VioLinn: Proximity-aware Edge Placement with Dynamic and Elastic Resource Provisioning

Klervie Toczé, Ali Jawad Fahs, Guillaume Pierre, Simin Nadjm-Tehrani

▶ **To cite this version:**

Klervie Toczé, Ali Jawad Fahs, Guillaume Pierre, Simin Nadjm-Tehrani. VioLinn: Proximity-aware Edge Placement with Dynamic and Elastic Resource Provisioning. ACM Transactions on Internet of Things, 2023, 4 (1), pp.1-31. 10.1145/3573125 . hal-03869221

HAL Id: hal-03869221
https://inria.hal.science/hal-03869221

Submitted on 24 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

VioLinn: Proximity-aware Edge Placement with Dynamic and Elastic Resource Provisioning

KLERVIE TOCZÉ, Linköping University
ALI J. FAHS, Activeeon; Univ Rennes, Inria, CNRS, IRISA
GUILLAUME PIERRE, Univ Rennes, Inria, CNRS, IRISA
SIMIN NADJM-TEHRANI, Linköping University

Deciding where to handle services and tasks, as well as provisioning an adequate amount of computing resources for this handling, is a main challenge of edge computing systems. Moreover, latency-sensitive services constrain the type and location of edge devices that can provide the needed resources. When available resources are scarce there is a possibility that some resource allocation requests are denied.

In this work, we propose the VioLinn system to tackle the joint problems of task placement, service placement and edge device provisioning. Dealing with latency-sensitive services is achieved through proximity-aware algorithms that ensure the tasks are handled close to the end user. Moreover, the concept of *spare* edge device is introduced to handle sudden load variations in time and space without having to continuously overprovision. Several spare device selection algorithms are proposed with different cost/performance trade-offs.

Evaluations are performed both in a Kubernetes-based testbed and using simulations and show the benefit of using spare devices for handling localized load spikes with higher quality of service (QoS) and lower computing resource usage. The study of the different algorithms shows that it is possible to achieve this increase in QoS with different trade-offs against cost and performance.

CCS Concepts: • **Networks** → **Cloud computing**; **Network resources allocation**.

Additional Key Words and Phrases: Edge/fog computing, resource management, Kubernetes, elasticity.

1 INTRODUCTION

The rapid development of Internet of Things (IoT) creates a paradigm shift where a majority of enterprise data are no longer produced within the data centers before being processed there [63]. Instead, an increasing fraction of IoT data are produced far from the traditional data centers, which creates significant pressure on long-distance networks to carry large amounts of data to be processed elsewhere [40]. The emerging field of mobile edge computing aims to extend the traditional data centers with additional compute, storage and networking resources located close to the sources of traffic so a majority of the data processing workload can be handled locally [5]. Edge computing promises faster reaction times, reliable operations with intermittent connectivity, and greater control of the overall data processing pipelines.

Driven by the promise of lower delays, new application areas are getting more and more attention such as virtual and augmented reality [7, 8] or gaming. Sega is for example investigating fog gaming, where Sega consoles may be used as edges devices to reduce the lag to less than a millisecond [22].

The geographical spread of the sources of traffic encourages the creation of a large number of "points-of-presence," each with relatively modest quantity of resources, where locally-generated data may be processed. Tasks generated by users need to be sent for execution to a device for 1) providing the required service and 2) completion with low enough latency to ensure that the quality of service (QoS) requirements are met. Placing these tasks on suitable edge devices in order to provide high QoS is challenging, especially in a context where the load is changing not only over time but also in terms of geographical location. Indeed, it is not possible to provision all the

Authors' addresses: Klervie Toczé, klervie.tocze@liu.se, Linköping University; Ali J. Fahs, ali.fahs@activeeon.com, Activeeon; , Univ Rennes, Inria, CNRS, IRISA; Guillaume Pierre, guillaume.pierre@irisa.fr, Univ Rennes, Inria, CNRS, IRISA; Simin Nadjm-Tehrani, simin.nadjm-tehrani@liu.se, Linköping University.

points of presence with enough resources to handle any type of load at all times, as it would mean severely over-provisioning the edge infrastructure.

In this article, we study the delivery of high QoS for tasks requiring low latency without over-provisioning. We propose the notion of *spare device* which may be dynamically added to the edge/fog computing platform. Spare devices may be server machines located inside or close to the points-of-presence, and are normally utilized for non-critical tasks such as an unused A-SAN node in a 5G system [41]. When the points-of-presence of a given area cannot handle the incoming tasks in a satisfactory manner, these spare devices may be temporarily reassigned to host extra edge services. Another possible scenario is "edge bursting" where one edge provider may temporarily lease local resources from an adjacent *spare resource provider* for the same purpose. Of course, these extra resources do not come for free so cost needs to be taken into account and weighted against QoS when deciding on where to execute user tasks.

