Contents lists available at ScienceDirect

Future Generation Computer Systems

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

QoS-aware offloading policies for serverless functions in the Cloud-to-Edge continuum

Gabriele Russo Russo [*], Daniele Ferrarelli, Diana Pasquali, Valeria Cardellini, Francesco Lo Presti

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

A R T I C L E I N F O

Keywords:
Serverless computing
Offloading
Edge computing
Quality of service

A B S T R A C T

Function-as-a-Service (FaaS) paradigm is increasingly attractive to bring the benefits of serverless computing to the edge of the network, besides traditional Cloud data centers. However, FaaS adoption in the emerging Cloud-to-Edge Continuum is challenging, mostly due to geographical distribution and heterogeneous resource availability. This emerging landscape calls for effective strategies to trade off low latency at the edge of the network with Cloud resource richness, taking into account the needs of different functions and users.

In this paper, we present QoS-aware offloading policies for serverless functions running in the Cloud-to-Edge continuum. We consider heterogeneous functions and service classes, and aim to maximize utility given a monetary budget for resource usage. Specifically, we introduce a two-level approach, where (i) FaaS nodes rely on a randomized policy to schedule every incoming request according to a set of probability values, and (ii) periodically, a linear programming model is solved to determine the probabilities to use for scheduling. We show by extensive simulation that our approach outperforms alternative approaches in terms of generated utility across multiple scenarios. Moreover, we demonstrate that our solution is computationally efficient and can be adopted in large-scale systems. We also demonstrate the functionality of our approach through a proof-of-concept experiment on an open-source FaaS framework.

1. Introduction

The Function-as-a-Service (FaaS) paradigm allows developers to deploy modular code units (i.e., *functions*), written in the programming language of their choice, and execute them on demand in a serverless fashion, without the need of provisioning and managing the underlying computing infrastructure. It is up to the FaaS provider to provision the necessary computing resources anytime a function is invoked. This is usually done by creating a software container in a node of the provider's infrastructure, providing an ephemeral, isolated environment to execute an instance of the function. Compared to traditional approaches, FaaS enables faster and seamless elasticity, as a large number of function instances can be spawned when necessary to cope with load peaks. Moreover, users are only billed for the actual amount of computing resources used to execute their functions (e.g., CPU time, memory), eliminating costs associated with idle periods.

The popularity of FaaS has been consistently growing within the Cloud ecosystem, with all the major Cloud providers now offering FaaS services (e.g., AWS Lambda, Google Cloud Functions, Azure Functions). The rapid proliferation of computational resources out of Cloud data centers, in the Cloud-to-Edge continuum, has naturally led to a growing interest in adopting FaaS at the edge of the network [1–3]. By doing so, developers can leverage the benefits of serverless functions for geographically distributed, pervasive and possibly latency-sensitive applications. However, the adoption of FaaS in the continuum poses several challenges [4], mostly related to limited and heterogeneous resource availability, bandwidth-constrained and less reliable network connectivity, privacy, and security concerns.

The research community has started investigating specific solutions to ease the adoption and improve the performance of FaaS in the Cloud-to-Edge continuum. Novel frameworks have been proposed 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), or decentralized architectures to overcome the limitations of Cloud-oriented frameworks (e.g., Colony [7], Serverledge [8], both designed for the continuum). Although these frameworks significantly reduce the gap towards seamless adoption of FaaS at the edge of the network, various key challenges remain about how to optimally allocate and schedule the limited computing resources available in the continuum, while meeting

* Corresponding author.
E-mail addresses: gabriele.russo.russo@uniroma2.it (G. Russo Russo), daniele.ferrarelli@alumni.uniroma2.eu (D. Ferrarelli), diana.pasquali@alumni.uniroma2.eu (D. Pasquali), cardellini@ing.uniroma2.it (V. Cardellini), lopresti@info.uniroma2.it (F.L. Presti).

https://doi.org/10.1016/j.future.2024.02.019
Received 31 July 2023; Received in revised form 20 February 2024; Accepted 22 February 2024
Available online 23 February 2024

Quality of Service (QoS) requirements given by users. Indeed, despite initial efforts in this direction (e.g., [9–12]), we still lack strategies to effectively enjoy both the low latency of Edge and the resource richness of Cloud, taking into account the needs of different functions and users. In particular, function offloading provides a fundamental mechanism to Edge nodes to forward some of the incoming invocation requests to (likely) more powerful yet possibly distant nodes. However, devising a policy to properly decide when, where, and which requests should be offloaded remains an open research topic.

In this paper, we present a novel approach to compute QoS-aware offloading policies for serverless functions running in the Cloud-to-Edge continuum. Our approach considers heterogeneous functions and service classes, each associated with different performance requirements and a measure of their utility as perceived by users. Our approach is designed to maximize the utility generated over time, given a monetary budget for resource usage. We propose a two-level scheme to achieve this goal. First, we adopt a randomized approach to schedule every incoming request, possibly deciding to offload it to the Cloud or to neighboring Edge nodes according to a set of offloading probability values for every function and service class. Second, in order to optimize the overall utility and ensure that QoS requirements of different functions and user classes are met, we periodically recompute the offloading probabilities by solving a suitable optimization problem which takes the form of a simple and efficient linear programming (LP) model. This model takes into account current resource availability, expected workload, and QoS requirements for different functions and user classes.

Our key contributions can be summarized as follows:

- We present an approach to serverless function offloading in the Cloud-to-Edge continuum, which considers heterogeneous functions and QoS classes (Sections 3–4). To this end, we rely on a two-level scheme, where (i) each FaaS node runs a low-overhead randomized policy to schedule every incoming request (Section 5.1), parameterized by given offloading probabilities; (ii) on a larger time-scale, the probabilities in use by each FaaS node are optimized and updated.
- We formulate the problem of determining the offloading probabilities for each FaaS node, function and service class as an efficient linear programming problem (Sections 5.2–5.3), where we maximize the expected utility generated by each FaaS node, accounting for constraints on resource capacity and monetary cost. We also integrate various strategies to estimate cold start occurrence (Section 5.4) and to deal with limited memory capacity of FaaS nodes (Section 5.5) in the LP formulation.
- We evaluate our approach through extensive simulations and compare it with alternative approaches (Section 6). Our results show that our QoS-aware offloading policy consistently outperforms baselines in terms of generated utility and Cloud usage costs across multiple and different scenarios. Moreover, we demonstrate that our solution is computationally efficient and can be adopted in large-scale systems with limited overhead.
- We present a proof-of-concept implementation of the QoS-aware policy in Serverledge [8], an open-source FaaS framework proposed by our group, and demonstrate its applicability through prototype experiments (Section 7).

2. Related work

The increasing popularity of FaaS has attracted significant interest from the research community over the last years, as surveyed in [13–15]. Recently, we observed a surge of interest related to running serverless functions beyond traditional Cloud data centers [1,2], bringing functions closer to devices thanks to the emerging Cloud-to-Edge continuum (e.g., to handle IoT workloads [3]). However, adopting the FaaS paradigm at the edge of the network poses new or renewed challenges, mainly due to resource limitations and geographic distribution [4].

In this section, we narrow our attention to research efforts that explicitly cope with FaaS deployment and execution in the Cloud-to-Edge continuum. We first give an overview of the approaches to ease FaaS adoption in this computing environments, and then focus on the issues associated with function offloading.

2.1. FaaS in the Cloud-to-Edge continuum

Adopting FaaS is currently possible through commercial Cloud offerings (e.g., AWS Lambda) or open-source frameworks (e.g., OpenWhisk, OpenFaaS, KNative) that can be installed in on-premises infrastructures. However, when targeting geographically distributed environments for FaaS deployment, there is still little or no support from Cloud providers (e.g., AWS offers Lambda@Edge, which has some restrictions compared to AWS Lambda though) to properly exploit computing resources in the Cloud-to-Edge continuum. Similarly, the most popular open-source frameworks have been explicitly designed with clustered or Cloud-based deployments in mind, often relying on centralized schedulers or gateway components. For these reasons, researchers have recently proposed novel frameworks (e.g., [5–8,16,17]) to overcome these shortcomings.

Colony [7] is a framework for FaaS in the Cloud-Edge continuum, especially targeting parallel computations. Colony lets nodes process data on their resources, while also offering their computing capacity to the rest of the infrastructure. Differently from most the existing FaaS frameworks, Colony can transparently convert the logic of complex user-given functions into task-based workflows backing on task-based programming models. The generated workflows are then executed over the infrastructure, possibly offloading tasks both horizontally and vertically.

Serverledge [8] is an open-source FaaS framework developed within our group, which relies on a decentralized design to run serverless functions in Cloud-to-Edge environments. Similarly to Colony, Serverledge has built-in support for function offloading, both from Edge to Cloud and within Edge neighborhoods. Serverledge relies on Docker containers for function execution.

Designed for scalable and high performance remote function execution, $f$uncX [17] is a distributed FaaS framework that decouples cloud-based management functionality from edge-hosted function execution, supporting multiple runtime environments. Compared to Colony and Serverledge, management and scheduling duties remain in the Cloud.

Some systems (e.g., [5,6]) focus on the integration of lightweight function sandboxing mechanisms to better suit resource-constrained deployments. Faasm [5] is an open-source research prototype that introduced *Faaslets*, an isolation mechanism 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. Based 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, it does not explicitly consider geographically distributed nodes.

Similarly, Sledge [6,18] is another FaaS framework specifically designed for Edge environments, also exploiting SFI and WebAssembly-based runtime environments for lightweight isolation. Sledge can orchestrate and schedule the execution of function compositions through QoS-aware policies. However, to the best of our knowledge, Sledge only targets single-node deployments and lacks the ability to exploit additional hosts in the Cloud or in the continuum. The same limitation affects tinyFaaS [16], a research prototype that relies on traditional Docker containers for function isolation, but – to reduce overheads – relies on static container pools, which must be configured upfront for each function.

2.2. FaaS scheduling and offloading

The limited and possibly heterogeneous availability of computational resources in Edge environments requires careful planning of resource allocation to multiple functions and users, especially in presence of QoS requirements to meet. As such, several solutions have been proposed to place and manage serverless functions and applications at the edge of the network, also considering the integration with Cloud serverless services and platforms.

Strategies to schedule function execution across heterogeneous and possibly resource-constrained Edge servers have been considered in a number of works (e.g., [19–25]). 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. [22] propose a proactive algorithm to split the data traffic between Edge nodes. Schedulix [20] comprises a greedy algorithm to determine both the order and placement of functions in a hybrid public–private cloud. Costless [21] can fuse multiple functions to form a single function appropriately for cost reduction before placing it on Edge and Cloud nodes. NEPTUNE [23] exploits Mixed Integer Programming to dynamically place latency-constrained functions on Edge nodes according to users' locations, by avoiding their saturation and exploiting GPUs if available. Model-driven resource management algorithms based on queueing theory have also been proposed, for example, in LaSS [24] to determine the placement of each function and to auto-scale the allocated resources in response to workload dynamics. Peri et al. [25] propose a two-level scheduling approach, where the first level determines whether to schedule in a public Cloud or private Edge according to a simple cost-based heuristic, while the second level makes placement decisions on Edge nodes with the overall goal to satisfy both application and system requirements.

