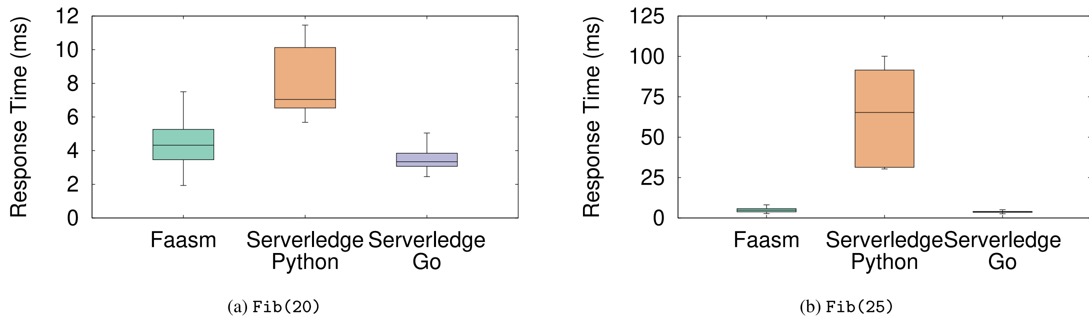


Fig. 8. Response time comparison of (a) tinyFaaS and Serverledge (with and without offloading), and (b) OpenWhisk and Serverledge.

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 2).



(a) `Fib(20)`

(b) `Fib(25)`

Fig. 9. Response time of Faasm and Serverledge running the `Fib` function with different inputs.

Table 3
Comparison of Serverledge and Faasm.

	Thr. (req/s)	Response time (ms)								
		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

6.2.2. *Results with Faasm*

Because of the aforementioned issue affecting Faasm, 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 3, with Fig. 9 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 implementation of the function. In this case, Serverledge shows response times comparable to those measured with Faasm, and even better on average.

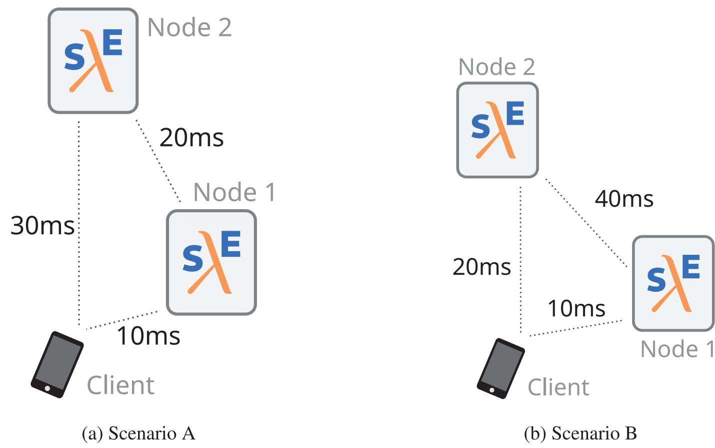
We remark that the benefits of Faasm are still evident looking at the maximum response time in Table 3. Indeed, exploiting lightweight function runtimes, cold starts in Faasm have reduced impact on response time compared to Serverledge. We also plan to consider alternatives to Docker containers for function execution as future work.

6.3. *Offloading mechanisms*

For this experiment, we consider two infrastructure configurations, indicated as *Scenario A* and *Scenario B*, in which we impose different network delay configurations between VMs hosting the Serverlegde nodes and the workload generator, as illustrated in Fig. 10. The client sends invocation requests to the Serverledge node indicated as "Node 1", which offloads them to "Node 2" using one of the available mechanisms (i.e., forwarding, redirection). We first evaluate the offloading mechanisms in the two different scenarios in distinct experiments. Then, we consider a 10-minute experiment where the infrastructure configurations transitions from Scenario A to Scenario B after 5 min.

Fig. 11 compares the behavior of the offloading mechanisms in the two infrastructure scenarios defined above. As expected, network delays have a significant impact on the resulting response times perceived by the client. Indeed, in Scenario A, forwarding is the most convenient offloading mechanism, resulting in about 25% response time reduction compared to redirection. Conversely, in Scenario B, redirection achieves the best performance, with 30% lower response times compared to forwarding. These results confirm that no single mechanism has the best performance in general and one of them should be selected depending on the infrastructure conditions.