This article is a joint extension of two conference papers [18, 62]. Voilà [18] presents a proximity-aware autoscaler integrated in the popular Kubernetes container orchestration framework to automatically (re-)deploy fog applications on the minimum set of servers that guarantees their quality-of-service requirements. Voilà, however, does not consider the dynamic addition of extra server resources to handle local overload situations. ORCH [62] presents an edge orchestration framework for managing a combination of mobile and stationary edge resources. The framework is populated with deadline-aware algorithms that ensure that time-constrained tasks are handled by a nearby edge resource. The framework however has only been evaluated with a proof-of-concept simulation and the model does not include the notion of elasticity in service provisioning.

The current article builds upon the above works and makes the following additional contributions:

- We propose the concept of *spare edge device* to handle dynamic load changes in an elastic way, as well as algorithms for provisioning these devices with different QoS/cost trade-offs.
- We integrate these algorithms and the associated cost model in a novel system named VioLinn, which leverages elements of the ORCH framework and the Voilà stack. Specifically: (i) We create a cost model for the used resources to enable a distinction between dedicated and spare devices; (ii) We extend the implementation of Voilà placement and autoscaling algorithms.
- We perform an extensive evaluation of the QoS-cost trade-offs in a physical testbed in order to measure actual latencies and overheads. We also perform scalability studies in a simulator.

Our evaluations use a physical testbed to show that dynamic provisioning of spare resources when the permanent resources do not provide adequate placement options is viable with regards to QoS. Moreover, the simulations show the scalability of the method. Evaluation of different spare device selection alternatives shows that the presented set of algorithms can handle localized load spikes, and that the cost for using the extra edge devices can be efficiently controlled.

2 SYSTEM DESIGN

2.1 Motivational use case

As an illustration of the problem, we consider the Open Radio Access Network (Open-RAN). In 5G, the base station and core network software is decoupled from the hardware running it, hence enabling the virtualized base station concept [26]. This makes it possible for companies selling network infrastructure and services to provide only the hardware, only the software or both parts.

In a telecommunication network, there is always a high variation in the incoming load, both in time and space, as end users are mobile. The traditional way of handling this has been to over-provision the network resources in order to provide good QoS even during load peaks. This, however, results in acquiring and maintaining resources that are going to be idle the majority of the time, which is neither profitable nor sustainable.

The virtualized base station makes it possible to deploy 5G Open-RAN software onto supplementary computing resources, not necessarily the one present together with the radio equipment. However, parts of the 5G Open-RAN software (e.g. related to encryption) are still required to be close (latency-wise) to the radio. Hence, this opens for the possibility (on the software side) to avoid over-provisioning by introducing the possibility to add extra resources at the edge only when required, as proposed in this work. This enables to maintain a good QoS while improving the profitability for the service provider. One such company, IS-Wireless[1] is advocating the use of Virtual Network Functions in their 5G Open-RAN software, which has as one benefit the "ease of deployment on any computing resource including dedicated and shared ones" [67].

Therefore, a system running Open-RAN software components ought to leverage both dedicated and shared resources, depending on the current incoming load. This system should dynamically deal with the two types of resources in order to achieve a good QoS and at the same time, an increased profitability. Traditional fog and edge systems only consider dedicated resources while this paper proposes a solution that includes the two types.

2.2 Problem description

The complex problem of managing a dynamic load at edge devices can be decomposed to the following three subproblems.

The **task placement subproblem**: for each task r of service S incoming to the closest edge device e , e has to decide on which service instance s located on edge device e' the task r will be executed so that s has access to enough resources to compute r according to its latency requirements.

The **service placement subproblem**: only a subset of the edge devices contain the required environment to run a service at a given point in time (i.e. the service instances). This avoids reserving resources for a service which is not used by users in this area at the moment. The subproblem consists in determining this subset after every major load change such that the incoming load is served with as high QoS as possible.

The **edge device provisioning subproblem**: sometimes, handling a load change is only possible by increasing the amount of resources (namely the edge computing devices) available at a certain location. Therefore, at every load change requiring it, this subproblem consists of enabling minimum extra resources (mobile or stationary) at specific locations such that the incoming load is served with as high QoS as possible.

2.3 System model

In this work, a system model comprising two layers is used: one for the edge devices and one for the end devices (i.e. the ones used by the end user). A third layer that would add connectivity to the cloud is only going to make the problem easier by increasing the amount of resources. Therefore, we focus on the harder case where the edge infrastructure cannot fall back on cloud resources due to latency requirements. Example cases would be locally generated augmentations to virtual reality or autonomous vehicles. Indeed, such services require end-to-end latency between 10 to 20 ms [2, 13] when the network latency to reach a cloud datacenter is estimated to 20-40 ms over a wired connection and up to 150 ms over 4G mobile networks [45]. Not even 5G can support this, as the user-to-Internet latency is reported as in the range 30-40 ms [12, 64].