Ascigil et al. [26] 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 (e.g., [7–10,27–32]), where a FaaS node, after receiving an invocation request, decides to forward the request to another node instead of serving it. Offloading can be motivated by a variety of reasons, including resource availability, load balancing, energy efficiency, and should be carefully leveraged to avoid negative impact on QoS. 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).

Among the frameworks described above, we remark that Colony and Serverledge support both horizontal and vertical offloading, whereas Faasm only supports horizontal offloading. Conversely, popular open-source frameworks for the Cloud (e.g., OpenWhisk, OpenFaaS) have no native support for offloading, although researchers have built federated systems on top of them with offloading abilities. For example, horizontal offloading is exploited in DFaaS [27], 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. [10], who propose an Internet Protocol-inspired algorithm to offload invocation requests within a network of FaaS nodes.

A few works exploit vertical offloading, usually to forward requests from the edge of the network towards the Cloud. For instance, Das et al. [9] consider the problem of scheduling the execution of serverless pipelines either at the Edge or in the Cloud. The proposed approach allows users to specify latency and cost requirements and determines where to execute the task based on prediction models of the task duration. We will compare our policy to this approach, adapting it to our scenario.

Reinforcement learning has been recently exploited in few works [28,32] to drive offloading decisions of serverless functions using a distributed architecture. In [28], a deep reinforcement learning (DRL) approach for function offloading is proposed, 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, similar to the utility metric we will introduce in the following. AttentionFunc [32] is an approach for distributed function offloading in the Edge-Cloud continuum. It is based on multi-agent DRL, which determines whether to execute functions on an Edge node or in the Cloud with the goal to optimize the function completion time and cost. However, differently from our approach, these two works do not consider multiple classes of users and horizontal offloading and evaluate the proposed policy only by means of simulation. Most importantly, relying on DRL, their solutions require computationally intensive training for each node in the system. Instead, we look for a lightweight solution that can be adopted in large-scale FaaS systems.

A game-theoretic approach to address offloading is presented in [29], 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. Another game-theoretic approach in the form of a Stackelberg game is proposed in [33] to model the interaction between a profit-maximizing FaaS Edge operator, that decides the price, resource allocation and set of functions to cache, and cost-minimizing wireless devices, which decide whether to offload their functions. We observe that these game-theoretic approaches, although interesting, suffer from the limitation of considering only vertical offloading and to a single Edge node.

Vertical offloading is also exploited in AuctionWhisk [30], 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 [31] 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.

For a comprehensive discussion of task offloading in Edge and Cloud systems, beyond the specific FaaS domain, we refer interested readers to [34,35].

3. System architecture

We consider a FaaS system comprising several distributed computing nodes spread across the Cloud-to-Edge continuum. As depicted in Fig. 1, each FaaS node is located either in a Cloud data center or within an Edge zone[1] (e.g., an urban area).

Inspired by emerging FaaS frameworks for the Edge (e.g., [5,8,16]), we assume that each node is capable of autonomously scheduling and serving function invocation requests, without centralized schedulers or gateways.

[1] We use the expression "Edge zone" in a broad sense to simplify terminology, possibly referring to different types of computing environments (e.g., Edge, far Edge, Fog), other than the Cloud.

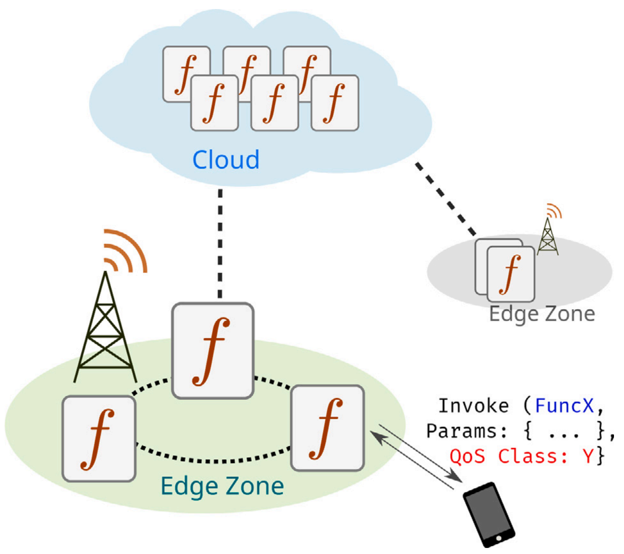


Fig. 1. System overview.

The FaaS system enables users to invoke multiple serverless functions, likely characterized by heterogeneous resource demands. As each node can concurrently serve requests for different functions, possibly coming from multiple users, serverless functions are usually executed within isolated execution environments. With no loss of generality, in the following we will assume these environments to be software containers, as for most existing FaaS frameworks.

Containers are created by the FaaS platform and initialized with the code and libraries needed for each function. Therefore, each node maintains a pool of active containers for each function, which is initially empty. In principle, upon arrival of an invocation request, a new container is created and added to the pool. In this case, the incoming request has to wait for the container to be fully initialized before being served and it is said to experience a *cold start*, which can introduce additional delays in the order of hundreds of milliseconds or even seconds. To limit the occurrence of cold starts, upon the termination of a function, idle containers are marked as *warm* and kept in the pool. When a new request for the function arrives, any available warm container is reused to serve it. Idle warm containers usually stay in the pool for a maximum time (e.g., about 10–15 minutes in commercial Cloud FaaS offerings) or until their resources are reclaimed by the node.

Clearly, spawning a new container is only possible if enough computational resources (e.g., memory) are available on the node. When no container is available nor can be created to serve a request, a node can either discard the request or *offload* it to another node. Specifically, Edge nodes are able to offload requests both to the Cloud (i.e., *vertical offloading*) or to a neighboring node within the same Edge zone (i.e., *horizontal offloading*). For this purpose, we assume that Edge nodes pick a subset of *peers* in their same zone, based, e.g., on network proximity. Peers periodically exchange information on resource availability (as done, e.g., in [8]). By doing so, the target node for Edge offloading can be selected through weighted round-robin or weighted randomization.

Invocation requests are also associated with a *service class*, characterized by one or more Quality-of-Service (QoS) requirements and attributed (e.g., maximum response time, priority level). Depending on the specific application scenario, the service class for each request may be explicitly set by users or automatically determined by the system for requests of each user (e.g., based on the subscription level of each user).

4. System model

We introduce the model of the FaaS system we consider in this section, including the computing infrastructure on which the FaaS system relies, the offered functions, and the different service classes with their QoS requirements.

4.1. Faas computing infrastructure

The FaaS system comprises a set of nodes $N = N^E \bigcup \{\Omega\}$, where N^E are Edge nodes and Ω is a "virtual" node that abstracts the whole Cloud-end of the system as a single node. The assumption of having a single Cloud node allows us to simplify notation in the following and is coherent with the black-box view that we usually have of public Cloud FaaS platforms (e.g., AWS Lambda), whose internals are not exposed to users.

Each node $u \in N$ is characterized by M_u, that is the amount of memory available on the node, σ_u, a relative speedup factor with respect to a reference processor, and c_u, a monetary price associated with function execution. Inspired by the pricing of commercial FaaS offerings, we assume c_u to indicate the cost of running a function per unit of time *and* per unit of allocated memory.

Furthermore, given any pairs of nodes $u, v \in N$, we denote by $\delta_{u,v}$ the network round-trip time (RTT) between u and v, with $\delta_{u,u} = 0$. Similarly, we denote by $b_{u,v}$ the available network bandwidth for data transfers between nodes u and v. Based on network proximity, each Edge node $u \in N^E$ picks a set π_u of neighboring Edge nodes that can be contacted for request offloading. For instance, in the prototype we use for implementation (see, Section 7), nodes run the Vivaldi algorithm [36] to establish a network coordinate system and evaluate proximity.[2]

4.2. Functions and response time model

The set of functions available for execution is denoted as $\mathcal{F}$. Each function $f \in \mathcal{F}$ is characterized by its memory demand $m_f \in \mathbb{Z}^+$, which defines the amount of memory that must be reserved for each instance of f. Moreover, we denote as d^i_f the random variable associated with the size of the input data provided by users when invoking f. Similarly, d^o_f denotes the data size of output returned by f.

We denote as T^u_f the random variable associated with the execution time of f on a computing node u. Similarly, I^u_f denotes the time required to initialize a new container for f on u (i.e., the cold start delay). When the reference to the node is not ambiguous, we will simply write T_f and I_f to improve readability.

Accordingly, for a node u, the response time of locally served requests R^u_f can be formulated as follows:

$$R^u_f = T^u_f + \hat{p}^u_f I^u_f \tag{1}$$

where $\hat{p}^u_f$ is the probability of a cold start of function f to occur on node u.

For requests offloaded to another node v, the response time $R^{u,v}_f$ accounts also for the network delay between u and v as well as the time required to transfer user-given input to the destination node. The response time can be expressed as follows:

$$R^{u,v}_f = T^v_f + \hat{p}^v_f I^v_f + \delta_{u,v} + \frac{d^i_f + d^o_f}{b_{u,v}} \tag{2}$$

where the term $\delta_{u,v}$ accounts for the communication delay, and $\frac{d^i_f + d^o_f}{b_{u,v}}$ is the data transfer time.

[2] It is worth noting that, although the presented model treats infrastructure parameters as constant, as explained in the next sections, our approach periodically refreshes such parameters at run-time to compute an updated offloading policy. Therefore, our approach can cope with dynamic working conditions (e.g., varying resource prices or network latencies).

4.3. Service classes

Function invocation requests specify, along with the function $f \in \mathcal{F}$ to be executed, an associated service class. The set of available service classes is denoted as $\mathcal{K}$. Each class $\kappa \in \mathcal{K}$ is characterized by:

- $R_\kappa^{max} \in \mathbb{R}^+$, maximum desired *response time*. In our context, response time represents the interval between the time a request first reaches any FaaS node and the time a response is sent to the invoking client.
- $u_\kappa \in \mathbb{R}_0^+$, *utility* generated by successfully serving a request. We consider a request to be successfully served if (i) the function code is executed on any node and (ii) the response time of the request does not exceed the maximum response time R_κ^{max}.
- $\phi_\kappa \in \mathbb{R}_0^+$, *utility penalty* paid in case of a request is served by the system and the response time exceeds the maximum response time R_κ^{max}.

For every function–class pair (f, κ), we denote as $\lambda_{f,\kappa}$ the (measured) average arrival rate of requests for f in class κ.