Fig. 12 shows the response times measured in the experiments where the infrastructure is dynamically reconfigured from Scenario A to Scenario B at execution time. We consider the two offloading mechanisms and the adaptive configuration where the algorithm we presented in Section 4 is used to select the best mechanism at run time. Using the redirect-based offloading, Serverledge performs overall better compared to forwarding, with the median response time around, respectively, 110 and 220 ms.



(a) Scenario A   (b) Scenario B

Fig. 10. Infrastructure configurations used for offloading evaluation.

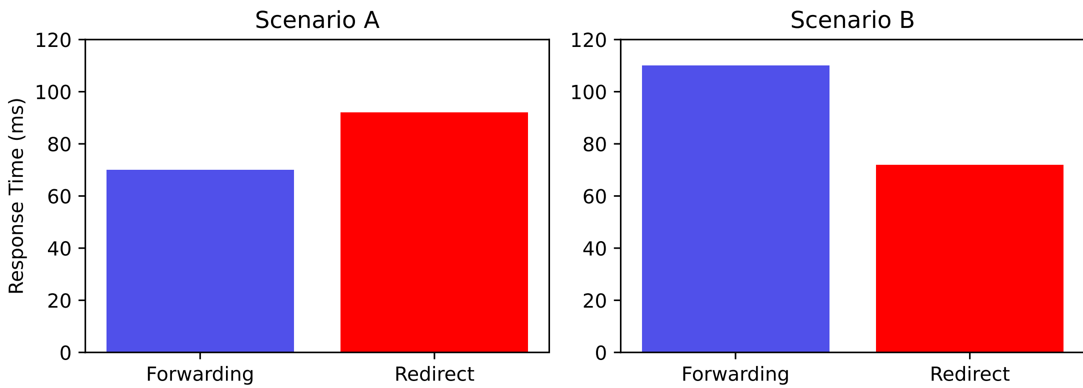


Fig. 11. Response time with different offloading mechanisms and infrastructure configurations.

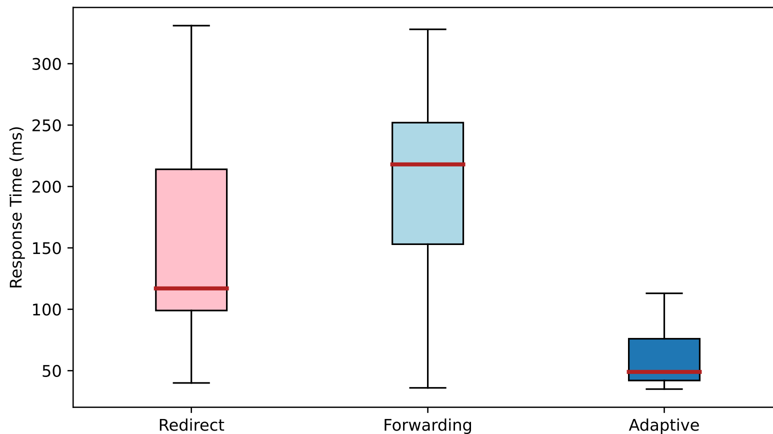


Fig. 12. Response time with different offloading mechanisms in an experiment where the infrastructure is reconfigured at run time (i.e., from Scenario A to Scenario B).

The adaptive mechanism selection significantly outperforms the static solutions, with a median response time lower than 50 ms. This is not surprising, as the adaptive selection manages to always pick a mechanism that performs well depending on the current network delays.

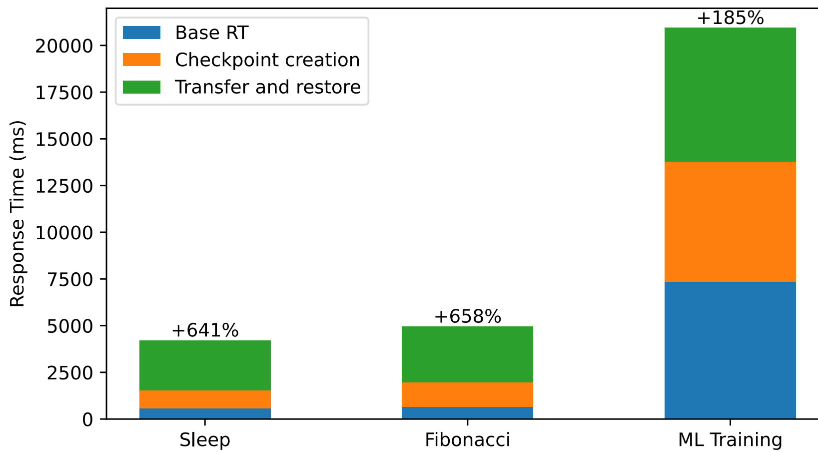


Fig. 13. Response time breakdown for migrated functions.

6.4. Function migration

To evaluate live function migration, we consider 3 functions having different resource demands, namely Fibonacci, ML-training and Sleep, and configure a Serverledge cluster composed of 2 nodes. The first node receives requests from the client, whereas the second one is available as a migration target. We configure the first node to periodically migrate all the executed functions every 500 ms, with the aim of evaluating the overhead caused by migration.

Fig. 13 reports the response time of the three considered functions when migrated. We observe that migration introduces a significant overhead on function response time, up to 680%. The migration overhead clearly does not depend on the total execution time of the function and, hence, is particularly significant for Sleep and Fibonacci, which have shorter duration compared to the long-running ML-training. We also measured the size of the container checkpoint archive generated during migration for the three functions. The checkpoint is about 40 MB large for Sleep, 50 MB for Fibonacci and 250 MB for ML-training, explaining the longer duration of checkpoint creation and restore for ML-training. We also noted that a significant part of the checkpoint transfer time is spent in serialization and de-serialization of the archive. We will consider different implementations in future work to possibly reduce this overhead.

6.5. CPU capping and request queueing

In this experiment we demonstrate the new mechanisms we introduced for CPU allocation and request scheduling. We consider the functions Sieve and Fibonacci introduced above and a single Serverledge node. We let a varying number of users generates requests to the node with a think time of 0.3 s. We compare the case (i) where Serverledge has no buffer for the incoming requests and necessarily discards them if there are not enough resources to serve them, and (ii) where the scheduler can push requests to a queue. Furthermore, we consider three CPU allocation settings, where each function instance receives up to 25%, 50% and 100% of a CPU core.

Figs. 14 and 15 report throughput and response time measured in these experiments, respectively, for the function Sieve and Fibonacci. As regards throughput, we observe that enabling the queue avoids request dropping as expected and leads to higher throughput values. The throughput difference is particularly evident without CPU capping, where the scheduler can accept a lower number of concurrent function instances. Conversely, response times are clearly worsened by the presence of the queue, as incoming requests may be forced to wait in the buffer before being served. Response times without the queue are almost constant regardless of the number of users without CPU capping. Instead, when allocating fractions of the CPU time, a larger number of concurrent instances must be scheduled, thus leading to overhead and higher response times with a large number of users.

6.6. Larger deployments

We conclude the evaluation by considering larger deployments. As explained above, the design of Serverledge aims for decentralization and loose coupling to increase scalability. In particular, nodes are designed with the ability of scheduling and serving requests with minimal or no interaction among them. Scheduling decisions involving multiple nodes (e.g., choosing offloading or migration target nodes) can be restricted to a small number of remote peers based on network proximity, thanks to neighborhood monitoring.

We run experiments to demonstrate the behavior of Serverledge in various infrastructure configurations, where we deploy an increasing number of Edge and Cloud nodes. We start with a single Edge node, and reach a maximum of 17 Serverledge nodes (i.e., 12 Edge nodes, a load balancer and 4 Cloud nodes). We again rely on distinct AWS regions to deploy Edge and Cloud nodes