5. QoS-aware offloading policy

According to the system description given in Section 3, when an Edge node receives a new request of class κ for a function f, the scheduler component of the node has to make a decision regarding the execution and possible offloading of the request. Specifically, the following actions are available to the scheduler:

(1) *local execution* of the function (L, for short);
(2) *offloading to the Cloud* (C);
(3) *offloading to a neighbor Edge node* (E);
(4) *discarding* the request (D).

Actually, some of the actions above may not be eligible for some requests. Indeed, local execution is only possible if the node has computational resources for it (i.e., a warm container or enough memory to launch a new one). Moreover, we do not allow the scheduler to offload a request more than once, to prevent long "offloading chains" and ping-pong effects.

We recall that each request generates some utility, if it is completed within the associated maximum response time. The amount of generated utility depends on the QoS class of the request and, for some classes, violating the response time requirement will cause a utility penalty instead. As such, our goal is to devise a policy to properly schedule incoming requests on every node so as to maximize the utility generated over time, while minimizing the penalties. In the following, we will use the expression *net utility* to refer to the generated utility left after subtracting the penalties, which represents our optimization objective.

Furthermore, according to the cost model introduced above, resource usage for function execution causes the payment of a monetary cost. We consider a maximum hourly budget C^{max} to be spent for resource usage.

To devise a policy that addresses this problem, we resort to a two-level solution. We equip each node in the system with a lightweight randomized scheduling heuristic, which introduces minimal overhead in the processing of every request. As described below, the heuristic approach relies on a set of parameters (i.e., probability vectors) that are periodically and asynchronously updated through the resolution of an optimization problem. In the rest of this section, we first present the randomized policy and then illustrate the optimization problem.

Algorithm 1: Node scheduling policy

```
    Data: $(f, \kappa)$            ▷ Request function and QoS class
    Data: $n_o \geq 0$  ▷ Times the request has been offloaded
    Data: $\mathbf{p} = [p^L_{f,\kappa}, p^C_{f,\kappa}, p^E_{f,\kappa}, p^D_{f,\kappa}]$
    Result: decision $\in \{L, C, E, D\}$
1   if $n_o > 0$ then
2       $p^C_{f,\kappa}, p^E_{f,\kappa} \leftarrow 0$                ▷ No offloading
3   if CannotExecuteLocally($f$) then
4       $p^L_{f,\kappa} \leftarrow 0$                        ▷ No local exec.
5   decision ← RandomizedChoice($\mathbf{p}$)
```

5.1. Scheduler policy

We propose the following randomized policy to make a scheduling and offloading decision for each request. Each node relies on a vector $\mathbf{p}_{\mathbf{f},\boldsymbol{\kappa}} = [p^L_{f,\kappa}, p^C_{f,\kappa}, p^E_{f,\kappa}, p^D_{f,\kappa}]$ for each function–class pair (f, κ), which is a discrete probability distribution over the scheduling actions introduced above. Therefore, the probabilities in $\mathbf{p}_{f,\kappa}$ denote:

- $p^L_{f,\kappa}$: probability of serving a request locally;
- $p^C_{f,\kappa}$: probability of offloading a request to the Cloud;
- $p^E_{f,\kappa}$: probability of offloading a request to a neighboring Edge node;
- $p^D_{f,\kappa}$: probability of rejecting a request;

According to this probability distribution, the node can make a randomized decision for every incoming request, possibly accounting for the ineligibility of some actions. The algorithm first checks whether the request has already been offloaded (line 1) and, if this is the case, it prohibits Edge offloading by setting the associated probability to zero (line 2). Similarly, we prohibit local execution if the node has currently not enough resources available (line 4). Then, we randomly make a decision based on the resulting probability vector.[3]

Budget enforcement. The algorithm can be slightly extended to introduce a stricter enforcement of the monetary budget for Cloud usage. For this purpose, an additional check is performed at the end of the algorithm if Cloud offloading has been selected. Specifically, we verify whether the current average hourly expense $C_H(t)$ exceeds the budget C^{max} and, if this happens, we do not offload the request and drop it.

5.2. Offloading policy optimization

The randomized policy used by each node is based on the probability vectors $\mathbf{p}_{f,\kappa}$, which must be determined for every function f and class κ. In order to optimize system performance, we assume that each node computes its own set of offloading probabilities, by solving a suitable utility-based optimization problem, which takes the form a linear programming problem.

The decision variables in our problem correspond to the matrix $\mathbf{P}$, which comprises all the probability vectors $\mathbf{p}_{f,\kappa}$:

$$\mathbf{P} = \begin{bmatrix} — & \mathbf{p}_{f_1,\kappa_1} & — \\ — & \mathbf{p}_{f_1,\kappa_2} & — \\ & \dots & \\ — & \mathbf{p}_{f_i,\kappa_j} & — \\ & \dots & \end{bmatrix}, \forall f \in \mathcal{F}, \kappa \in \mathcal{K}$$

It is worth observing that these probabilities are computed separately for each FaaS node, as their workload and resource capacity may

[3] After updating one or more probabilities, the sum of the elements of $\mathbf{p}$ may differ from 1. We assume a normalization to be carried out when needed.

differ. Hereafter, we will use the symbol u to indicate the node for which we perform the probability computation. For every function and QoS class, we will have to suitably adjust the probability of executing requests on u, offloading to the Cloud Ω, or offloading to a neighboring Edge node. As explained above, the decision of offloading to a neighboring Edge node does not include information about the specific node where the request is offloaded. Therefore, in our optimization problem we model Edge nodes as a single "virtual" node, whose resource capacity equals the sum of the capacity of each Edge peer. Specifically, we will denote the virtual Edge node as e. The memory capacity of e, denoted as $\tilde{M}_e$, is computed as:

$$\tilde{M}_e = \sum_{w \in \pi_u} \alpha_w^E \tilde{M}_w \tag{3}$$

where $\tilde{M}_w$ is the unused memory of each peer node $w \in \pi_u$, and $\alpha_w^E \in [0,1]$ indicates the memory fraction that w exposes to peers (e.g., $\alpha_w^E = 1$ means all the available memory).

In case of Edge offloading, the choice of the actual peer where the request is forwarded is made based on current memory availability (e.g., through randomization or weighted round-robin). For this purpose, we compute the probability p_w of selecting each peer $w \in \pi_u$ as:

$$p_w = \frac{\alpha_w^E \tilde{M}_w}{\tilde{M}_e} \tag{4}$$

To estimate the expected execution time of a function f on e, we compute it as follows, considering the probability of the request being actually executed on each peer:

$$T_f^e = \sum_{w \in \pi_u} p_w T_f^w \tag{5}$$

By the same reasoning, we estimate the round-trip time to e, its monetary usage cost and the other metrics of interest.

As regards the optimization objective, our goal is to maximize the net generated utility, defined as follows.

$$U(\mathbf{P}) = U^+(\mathbf{P}) - U^-(\mathbf{P}) \tag{6}$$

where $U^+(\mathbf{P})$ is the expected utility generated with the probability matrix $\mathbf{P}$, and $U^-(\mathbf{P})$ are the penalties incurred for response time violations.

The expected *utility* is formulated as follows:

$$\begin{aligned} U^+(\mathbf{P}) = \sum_{\kappa \in \mathcal{K}} u_\kappa \sum_{f \in \mathcal{F}} \lambda_{f,\kappa} \Big[& p_L^{f,\kappa} P(R_f^u \le R_\kappa^{max}) + \\ & + p_C^{f,\kappa} P(R_f^\Omega \le R_\kappa^{max}) + \\ & + p_E^{f,\kappa} P(R_f^e \le R_\kappa^{max}) \Big] \end{aligned} \tag{7}$$

where we sum the utility generated by each class, which is computed multiplying the class utility u_κ and the rate of requests in class κ satisfying the response time requirement. For this purpose, we need to compute the fraction of requests of every function that meets their deadline when executing on the local node, in the Cloud and in another Edge node (see, Section 5.4 for details on cold start probability estimation).

The expected incurred *penalty* is formulated in analogous manner:

$$\begin{aligned} U^-(\mathbf{P}) = \sum_{\kappa \in \mathcal{K}} \phi_\kappa \sum_{f \in \mathcal{F}} \lambda_{f,\kappa} \Big[& p_L^{f,\kappa} P(R_f^u > R_\kappa^{max}) + \\ & + p_C^{f,\kappa} P(R_f^\Omega > R_\kappa^{max}) + \\ & + p_E^{f,\kappa} P(R_f^e > R_\kappa^{max}) \Big] \end{aligned} \tag{8}$$

5.3. Linear programming formulation

To determine the probability matrix $\mathbf{P}$ we formulate the following linear problem:

$$\max\ U(\mathbf{P})$$

$$c_\Omega \sum_{f \in \mathcal{F}} \sum_{\kappa \in \mathcal{K}} \lambda_{f,\kappa} p_{f,\kappa}^C T_f^\Omega m_f \le C_s^{max} \tag{9}$$

$$\sum_{f \in \mathcal{F}} m_f \sum_{\kappa \in \mathcal{K}} x_{f,\kappa} \le \tilde{M}_u \tag{10}$$

$$\sum_{f \in \mathcal{F}} m_f \sum_{\kappa \in \mathcal{K}} y_{f,\kappa} \le \tilde{M}_e \tag{11}$$

$$p_{f,\kappa}^L \lambda_{f,\kappa} T_f^u = x_{f,\kappa} \qquad \forall f \in \mathcal{F}\ \forall \kappa \in \mathcal{K} \tag{12}$$

$$p_{f,\kappa}^E \lambda_{f,\kappa} T_f^e = y_{f,\kappa} \qquad \forall f \in \mathcal{F}\ \forall \kappa \in \mathcal{K} \tag{13}$$

$$p_{f,\kappa}^L + p_{f,\kappa}^C + p_{f,\kappa}^D + p_{f,\kappa}^E = 1 \qquad \forall f \in \mathcal{F}\ \forall \kappa \in \mathcal{K} \tag{14}$$

$$p_{f,\kappa}^X \in [0,1] \qquad X \in \{L, C, E, D\}\ \forall f \in \mathcal{F}\ \forall \kappa \in \mathcal{K}$$

$$x_{f,\kappa} \in \mathbb{R}^+ \qquad \forall f \in \mathcal{F}\ \forall \kappa \in \mathcal{K}$$

$$y_{f,\kappa} \in \mathbb{R}^+ \qquad \forall f \in \mathcal{F}\ \forall \kappa \in \mathcal{K}$$

where: (1) in addition to the scheduling action probabilities $p_{f,\kappa}^X$, $X \in \{L, C, E, D\}$, $f \in \mathcal{F}$, $\kappa \in \mathcal{K}$, we introduce the following auxiliary variables

- $x_{f,\kappa}$ that represent the average number of instances of f provisioned on the local node serving requests of class κ, for every function and class; in other words, they represent the share of resources on the node allocated to (f,κ). Note that these variables are not strictly necessary for problem formulation, as they can expressed in terms of probability variables, but we include them to improve formulation readability.
- Similarly, $y_{f,\kappa}$ that represent the average number of instances of f provisioned on the Edge node e to serve requests of class κ

and, (2) constraints (9)–(13) captures the resources' related constraints, that is,

- Constraint (9) imposes a maximum monetary budget for Cloud usage. As the budget C^{max} is defined on an hourly basis, we simply let $C_s^{max} = C^{max}/3600$ represent the budget per second.
- Constraint (10) models the limited memory capacity of the local node. Note that the memory capacity bound is given by $\tilde{M}_u \le M_u$, indicating that the memory exposed by the node can be lower than the total capacity (we will better explain this point in Section 5.5). We also remark that we limit our discussion to memory capacity, following the approach taken by most public FaaS offerings, where users can only configure memory allocation and other resources (e.g., CPU shares) are proportionally assigned. Nevertheless, our model can be readily extended to account for other computational resources in addition to memory.
- Constraint (11) models the limited memory capacity of the Edge node e, as defined in (3).
- Constraint (12) relates the probability of serving requests locally $p_{f,\kappa}^L$ to the number of provisioned instances $x_{f,\kappa}$, applying Little's Law [37]. Constraint (13) has the same role for variables $p_{f,\kappa}^E$ and $y_{f,\kappa}$.