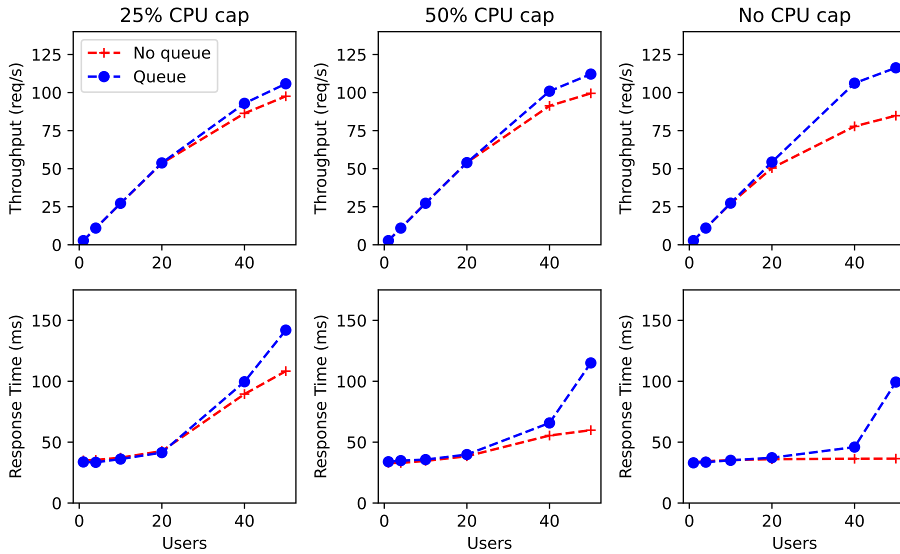


Fig. 14. Throughput and response time of the Sieve function with and without scheduling queue and CPU capping.

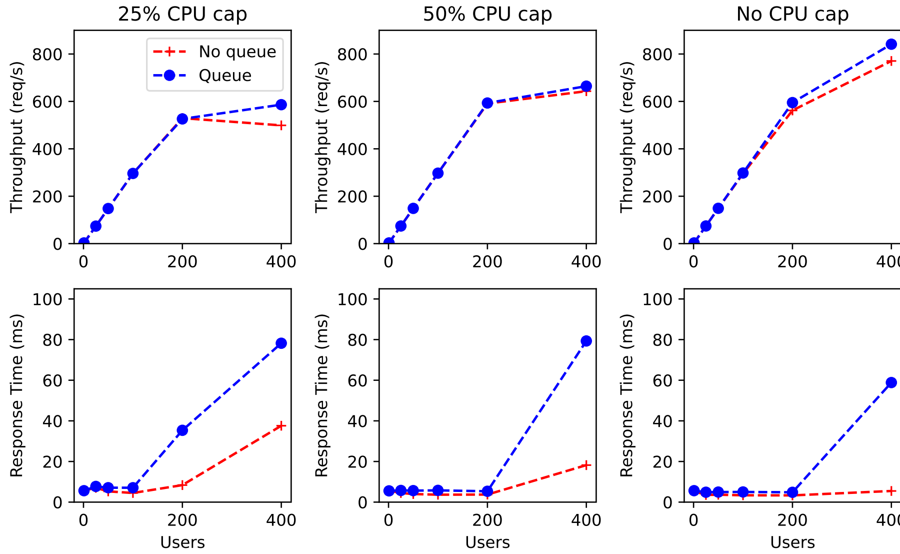


Fig. 15. Throughput and response time of the Fibonacci function with and without scheduling queue and CPU capping.

(see, Section 6.1). A single load balancing instance is deployed in the Cloud region to dispatch incoming requests to Cloud nodes, using a simple round-robin policy. Similarly, a single Etcd server is deployed in the Cloud to act as the Global Registry. We generate about 400 invocation requests per second for the *Sieve* function towards every deployed Edge node, and, thus, scale the generated workload along with the infrastructure. We disable request queueing in these experiments (i.e., requests are either served locally, offloaded, or dropped).

The results of these experiments are shown in Fig. 16. We can observe that system throughput scales well with the increasing workload resource availability, growing from 240 req/s to 4,600 req/s. As already noted, offloading is fundamental for Edge nodes to sustain incoming arrival rates. Indeed, when Cloud is not available, Serverledge manages to successfully serve no more than 62% of the incoming invocations. Conversely, when at least one Cloud node is available for offloading, the system completes more than 99.5% of the incoming requests. Looking at response times (right side of Fig. 16), we observe that, when only the Edge is used, Serverledge returns a response within less than 10 ms for all requests. When Cloud offloading is enabled, as expected, response times become noticeably higher because of network latency between Edge and Cloud in our scenario. Besides this, we note that response times remain almost unchanged with the increasing infrastructure scale.