Being linear, the problem can be efficiently resolved using standard techniques and solvers for linear programming. Moreover, it can be observed that both the number of variables and constraints in the formulation grow linearly with functions and QoS classes, and are independent of the number of considered Edge neighbors, as they are abstracted away by the virtual node e. The results presented in the following will confirm that computing the optimal probabilities can be done with limited computational effort.

5.4. Cold start probability estimation

Formulating the utility expression introduced in (7) requires us to compute the CDF of function response time on the local node u, in the

Cloud and on the Edge node e. The same computation can be used to formulate the penalties in (8), as $P(X > x) = 1 - P(X \leq x)$.

Expressions for the response time of locally served requests and offloaded requests have been given, respectively, in (1) and (2). We recall from those equations that response time mostly depends on (i) function execution time, (ii) initialization time, in case of cold start, and (iii) probability of cold start occurrence.

As regards (i), the problem of studying the execution time distribution of serverless functions, and tasks in general, has been widely investigated, via, e.g., profiling, log analysis. As these techniques are beyond the scope of this paper, we assume that information on the execution time distribution is provided to the scheduler. Specifically, we assume function execution time to follow the exponential distribution. Similarly, we assume estimates of the initialization time of each function to be available (e.g., after analysis of execution logs).

As regards (iii), we consider different techniques to estimate the probability of having a cold start for each function on the local/Cloud/Edge nodes. In particular, the following approaches are considered:

- *Heu1*: a naive approach that uses the historical cold start frequency on a given node to estimate the future probability.
- *Heu2*: an approach similar to Heu1, where historical cold start frequency *per function* is used to estimate the future probability.
- *SMP*: a model based on semi-Markov processes, presented in [38] and implemented as open-source software,[4] to study the steady-state cold start probability of serverless platforms.

5.5. Dynamic memory constraint

The problem presented in Section 5.3 takes into account the memory capacity of the local node to determine how many instances of each function can be concurrently allocated. We observed that constraint (10) uses a value $\tilde{M}_u \leq M_u$ to bound the memory availability of the node. The reason we do not simply use the nominal memory capacity M_u is that the probability optimization possibly leads to complete allocation of the node resources. While it is desirable in principle, it is well known from queueing theory that system performance quickly degrades when resource utilization is closer to 100%.

To address this problem, a common approach is to keep resource utilization below a pre-defined threshold θ (e.g., $\theta = 75\%$). However, determining the optimal value for this threshold *a priori*, for any given workload and infrastructure, is not trivial and would require extensive tuning. In this work we adopt a different approach and let the system automatically adapt such threshold at run-time, by dynamically adjusting the fraction of the available memory that can be allocated to functions, i.e., $\tilde{M}_u = \theta_u M_u$.

Specifically, we keep a count of how many requests scheduled for local execution must be re-scheduled due to resource shortage. Dividing this count by the total number of requests scheduled for local execution, we obtain an empirical estimate of the "blocking" probability p_B of the node. If $p_B > 0$ in the reference time window, we update θ as follows:

$$\theta \leftarrow \theta(1 - \frac{p_B}{\beta}) \tag{15}$$

Otherwise, we increase θ:

$$\theta \leftarrow \min\{1, \theta(1+\gamma)\} \tag{16}$$

In this work, we set $\beta = 2$ and $\gamma = 0.1$.

A similar approach is used by each node to adjust the fraction of its unused memory to be offered to peers for offloading (see, Eq. (3)). In this case, we consider as "blocked" requests offloaded from other nodes that cannot be executed due to resource shortage.

[4] https://github.com/pacslab/serverless-performance-modeling

6. Evaluation

We evaluate the proposed offloading policy by simulation. We implement the simulator in Python, using the *PuLP*[5] library for the LP formulation and the *GLPK*[6] solver for the resolution. In this section, we first describe the experimental setup we use and then discuss the simulation results.

6.1. Experimental setup

Infrastructure. According to the system model presented in Section 4, we consider 1 Cloud node Ω and a set N^E of 10 Edge nodes. Memory capacity is set as $M_\Omega = 128$ GB and $M_u = 4$ GB for every $u \in N^E$. For simplicity, we consider an identical speedup value $\sigma_u = 1$ for Cloud and Edge nodes, and set the Cloud usage cost as $c_\Omega = 0.00005$ \$/GB-s. In most the experiments, relying on the same approach adopted in [9], we associate a cost only with the Cloud node Ω (e.g., a commercial FaaS platform), and assume that the FaaS provider owns/rents Edge nodes, and hence the amortized cost per single request is negligible. To demonstrate that the model is general enough, we also present a specific experiment where we associate a cost with Edge nodes too.

For all the pairs of Edge nodes, we set identical network latency equal to 5 ms; for the latency between Edge and Cloud, we consider different values $\{50 \text{ ms}, 100 \text{ ms}, 200 \text{ ms}\}$. As regards the network bandwidth, we assume 100 Mbps between Edge nodes, and 10 Mpbs between Edge and Cloud. Each Edge node randomly picks 3 peers for horizontal offloading.

Functions and invocations. In each experiment, we consider invocation requests targeting 5 functions with different resource demands. To avoid the results being affected by an arbitrary choice of request inter-arrival time and service time distribution, in the experiments we randomly mix different distributions. In particular, for the execution time of each function we randomly pick a distribution among Exponential, Erlang-2 and Erlang-4, with the mean value uniformly sampled from $(100, 500)$ ms. Similarly, the memory demand of each function is uniformly sampled from $(128, 1024)$ MB, and the initialization time from $(250, 750)$ ms. The input data size for each function is sampled from a truncated normal distribution, with mean and standard deviation equal to 1 KB, and supports (100B, 5MB).

Invocation requests to each function are modeled as independent arrival processes, with mean arrival rate $\lambda = 10$req/s. While the mean arrival rate is fixed, in each experiment we randomly associate each function with a different inter-arrival time distribution, choosing from: Exponential, Erlang-2, Hyperexponential, a 2-state Markov Modulated Poisson Process (MMPP). These distributions lead to arrival processes with different levels of variability. Moreover, MMPP differ from the other ones as they model bursty arrivals.

We assume that all invocation requests are directed to Edge nodes.[7] Moreover, except for an experiment where it is differently stated, we consider request arrivals to a single Edge node in the infrastructure, to simplify configuration and analysis. We will also consider an experiment where arrival rates change over time, increasing/decreasing workload intensity.

QoS classes. We consider the 4 service classes specified in Table 1. Invocation requests are randomly tagged with one of the classes, according to the probabilities reported in the last column of the table.

Offloading policies. We compare different offloading policies in the experiments, including the one presented in this work and various baselines. Specifically, we consider the following approaches:

[5] https://coin-or.github.io/pulp/

[6] https://www.gnu.org/software/glpk/

[7] In practice, we can expect some requests to directly reach Cloud nodes. However, as the Cloud has abundant computing resources to accommodate incoming requests, this additional traffic has negligible impact on our problem.

Table 1
Service classes used in the experiments.

Class κ	R^{max}_{κ}	u_{κ}	ϕ_{κ}	Class prob.
Standard	0.5 s	0.01	–	70%
Critical-1	0.5 s	1.00	–	10%
Critical-2	0.5 s	1.00	0.75	10%
Batch	∞	1.00	–	10%

Table 2
Considered policies and associated labels. (*): our contribution.

	Label	Policy
	Rand	Random
	Bc	Basic: local execution when possible, otherwise Cloud
	Be	Basic with Edge offloading
	Bc+	Basic with budget enforcement
	minR	Local/Cloud to minimize resp. time
	minR+	*minR* + budget enforcement
	minC	Local/Cloud to minimize cost, given a maximum response time
*	QoSc	QoS-aware, without Edge offloading
*	QoSc+	QoS-aware, without Edge offloading + budget enforcement
*	QoS	QoS-aware
*	QoS+	QoS-aware + budget enforcement

- Our QoS-aware policy (denoted below as **QoS**, for short). We also consider a variant where offloading is only allowed to the Cloud (**QoSc**), and two additional variants where the monetary budget is strictly enforced (see the last part of Section 5.1), denoted as **QoS**+ and **QoSc**+.
- A baseline policy (**Rand**) that randomly makes decisions for every request.
- A basic heuristic policy (**Bc**), inspired by the one used in [8], that greedily executes all requests on the local node if enough resources are available, and offloads to the Cloud otherwise. We also consider a variant **Bc**+ where the availability of Cloud offloading is subject to the budget constraint; and a variant **Be** that offloads requests to Edge peers when necessary rather than to the Cloud.
- A heuristic policy (**minR**), adapted from the one presented in [9], that for each request estimates the response time (i) if the request is executed on the node, and (ii) if it is offloaded to the Cloud, accounting for cold start probability, network latency, and function execution time. The option leading to the minimum response time is selected. Similarly to the other policies above, we also define a variant **minR**+ where Cloud offloading is disabled when the average hourly expenses exceed the budget C^{max}.
  Moreover, we consider a variant policy **minC** that, after estimating the local and remote (i.e., in the Cloud) response time, picks the least-costing option whose expected response time is below the deadline (i.e., the maximum response time for the service class). In other words, Cloud offloading is only selected if it allows us to meet the QoS requirement and the local node does not.

For convenience, Table 2 provides a summary of the policies considered and associated short labels.

Other parameters. We simulate the execution of the system for one hour, replicating every experiment 10 times using different seeds for random number generation. We consider different values for the Cloud usage budget $C^{max} \in \{0.25, 0.5, 1, 10\}$ \$/h. For our policy, probabilities are computed every 120 s by resolving the LP model. To estimate the arrival rate $\lambda_{f,\kappa}$ for all functions and classes, FaaS nodes measure the arrival rate during each time window between successive LP resolutions and use an exponential moving average to update $\lambda_{f,\kappa}$. Being $\lambda'_{f,\kappa}$ the arrival rate measured in the last time window, we get $\lambda_{f,\kappa} \leftarrow \alpha\lambda'_{f,\kappa} + (1-\alpha)\lambda_{f,\kappa}$. In the experiments, we set $\alpha = \frac{1}{3}$.

For cold start estimation, we performed preliminary experiments to identify the best strategy to use among those mentioned in Section 5.4. Although we did not observe significant performance difference among them, we eventually chose the following configuration: the heuristic *Heu1* is used to estimate cold start probability on the local node, while the model-based *SMP* is used for Cloud and Edge nodes. The *minR* policy also relies on cold start estimation and uses the same configuration except for the local node, where it relies on real-time knowledge of container availability to predict cold starts.

6.2. Results

Overall policy comparison. In Fig. 2 we compare the performance of the various offloading policies in terms of (i) net generated utility, and (ii) cost-to-budget percentage, across the considered budget and network latency configurations. In the figure, the area shaded in red shows the budget violation. Utility and cost measures are also reported in Table 3 for a single latency scenario.

As expected, the random policy leads to very low utility, compared to the other policies, and occasional budget violations. The basic policy *Bc* from [8] achieves much higher utility, but, not taking costs into account, it violates the budget constraint in half the considered runs. This cost issue is solved by *Bc+*, which prohibits Cloud offloading if the budget is being violated. This policy manages to keep the cost within the given budget, with an utility reduction compared to the baseline *Bc*. As regards the variant *Be*, which only relies on Edge offloading, clearly in this case there are no budget violations, as we associated monetary expenses only with Cloud usage. However, the generated utility is the lowest among all the policies. These results suggest that Cloud and Edge offloading should be jointly exploited to optimize both utility and cost, as our QoS-aware approach does.

The *minR* heuristic policy leads to the highest utility, both in terms of median and 95th percentile. However, similarly to *B*, it neglects Cloud usage costs and violates the budget in most the considered configurations (up to 4 times higher than the budget). The policy *minR*+ addresses this issue enforcing the budget constraint, with about 40% reduction of the median utility. The variant *minC* only opts for Cloud offloading if local execution does not meet the response time requirements. Unfortunately, the results show that this policy eventually behaves similarly to *minR*, resorting to Cloud too often and violating the budget constraint, while achieving a high utility though.

Our QoS-aware policy achieves good, consistent performance across the considered configurations, with minimal differences among the four variants. Specifically, the *QoS* policy leads to a 1.5% median utility increase with respect to *QoSc* (where only Cloud offloading is admitted), with a 2% median cost reduction. The variant with the budget enforcement mechanism *QoS*+ reduces the percentage of runs where the budget is exceeded from 47% to 9%. However, it is worth remarking that the average excess cost is less than 1.5% for *QoS* and less than 0.001% for *QoS*+, compared to 400% violations observed for the approaches above. Overall, *QoS*+ achieves 40% higher median utility compared to *minR*+, with almost identical costs.

Impact of network latency and budget. Fig. 3 provides insight about the impact of network latency on function offloading (we report the results only for a relevant subset of the policies discussed above). As expected, the maximum utility achieved by the policies decreases as network latency increases, making Cloud offloading less and less convenient. The only exception to this trend is represented by *Be*, which only relies on Edge offloading and is not impacted by the latency variation, but its performance is never competitive. Interestingly, compared to the *Bc* and *minR* heuristics, our QoS-aware policy is way less impacted by the increase of latency, as it also exploits Edge offloading and request dropping to efficiently use the available resources.

Similarly, Fig. 4 shows what happens when we change the monetary budget C^{max}. Increasing the budget allows the scheduler to offload a larger number of requests to the Cloud, leading to higher utility in general. The highest utility is achieved by the *minR* policy, but – as already observed – this policy also leads to evident budget violations,

Table 3
Paid cost and generated utility with different budget values and Edge-Cloud latency set to 100 ms. Darker background colors indicate higher budget violations in the cost columns, and higher utility in the utility columns.

Budget ($/h)	Policy	Cost ($/h)			Net utility		
		P_{50}	P_{90}	P_{95}	P_{50}	P_{90}	P_{95}
0.25	Rand	0.56	0.73	0.81	2.14	2.62	2.75
	Bc	0.91	1.67	1.71	3.30	3.61	3.65
	Bc+	0.25	0.25	0.25	1.21	1.70	1.89
	Be	0.00	0.00	0.00	1.59	2.16	2.18
	minR	1.30	1.63	1.69	3.74	4.08	4.42
	minR+	0.25	0.25	0.25	1.20	2.11	2.11
	minC	1.13	1.54	1.54	3.93	4.52	4.55
	QoSc	0.27	0.27	0.27	3.54	3.96	4.08
	QoSc+	0.25	0.25	0.25	3.39	3.59	3.60
	QoS	0.27	0.28	0.28	3.62	3.95	4.08
	QoS+	0.25	0.25	0.25	3.64	3.96	4.00
0.50	Rand	0.46	0.74	0.74	2.31	2.45	2.61
	Bc	1.09	1.57	1.65	3.36	3.67	3.80
	Bc+	0.50	0.50	0.50	1.86	2.39	2.69
	Be	0.00	0.00	0.00	1.29	1.50	1.84
	minR	1.21	1.88	2.00	3.66	4.44	4.63
	minR+	0.50	0.50	0.50	3.06	3.70	3.73
	minC	0.93	1.45	1.47	3.78	4.35	4.67
	QoSc	0.51	0.52	0.53	3.56	4.45	4.62
	QoSc+	0.50	0.50	0.50	3.61	4.33	4.52
	QoS	0.51	0.51	0.52	3.75	4.14	4.30
	QoS+	0.50	0.50	0.50	3.77	4.17	4.25
1.00	Rand	0.51	0.62	0.68	2.17	2.39	2.40
	Bc	1.06	1.62	1.64	3.38	3.78	3.87
	Bc+	0.94	1.00	1.00	3.19	3.63	3.72
	Be	0.00	0.00	0.00	1.38	2.62	2.65
	minR	1.31	1.61	1.73	3.69	4.52	4.55
	minR+	0.86	1.00	1.00	3.69	4.12	4.20
	minC	0.90	1.08	1.17	3.90	4.20	4.23
	QoSc	1.00	1.00	1.01	3.32	4.51	4.51
	QoSc+	1.00	1.00	1.00	3.89	4.12	4.13
	QoS	1.00	1.00	1.00	3.71	4.18	4.24
	QoS+	1.00	1.00	1.00	3.87	4.15	4.16

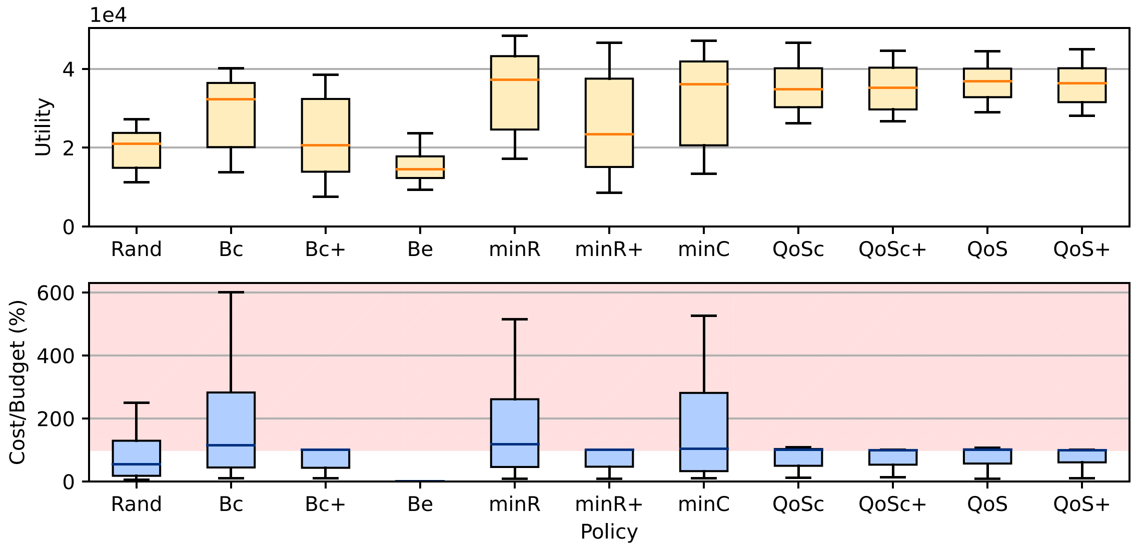


Fig. 2. Comparison of our QoS-aware policies against baselines. See legend in Table 2.

especially for $C^{max} = 0.25\$/\mathrm{h}$ and $C^{max} = 0.5\$/\mathrm{h}$. The *minR+* policy avoids this issue and matches the utility of *minR* with the highest budget $C^{max} = 1\$/\mathrm{h}$. In turn, our approach *QoS+* outperforms *minR+* (and the other policies), especially with the stricter budget constraints, (e.g., achieving more than 100% utility increase for $C^{max} = 0.25\$/\mathrm{h}$). We do not report the results with higher budget configurations, as they are not significantly different from the case $C^{max} = 1\$/\mathrm{h}$. These results show that the benefits of our QoS-aware policy are particularly evident when the scheduler has to carefully allocate Cloud resources, while the *minR+* heuristic performs well with "unconstrained" Cloud usage.

Different workloads. We relax the assumption of having all the requests directed to a single Edge node to verify that the performance of the policies is not impacted. Specifically, in this experiment we evenly split the incoming traffic across the available Edge nodes. Fig. 5 shows the results of this experiment, where we can note that the different

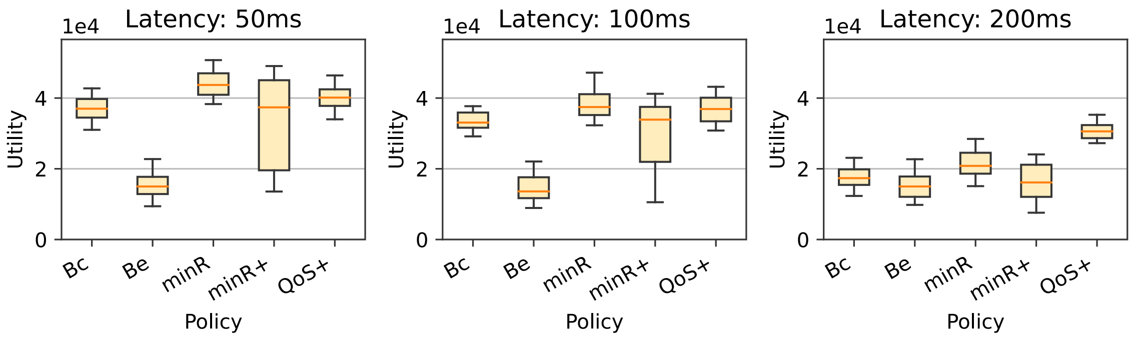


Fig. 3. Results with different Edge-Cloud network latency configurations.