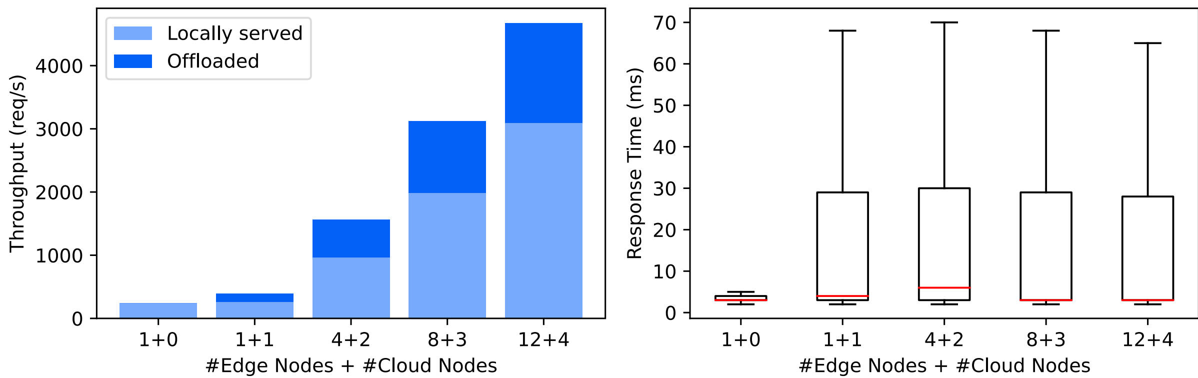


Fig. 16. Throughput and response time with an increasing number of Serverledge nodes.

7. 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. Compared to the previous version of Serverledge described in [13], we introduced new mechanisms for function offloading, live migration, CPU capping and request queueing that enrich the toolbox available to schedule and execute serverless functions. 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. We demonstrated that different function offloading mechanisms should be used depending on the infrastructure conditions, especially in terms of network delay, and proposed an algorithm to dynamically select the mechanism to use at run-time.

The proposed system still has some limitations, that we aim to overcome in future work. Serverledge currently supports the execution of single functions. Therefore, complex workflows involving multiple functions must be orchestrated by the client. Similarly, the system has no built-in support for *stateful* functions and, hence, any state information must be stored in external data stores. Moreover, Serverledge does not currently include user authentication and authorization mechanisms, which would be necessary to restrict function modification and invocation to specific classes of users.

For future work, besides overcoming the aforementioned limitations, we will consider the integration of lighter function sandboxing techniques, following the approach adopted, e.g., by [10,11], to reduce initialization times and cold start impact.

CRediT authorship contribution statement

Gabriele Russo Russo: Conceptualization, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing, Investigation. **Valeria Cardellini:** Conceptualization, Funding acquisition, Methodology, Supervision, Writing – review & editing. **Francesco Lo Presti:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work has been partially supported by the Spoke 1 “FutureHPC & BigData” of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC), Italy funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di “campioni nazionali” di R&S (M4C2-19) - Next Generation EU (NGEU). The authors thank Mary Ann Vasquez Rodriguez and Matteo Ferretti for their contribution to the experimental evaluation.