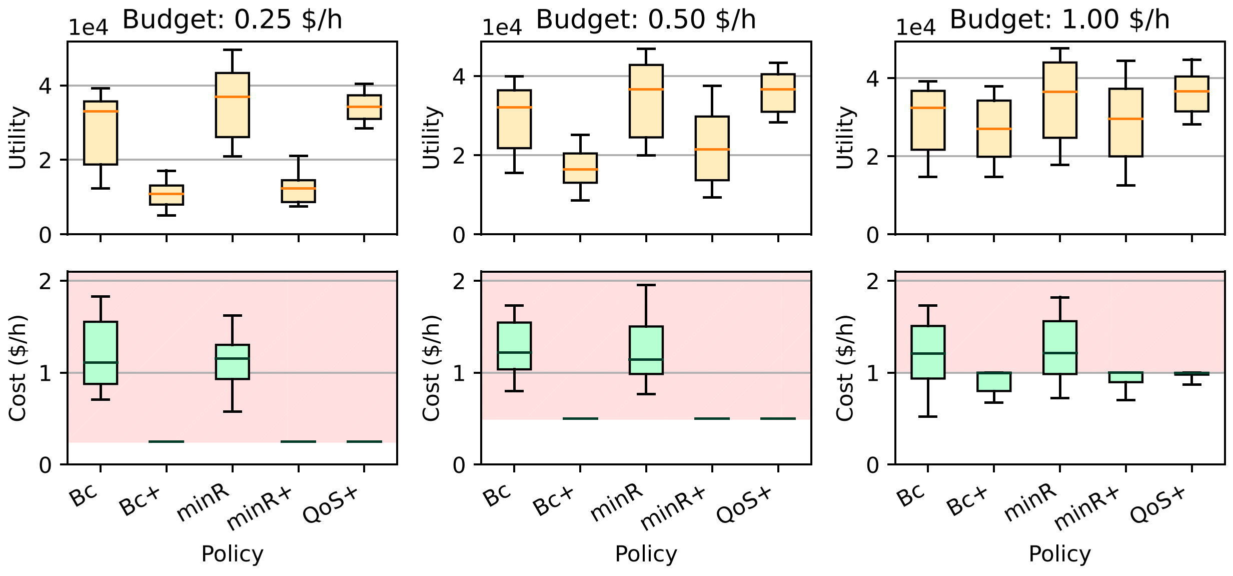


Fig. 4. Results with different budget configurations.

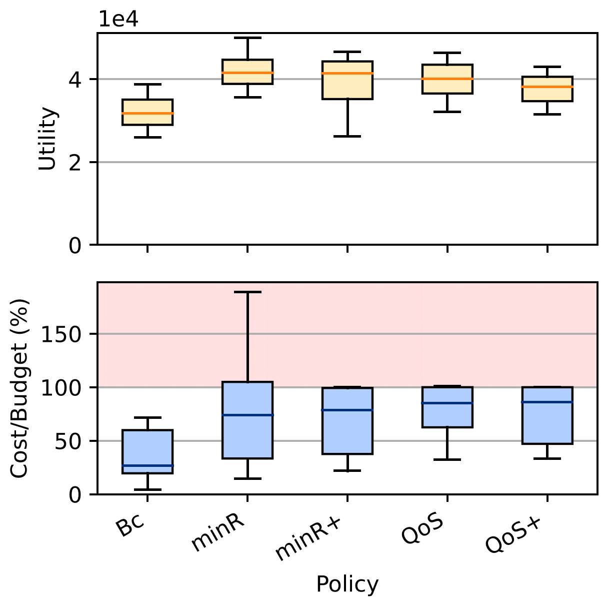


Fig. 5. Results with incoming request flows split across Edge nodes.

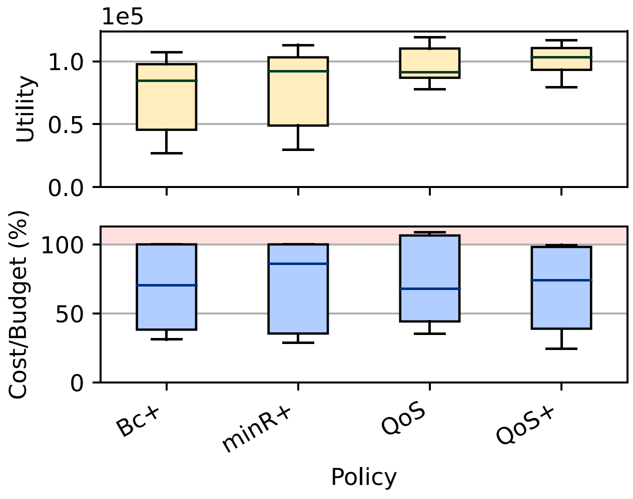


Fig. 6. Results with varying arrival rates.

workload configurations does not significantly affect the relative performance of the considered policies. Moreover, we run an experiment where the arrival rates are not constant. In particular, we introduce random rate variations every $T = 60$ s. Being λ_i the arrival rate of a function at time $i \cdot T$, we sample the arrival rate λ_{i+1} as a uniform random variable taking values in $(\lambda_i/5, 5\lambda_i)$. Fig. 6 shows the results of this experiment, which demonstrates that our QoS-aware policies outperform the baselines even in this scenario.

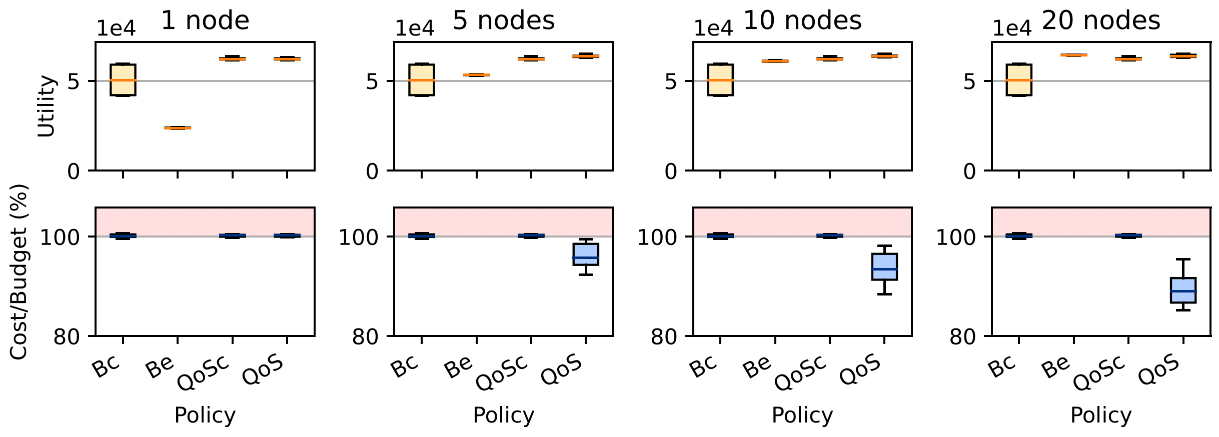


Fig. 7. Results with increasing Edge resources.

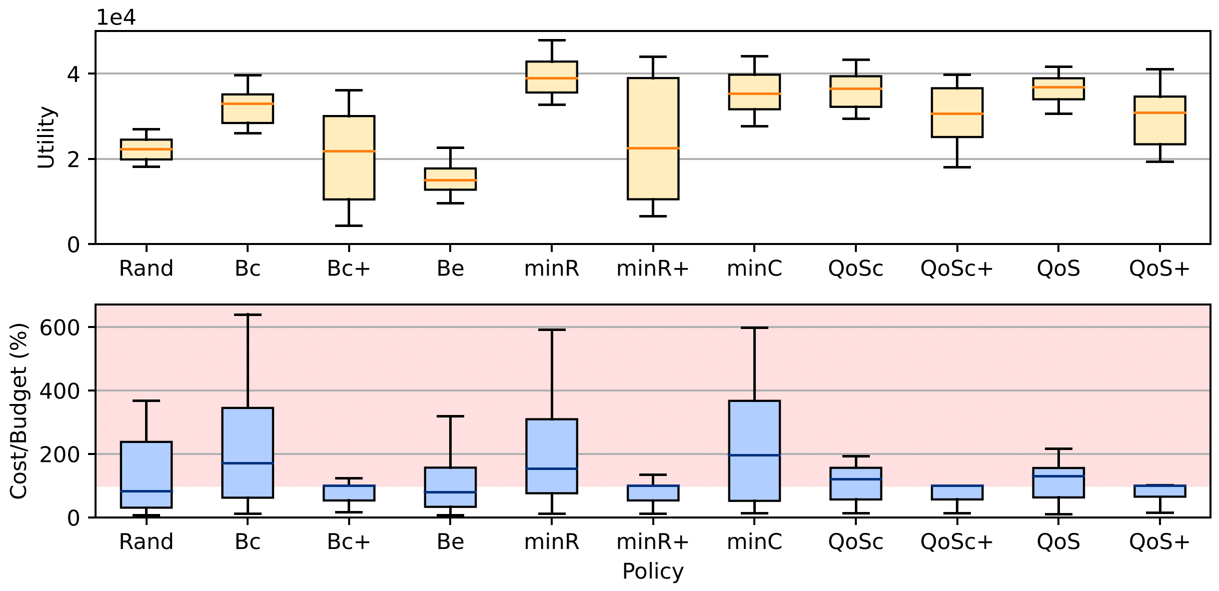


Fig. 8. Comparison of offloading policies in a scenario where a monetary usage cost is associated both with edge and cloud nodes. See legend in Table 2.

Different cost model. So far, we relied on the assumption of having a monetary cost associated with cloud usage only, to emphasize the diverse impact of offloading decisions. To demonstrate that the proposed approach is general enough, we consider an additional scenario where the same cost applies to edge node usage as well. Comparing the results in Fig. 8 to the analogous Fig. 2, we observe that introducing a cost for edge usage causes the monetary budget to be violated more, especially by baselines. Our proposed policies *QoSc+* and *QoS+* still manage to meet the imposed budget constraint and generate high utility.

Varying number of Edge nodes. To further explore the potential benefits of Edge offloading, we run experiments increasing the number of Edge nodes in the infrastructure. Clearly, in these experiments we do not limit the number of peers selected for offloading, which was set to 3 in the default configuration. Fig. 7 confirms that increasing the number of Edge neighbors has no impact for the *Bc* and *QoSc* policies, which only rely on Cloud offloading, and instead significantly boosts the utility generated by *Be*, which relies on Edge offloading. As regards the complete QoS-aware policy *QoS*, we observe that the utility gain with more Edge nodes is not significant. Conversely, the policy manages to largely reduce costs as Edge offloading can increasingly be exploited to avoid Cloud usage.

Scalability. To verify the scalability of our solution, we measure the time it takes to resolve the optimization problem presented in Section 5.3 and, hence, update the offloading probabilities. We execute the experiment increasing the number of different functions in the system and considering 4 and 8 service classes. We run it on a Intel(R) Xeon(R) Silver 4310, relying on GLPK for LP resolution. Fig. 9 shows the results of this test. We observe that, with 4 service classes, the execution time does not reach 400 ms with up to 150 functions, and it does not reach 800 ms with 8 service classes.

We consider the lightweight computational demand of our solution to enable its adoption even in large-scale FaaS systems, especially as we remark that LP resolution can be performed asynchronously and not necessarily on the Edge nodes. Moreover, as each node has its own probabilities, the scale of the LP problem does not change with larger infrastructures. As a final note, we observe that, if necessary, resolution times can be further reduced resorting to commercial LP solvers.[8]

7. Proof-of-concept prototype experiment

To demonstrate the functionality of our QoS-aware policy in a real FaaS prototype, we implement it in Serverledge [8], an open-source FaaS framework developed within our research group and written in

[8] The issue is discussed, e.g., here: https://en.wikibooks.org/wiki/GLPK/Reviews_and_benchmarks

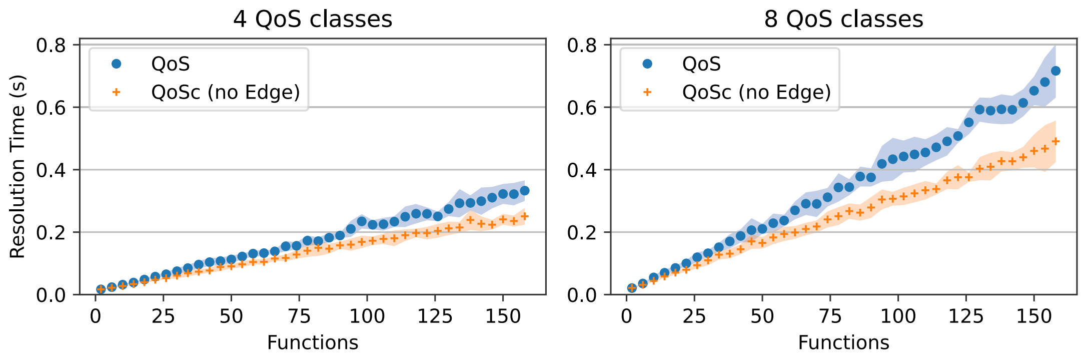


Fig. 9. Scalability to the number of functions.

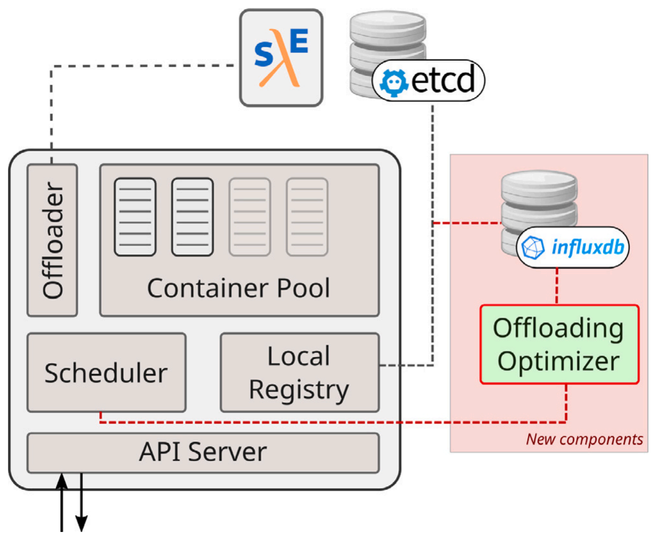


Fig. 10. Architecture of a Serverledge node, with the newly introduced components.

Go. Serverledge targets Edge and Cloud infrastructures and has built-in support for function offloading, so it represents an ideal choice to implement our policy.

Each Serverledge node comprises a few key components, as depicted in Fig. 10: API Server, Scheduler, Local Registry, Offloader, and Container Pool. The API Server acts as a front-end for requests from clients. The Scheduler manages function invocation requests, possibly deciding to offload some of them. Functions executed locally rely on a Container Pool, that is maintained by the node. To offload a request, the Offloader component is activated, which communicates with remote Serverledge nodes. Metrics collection and neighborhood monitoring is performed by the Local Registry, which also acts as a cache for function data stored in the etcd[9]-based Global Registry.

Specifically, the Local Registry runs the Vivaldi algorithm [36] to estimate the network delay towards all of its edge neighboring nodes. Serverledge also exploits the UDP messages exchanged by edge nodes within the Vivaldi algorithm to piggyback status information about the nodes (e.g., currently available memory), as needed by offloading target selection (see Section 5.2).

To integrate our offloading policy, we extended Serverledge as follows:

- We extended the metrics system to collect more data about function execution and offloading, as needed by our optimization problem. In this regard, we used InfluxDB[10] to collect monitoring data and ease their analysis.
- We implemented the randomized scheduling algorithm within the Scheduler component of Serverledge, relying on the interfaces provided by the framework that ease the integration of custom policies.
- We introduced a new *Offloading Optimizer* component that assists Edge nodes by solving the optimization problem and computing their offloading probabilities, on demand. Edge nodes periodically contact the Offloading Optimizer to request new probabilities via gRPC.[11] The Offloading Optimizer retrieves the required metrics from InfluxDB.

7.1. Experimental setup

We run some proof-of-concept experiments with a Serverledge installation consisting of 4 Linux-based VMs: three of them act as Edge nodes, while the fourth represents a Cloud node and also hosts the centralized components required by Serverledge (i.e., etcd, InfluxDB, and the Optimizer component). Each Edge VM is equipped with 4 CPU cores and 3 GB of memory. The Cloud node is configured with 20 CPU cores and 20 GB of memory. Network latency between Edge and Cloud is emulated using Linux Traffic Control `tc`. An additional VM is used to generate the workload through Apache JMeter.[12]

We consider two functions in the experiments: *Fibonacci*, a CPU-intensive function that computes the Fibonacci sequence (up to 25,000 in our setup); and *ImageClassifier*, which uses a convolutional neural network to solve a “cat vs. dog” image classification task.[13]

We consider two service classes, namely a time-sensitive one for “Premium” users, with a given maximum response time requirement (default value: 800 ms) and utility per request equal to 1; a best-effort one, with no response time requirements and utility per request set to 0.01. Incoming requests are randomly associated to one of the two classes.

We compare our *QoSc+* and *QoS+* policy to the basic *Bc* policy, already provided by Serverledge, as well as *minR*. We run two experiments in which, respectively, we consider (i) different monetary budget configurations, and (ii) different response time requirements for the premium class.

[9] https://etcd.io

[10] https://www.influxdata.com

[11] https://grpc.io

[12] https://jmeter.apache.org

[13] The model is small version of Xception, trained as explained here: https://keras.io/examples/vision/image_classification_from_scratch/#image-classification-from-scratch

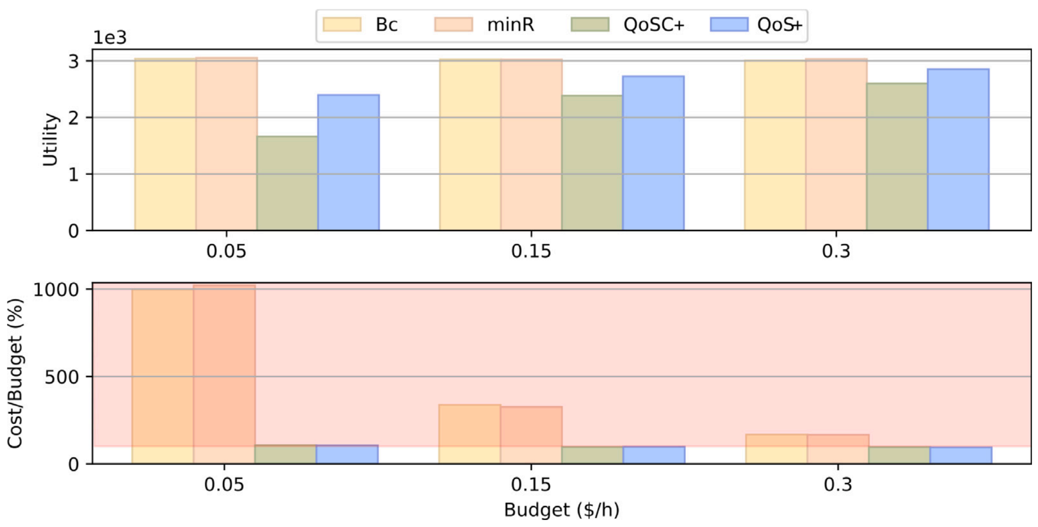


Fig. 11. Results with the prototype implementation with different monetary budget configurations.

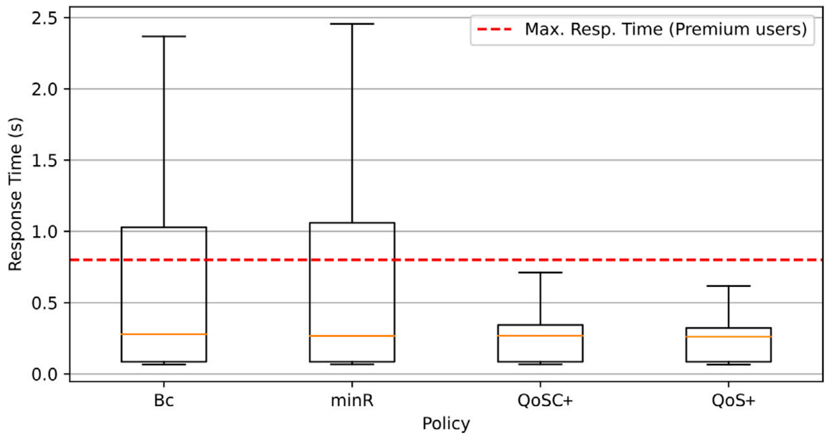


Fig. 12. Response time distribution in a prototype experiment.

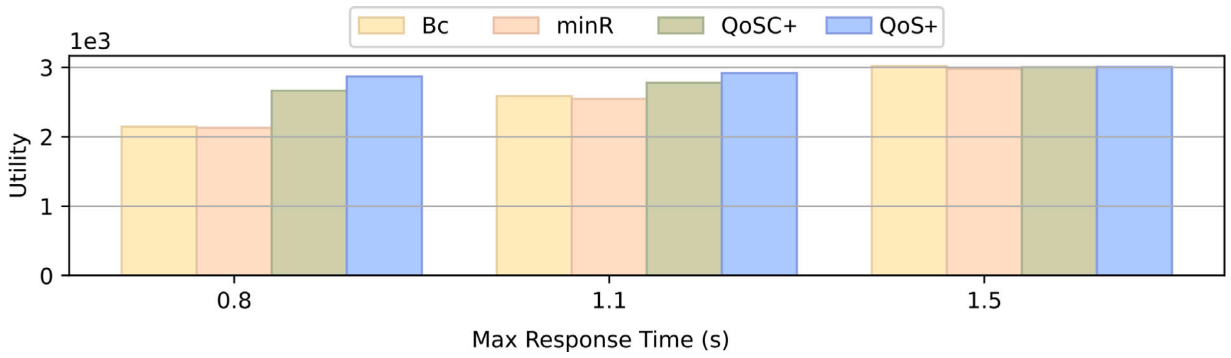


Fig. 13. Results with the prototype implementation with different maximum response time configurations for the *premium* class of users.