References

[1] Serverless architecture market size and forecast, in: Verified Market Research, 2022, https://www.verifiedmarketresearch.com/product/serverless-architecture-market/.
[2] S. Kounev, N. Herbst, C.L. Abad, A. Iosup, I. Foster, P. Shenoy, O. Rana, A.A. Chien, Serverless computing: What it is and what it is not? Commun. ACM 66 (9) (2023) 80–92, http://dx.doi.org/10.1145/3587249.
[3] M.S. Aslanpour, A.N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S.S. Gill, R. Gaire, S. Dustdar, Serverless edge computing: Vision and challenges, in: Proc. of 2021 Australasian Computer Science Week Multiconference, ACSW '21, ACM, 2021, http://dx.doi.org/10.1145/3437378.3444367.
[4] R. Xie, Q. Tang, S. Qiao, H. Zhu, F.R. Yu, T. Huang, When serverless computing meets edge computing: Architecture, challenges, and open issues, IEEE Wirel. Commun. 28 (5) (2021) 126–133, http://dx.doi.org/10.1109/MWC.001.2000466.
[5] G. Russo Russo, V. Cardellini, F. Lo Presti, Serverless functions in the cloud–edge continuum: Challenges and opportunities, in: Proc. of 31st Euromicro Int'l Conference on Parallel, Distributed and Network-Based Processing, PDP '23, IEEE, 2023, pp. 321–328, http://dx.doi.org/10.1109/PDP59025.2023.00056.
[6] M. Satyanarayanan, The emergence of edge computing, Computer 50 (1) (2017) 30–39, http://dx.doi.org/10.1109/MC.2017.9.
[7] A. Das, A. Leaf, C.A. Varela, S. Patterson, Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications, in: Proc. of IEEE 13th Int'l Conference on Cloud Computing, CLOUD '20, IEEE, 2020, pp. 609–618, http://dx.doi.org/10.1109/CLOUD49709.2020.00090.
[8] M. Ciavotta, D. Motterlini, M. Savi, 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, http://dx.doi.org/10.1109/CloudNet53349.2021.9657141.
[9] C. Cicconetti, M. Conti, A. Passarella, A decentralized framework for serverless edge computing in the Internet of Things, IEEE Trans. Netw. Serv. Manag. 18 (2) (2021) 2166–2180, http://dx.doi.org/10.1109/TNSM.2020.3023305.
[10] S. Shillaker, 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, https://www.usenix.org/system/files/atc20-shillaker.pdf.
[11] P.K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, G Parmer, Sledge: A serverless-first, light-weight wasm runtime for the edge, in: Proc. of 21st Int'l Middleware Conference, Middleware '20, ACM, 2020, pp. 265–279, http://dx.doi.org/10.1145/3423211.3425680.
[12] T. Pfandzelter, 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, http://dx.doi.org/10.1109/ICFC49376.2020.00011.
[13] G. Russo Russo, T. Mannucci, V. Cardellini, F. Lo Presti, Serverledge: Decentralized function-as-a-service for the edge-cloud continuum, in: Proc. of 2023 IEEE International Conference on Pervasive Computing and Communications, PerCom '23, 2023, pp. 131–140, http://dx.doi.org/10.1109/PERCOM56429.2023.10099372.
[14] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, M. Guo, The serverless computing survey: A technical primer for design architecture, ACM Comput. Surv. 54 (10s) (2022) 1–34, http://dx.doi.org/10.1145/3508360.
[15] A. Mampage, S. Karunasekera, R. Buyya, A holistic view on resource management in serverless computing environments: taxonomy, and future directions, ACM Comput. Surv. 54 (11s) (2022) 1–36, http://dx.doi.org/10.1145/3510412.
[16] G.S. Cassel, V. Rodrigues, R. da Rosa Righi, M. Rosecler Bez, A.C. Nepomuceno, C. André da Costa, Serverless computing for Internet of Things: A systematic literature review, Future Gener. Comput. Syst. 128 (2022) 299–316, http://dx.doi.org/10.1016/j.future.2021.10.020.
[17] J. Wen, Z. Chen, X. Jin, X. Liu, Rise of the planet of serverless computing: A systematic review, ACM Trans. Softw. Eng. Methodol. 32 (5) (2023) 131:1–131:61, http://dx.doi.org/10.1145/3579643.
[18] H. Shafiei, A. Khonsari, P. Mousavi, Serverless computing: A survey of opportunities, challenges, and applications, ACM Comput. Surv. 54 (11s) (2022) 239:1–239:32, http://dx.doi.org/10.1145/3510611.
[19] C. Cicconetti, M. Conti, A. Passarella, FaaS execution models for edge applications, Pervasive Mob. Comput. 86 (2022) 101689, http://dx.doi.org/10.1016/j.pmcj.2022.101689.
[20] F. Lordan, D. Lezzi, R.M. Badia, Colony: Parallel functions as a service on the cloud–edge continuum, in: Proc. of 27th Int'l Conference on Parallel and Distributed Computing, Euro-Par '21, in: LNCS, vol. 12820, Springer, 2021, pp. 269–284, http://dx.doi.org/10.1007/978-3-030-85665-6_17.
[21] R.M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, R. Sirvent, COMP Superscalar, an interoperable programming framework, SoftwareX 3-4 (2015) 32–36, http://dx.doi.org/10.1016/j.softx.2015.10.004.
[22] Z. Li, R. Chard, Y.N. Babuji, B. Galewsky, T.J. Skluzacek, K. Nagaitsev, A. Woodard, B. Blaiszik, J. Bryan, D.S. Katz, I.T. Foster, K. Chard, funcX: Federated function as a service for science, IEEE Trans. Parallel. Distrib. Syst. 33 (12) (2022) 4948–4963, http://dx.doi.org/10.1109/TPDS.2022.3208767.
[23] X. Lyu, L. Cherkasova, R. Aitken, G. Parmer, 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, pp. 49–54, http://dx.doi.org/10.1145/3517206.3526274.
[24] A. Garbugli, A. Sabbioni, A. Corradi, P. Bellavista, TEMPOS: QoS management middleware for edge cloud computing FaaS in the Internet of Things, IEEE Access 10 (2022) 49114–49127, http://dx.doi.org/10.1109/ACCESS.2022.3173434.
[25] A. Hall, U. Ramachandran, An execution model for serverless functions at the edge, in: Proc. of Int'l Conference on Internet of Things Design and Implementation, IoTDI '19, ACM, 2019, pp. 225–236, http://dx.doi.org/10.1145/3302505.3310084.
[26] P. Gackstatter, P.A. Frangoudis, S. Dustdar, Pushing serverless to the edge with WebAssembly runtimes, in: Proc. of 22nd IEEE Int'l Symposium on Cluster, Cloud and Internet Computing, CCGrid '22, IEEE, 2022, pp. 140–149, http://dx.doi.org/10.1109/CCGrid54584.2022.00023.
[27] L. Liu, H. Tan, S.H.-C. Jiang, Z. Han, X.-Y. Li, H. Huang, Dependent task placement and scheduling with function configuration in edge computing, in: Proc. of 2019 IEEE/ACM 27th Int'l Symposium on Quality of Service, IWQoS '19, IEEE, 2019, pp. 1–10, http://dx.doi.org/10.1145/3326285.3329055.
[28] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, A.Y. Zomaya, Dependent function embedding for distributed serverless edge computing, IEEE Trans. Parallel Distrib. Syst. 33 (10) (2022) 2346–2357, http://dx.doi.org/10.1109/TPDS.2021.3137380.
[29] L. Baresi, D. Hu, G. Quattrocchi, L. Terracciano, NEPTUNE: Network- and GPU-aware management of serverless functions at the edge, in: Proc. of 17th Int'l Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '22, IEEE, 2022, pp. 144–155, http://dx.doi.org/10.1145/3524844.3528051.
[30] O. Ascigil, A. Tasiopoulos, T.K. Phan, V. Sourlas, I. Psaras, G. Pavlou, Resource provisioning and allocation in function-as-a-service edge-clouds, IEEE Trans. Serv. Comput. 15 (4) (2022) 2410–2424, http://dx.doi.org/10.1109/TSC.2021.3052139.
[31] B. Wang, A. Ali-Eldin, P. Shenoy, LaSS: Running latency sensitive serverless computations at the edge, in: Proc. of 30th Int'l Symposium on High-Performance Parallel and Distributed Computing, HPDC '21, ACM, 2021, pp. 239–251, http://dx.doi.org/10.1145/3431379.3460646.
[32] G. Russo Russo, A. Milani, S. Iannucci, V. Cardellini, Towards QoS-aware function composition scheduling in apache openwhisk, in: Proc. of 1st Workshop on Serverless Computing for Pervasive Cloud-Edge-Device Systems and Services, *LESS 2022, IEEE, 2022, pp. 693–698, http://dx.doi.org/10.1109/PerComWorkshops53856.2022.9767299.
[33] G. Proietti Mattia, R. Beraldi, P2PFaaS: A framework for FaaS peer-to-peer scheduling and load balancing in fog and edge computing, SoftwareX 21 (2023) 101290, http://dx.doi.org/10.1016/j.softx.2022.101290.
[34] A. Das, S. Imai, S. Patterson, M.P. Wittie, Performance optimization for edge-cloud serverless platforms via dynamic task placement, in: Proc. of 20th IEEE/ACM Int'l Symposium on Cluster, Cloud and Internet Computing, CCGrid '20, IEEE, 2020, pp. 41–50, http://dx.doi.org/10.1109/CCGrid49817.2020.00-89.