7.2. Results

Fig. 11 reports the generated utility and paid cost for the different policies when setting different monetary budget constraints. These results confirm what we already observed in simulated experiments. The baseline *Bc* and *minR* achieve very high utility values, but largely violate the budget constraint, especially when considering stricter limits. Fig. 12 shows the response time distribution in the scenario with budget 0.15 $/h, demonstrating how our QoS-aware policies are superior in minimizing response times, while also keeping costs within the budget.

Fig. 13 shows the generated utility for the different policies when setting different maximum response time requirements for the premium users. The experiment confirms that – as expected – looser response time requirements allow all the policies to increase the generated utility. It is interesting to observe that our QoS-aware approaches clearly outperform the baselines when stricter requirements are considered and, thus, resource allocation plays a more important role. In these scenarios, *QoS+* enjoys an advantage over *QoSc+*, by exploiting Edge nodes for offloading along with Cloud.

8. Conclusion

In this paper, we proposed an efficient approach to compute QoS-aware offloading policies for serverless functions. Our solution considers a FaaS system deployed in the Cloud-to-Edge continuum, serving users associated with multiple service classes and heterogeneous QoS requirements.

We presented a two-level approach aiming for scalability and reduced computational demand. A simple heuristic algorithm allows FaaS nodes to schedule every incoming request with minimal overhead. Periodically, a linear programming model is solved to determine the best parameters to use in the heuristic, so as to maximize the generated utility over time, given a monetary budget for resource usage. The simulation results we presented show that our approach outperforms all the considered baselines in terms of generated utility across different scenarios and scales well as the size of the problem grows. These results are also confirmed by a proof-of-concept implementation of our policy in a FaaS framework, Serverledge.

For future work, we plan to consider multiple directions to extend our approach. First, while our solution is currently reactive, we intend to consider forecasting techniques to predict, e.g., future workloads as well as varying resource prices, and devise a proactive approach. Furthermore, as sustainability and energy-awareness are increasingly important, we plan to extend our optimization problem to include energy consumption measures. We observe that, in general, introducing additional QoS metrics in the proposed approach is straightforward, only requiring to formulate new constraints or objective terms in the LP problem. However, properly estimating energy consumption of different functions on heterogeneous devices is a major challenge deserving further investigations.

CRediT authorship contribution statement

Gabriele Russo Russo: Writing – original draft, Visualization, Software, Methodology, Investigation, Conceptualization. **Daniele Ferrarelli:** Visualization, Software, Investigation. **Diana Pasquali:** Visualization, Software, Investigation. **Valeria Cardellini:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Francesco Lo Presti:** Writing – review & editing, Supervision, Methodology, Conceptualization.

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) 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).

References

[1] 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.

[2] 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.

[3] G.S. Cassel, V. Rodrigues, R. da Rosa Righi, M. Rosecler Bez, N.A.C., 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.

[4] G. Russo Russo, V. Cardellini, F.L. 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, EEE, 2023, pp. 321–328, http://dx.doi.org/10.1109/PDP59025.2023.00056.

[5] S. Shillaker, P. Pietzuch, Faasm: Lightweight isolation for efficient stateful serverless computing, in: Proc. of 2020 USENIX Ann. Tech. Conf., ATC '20, USENIX Association, 2020, pp. 419–433, URL https://www.usenix.org/system/files/atc20-shillaker.pdf.

[6] 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 Conf., Middleware '20, ACM, 2020, pp. 265–279, http://dx.doi.org/10.1145/3423211.3425680.

[7] F. Lordan, D. Lezzi, 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, in: LNCS, vol. 12820, Springer, 2021, pp. 269–284, http://dx.doi.org/10.1007/978-3-030-85665-6_17.

[8] G. Russo Russo, T. Mannucci, V. Cardellini, F. Lo Presti, Serverledge: Decentralized function-as-a-service for the edge-cloud continuum, in: Proc. of 21st IEEE Int'L Conf. on Pervasive Computing and Communications, PerCom '23, IEEE, 2023, pp. 131–140, http://dx.doi.org/10.1109/PERCOM56429.2023.10099372.

[9] A. Das, S. Imai, S. Patterson, 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, http://dx.doi.org/10.1109/CCGrid49817.2020.00-89.

[10] 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.

[11] G. Russo Russo, A. Milani, S. Iannucci, V. Cardellini, Towards QoS-aware function composition scheduling in Apache OpenWhisk, in: Proc. of 2022 IEEE Int' Conf. on Pervasive Computing and Communications Workshops and Other Affiliated Events, PerCom '22 Workshops, IEEE, 2022, pp. 693–698, http://dx.doi.org/10.1109/PerComWorkshops53856.2022.9767299.

[12] 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.

[13] 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) http://dx.doi.org/10.1145/3508360.

[14] Y. Li, Y. Lin, Y. Wang, K. Ye, C. Xu, Serverless computing: State-of-the-art, challenges and opportunities, IEEE Trans. Serv. Comput. 16 (2) (2023) 1522–1539, http://dx.doi.org/10.1109/TSC.2022.3166553.

[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) http://dx.doi.org/10.1145/3510412.

[16] 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.

[17] 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.

[18] 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.

[19] 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 IEEE/ACM 27th Int'l Symp. on Quality of Service, IWQoS '19, IEEE, 2019, pp. 1–10, http://dx.doi.org/10.1145/3326285.3329055.

[20] 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 Conf. on Cloud Computing, CLOUD '20, IEEE, 2020, pp. 609–618, http://dx.doi.org/10.1109/CLOUD49709.2020.00090.

[21] T. Elgamal, A. Sandur, K. Nahrstedt, G. Agha, Costless: Optimizing cost of serverless computing through function fusion and placement, in: Proc. of 2018 IEEE/ACM Symp. on Edge Computing, SEC '18, IEEE, 2018, pp. 300–312, http://dx.doi.org/10.1109/SEC.2018.00029.

[22] 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.

[23] L. Baresi, D.Y.X. Hu, G. Quattrocchi, L. Terracciano, NEPTUNE: A comprehensive framework for managing serverless functions at the edge, ACM Trans. Auton. Adapt. Syst. (2023) http://dx.doi.org/10.1145/3634750, Just accepted.

[24] B. Wang, A. Ali-Eldin, 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, pp. 239–251, http://dx.doi.org/10.1145/3431379.3460646.

[25] A. Peri, M. Tsenos, V. Kalogeraki, Orchestrating the execution of serverless functions in hybrid clouds, in: Proc. of 2023 IEEE Int'L Conf. on Autonomic Computing and Self-Organizing Systems, ACSOS '23, IEEE, 2023, pp. 139–144, http://dx.doi.org/10.1109/ACSOS58161.2023.00032.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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 Ann. Tech. Conf., ATC '23, USENIX Association, 2023, pp. 879–896, URL https://www.usenix.org/conference/atc23/presentation/sadeghian.

[32] K.L. Li, S. Nastic, AttentionFunc: Balancing FaaS compute across edge-cloud continuum with reinforcement learning, in: Proc. of 13th Int'L Conf. on Internet of Things, IoT '23, 2023.

[33] F. Tütüncüoglu, G. Dán, Joint resource management and pricing for task offloading in serverless edge computing, IEEE Trans. Mob. Comput. (2023) 1–15, http://dx.doi.org/10.1109/TMC.2023.3334914.

[34] 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. Networks 195 (2021) 108177, http://dx.doi.org/10.1016/j.comnet.2021.108177.

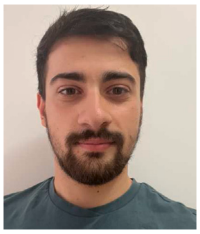
[35] B. Kar, W. Yahya, Y. Lin, A. Ali, Offloading using traditional optimization and machine learning in federated cloud–edge-fog systems: A survey, IEEE Commun. Surv. Tutor. 25 (2) (2023) 1199–1226, http://dx.doi.org/10.1109/COMST.2023.3239579.

[36] 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.

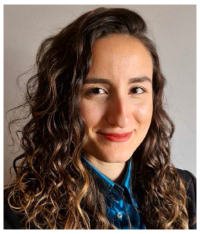
[37] J.D.C. Little, A proof for the queuing formula: L=λw, Oper. Res. 9 (3) (1961) 383–387, http://dx.doi.org/10.1287/opre.9.3.383.

[38] N. Mahmoudi, H. Khazaei, Performance modeling of serverless computing platforms, IEEE Trans. Cloud Comput. 10 (4) (2022) 2834–2847, http://dx.doi.org/10.1109/TCC.2020.3033373.

Gabriele Russo Russo is a research fellow at the University of Rome Tor Vergata, where he received his Ph.D. degree in 2021. His research interests are in the field of distributed computing systems, with emphasis on performance optimization and run-time adaptation. He has served as a reviewer for top-ranked journals and, since 2021, he has been a member of the Technical Review Board of IEEE Transactions on Parallel and Distributed Systems.



Daniele Ferrarelli graduated with honors in Computer Engineering at the University of Rome Tor Vergata, Italy, in 2023. His thesis studied offloading strategies for serverless functions in Edge-Cloud computing environments.



Diana Pasquali graduated with honors in Computer Engineering at the University of Rome Tor Vergata, in 2024. As part of her thesis project, she studied offloading mechanisms and policies for serverless functions in the Edge-Cloud Continuum.

Valeria Cardellini is full professor of computer science at the University of Rome Tor Vergata. She received the Doctorate degree in computer science from the University of Rome Tor Vergata in 2001. Her research interests are in the field of distributed computing systems, with a focus on Cloud systems and services. She has more than 110 publications in international conferences and journals. She is Associate Editor of IEEE TPDS and Elsevier JPDC. She was general co-chair of ACM/SPEC ICPE 2023, TPC co-chair of IEEE/ACM UCC 2018 and IEEE ICFC 2020. She has served as TPC member of conferences on performance and Web and as frequent reviewer for highly ranked journals.

Francesco Lo Presti is full professor of computer science at the University of Rome Tor Vergata. He received the Doctorate degree in computer science from the University of Rome Tor Vergata in 1997. His research interests include measurements, modeling and performance evaluation of computer and communications networks. He has more than 100 publications in international conferences and journals. He has served as TPC member of conferences on networking and performance areas, and as reviewer for various international journals.