[35] X. Yao, N. Chen, X. Yuan, P. Ou, Performance optimization of serverless edge computing function offloading based on deep reinforcement learning, Future Gener. Comput. Syst. 139 (2023) 74–86, http://dx.doi.org/10.1016/j.future.2022.09.009.

[36] F. Tütüncüoglu, S. Josilo, G. Dán, Online learning for rate-adaptive task offloading under latency constraints in serverless edge computing, IEEE/ACM Trans. Netw. 31 (2) (2023) 695–709, http://dx.doi.org/10.1109/TNET.2022.3197669.

[37] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, L. Thamsen, AuctionWhisk: Using an auction-inspired approach for function placement in serverless fog platforms, Softw. Pract. Exp. 52 (5) (2022) 1143–1169, http://dx.doi.org/10.1002/spe.3058.

[38] G. Sadeghian, M. Elsakhawy, M. Shahrad, J. Hattori, M. Shahrad, UnFaaSener: Latency and cost aware offloading of functions from serverless platforms, in: Proc. of 2023 USENIX Annual Technical Conference, USENIX Association, 2023, pp. 879–896, https://www.usenix.org/conference/atc23/presentation/sadeghian.

[39] D. Schäfer, J. Edinger, S. VanSyckel, J.M. Paluska, C. Becker, Tasklets: Overcoming heterogeneity in distributed computing systems, in: Proc. of 36th IEEE Int'l Conference on Distributed Computing Systems Workshops, ICDCS Workshops, IEEE Computer Society, 2016, pp. 156–161, http://dx.doi.org/10.1109/ICDCSW.2016.22.

[40] M. Breitbach, J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, C. Becker, Voltaire: precise energy-aware code offloading decisions with machine learning, in: Proc. of 19th IEEE Int'l Conference on Pervasive Computing and Communications, PerCom '21, IEEE, 2021, pp. 1–10, http://dx.doi.org/10.1109/PERCOM50583.2021.9439121.

[41] F. Saeik, M. Avgeris, D. Spatharakis, N. Santi, D. Dechouniotis, J. Violos, A. Leivadeas, N. Athanasopoulos, N. Mitton, S. Papavassiliou, Task offloading in edge and cloud computing: A survey on mathematical, artificial intelligence and control theory solutions, Comput. Netw. 195 (2021) 108177, http://dx.doi.org/10.1016/j.comnet.2021.108177.

[42] B. Kar, W. Yahya, Y. Lin, A. Ali, Offloading using traditional optimization and machine learning in federated cloud–edge–fog systems: A survey, IEEE Commun. Surv. Tutor. 25 (2) (2023) 1199–1226, http://dx.doi.org/10.1109/COMST.2023.3239579.

[43] P. Karhula, J. Janak, H. Schulzrinne, Checkpointing and migration of IoT edge functions, in: Proc. of 2nd Int'l Workshop on Edge Systems, Analytics and Networking, EdgeSys '19, ACM, 2019, pp. 60–65, http://dx.doi.org/10.1145/3301418.3313947.

[44] B. Soltani, A. Ghenai, N. Zeghib, A migration-based approach to execute long-duration multi-cloud serverless functions, in: Proc. of 3rd Int'l Conference on Advanced Aspects of Software Engineering, ICAASE '18, 2326 of CEUR Workshop Proceedings, 2018, pp. 42–50, https://ceur-ws.org/Vol-2326/paper5.pdf.

[45] I. Pelle, F. Paolucci, B. Sonkoly, F. Cugini, P4-assisted seamless migration of serverless applications towards the edge continuum, Future Gener. Comput. Syst. 146 (2023) 122–138, http://dx.doi.org/10.1016/j.future.2023.04.010.

[46] R. Cox, F. Dabek, M.F. Kaashoek, J. Li, R.T. Morris, Practical, distributed network coordinates, ACM SIGCOMM Comput. Commun. Rev. 34 (1) (2004) 113–118, http://dx.doi.org/10.1145/972374.972394.