This edge system model is presented in Figure 1, where edge layer devices are depicted in the upper layer and the end devices in the lower layer. The figure shows devices belonging to three edge orchestration areas, corresponding to three geographical areas. Of course, when they are

[1]https://www.is-wireless.com/

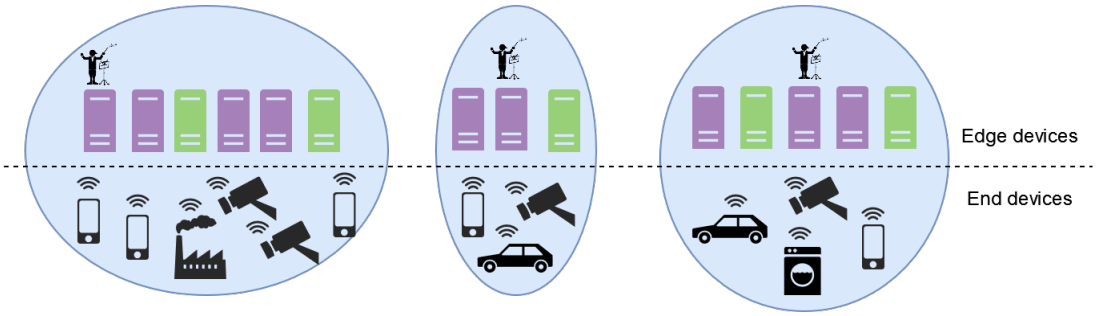


Fig. 1. System model (the blue ovals represent the edge orchestration areas).

mobile, devices can change edge orchestration area over time. The edge devices are connected together using e.g. fiber.

The end devices create a load which has to be served by the edge devices. This load is varying over time and over geographical locations, depending on the movements of the end devices and what they are used for. When the variation is major and sudden, a *load spike* appears in the system.

Moreover, the notion of dedicated and spare edge devices is introduced to represent the studied way of provisioning extra resources for handling sudden load spikes. The *dedicated edge devices* are those that the edge infrastructure provider has permanently allocated for serving the given edge orchestration area. For cost reasons, the provider dimensions the infrastructure according to a pre-defined anticipated load. In order to be able to handle unpredicted load spikes, the edge infrastructure provider secures additional access to *spare edge devices*. These are extra edge devices, not necessarily belonging to the edge infrastructure provider, which can be made accessible for temporary access at a contract-based price. In Figure 1, the dedicated edge devices are depicted in purple and the spare edge devices in green.

In our motivational use case, IS-Wireless mentions the use of "flexible resource pools" as a key factor for profitability while maintaining high QoS [26]. These pools enable the telecom integrators and operators providing 5G to have access to resources from their own local servers (what we call dedicated resources), but also from public edge providers or from IS-Wireless data centers [26]. The latter two can be shared by different actors and are examples of what we call spare resources.

To summarize, each edge orchestration area is composed of end devices and a set Δ of n edge devices $\Delta = \{\delta_1, ..., \delta_n\}$. Among those edge devices, a few are spare (denoted as $\delta_i \in \Delta^S$ or δ_i^S) whereas the rest are dedicated (denoted as $\delta_i \in \Delta^D$ or δ_i^D, with $\Delta = \Delta^S \cup \Delta^D$ and $\Delta^S \cap \Delta^D = \emptyset$). At any point in time, one dedicated edge device $\delta \in \Delta^D$ is responsible for area orchestration in each area. This edge device is referred to as *area orchestrator*. It is depicted with a conductor icon on top in Figure 1 and will be denoted by o in the rest of the paper.

2.4 Area model

In this work, edge devices are spread out geographically within the orchestration area. We consider that there exist specific positions called *serving positions* from which an edge device can receive load. This corresponds either to the location of a stationary edge device (e.g., a base station), or to a fixed position at which a mobile edge device can stay while serving tasks (in order to save resources that are otherwise used for navigation).

Each edge device has a connectivity range and end devices connect to one edge device at a time. This connection can use e.g. 5G. Details about how this connection happens (e.g., how the end

device selects which edge device to connect to) are out of the scope of this paper. It is therefore not required to know the exact position of all end devices, but instead to know which load they create on the different edge devices.

2.5 Service model

The end users of the system need to access different services. We consider the case where the service code is running at the edge, the end devices send *tasks* to the edge for the service to execute, and receive the results of the task execution. Thus, a task contains the required input for the service.

Moreover, not all edge devices should provide every service environment, which is a software application instance (also called *application replica* or replica for short) for all services, as this would be an inefficient use of the (scarce) edge resources (e.g., memory). It is assumed that only one replica for a given service is deployed on a specific edge device.

For a given service S, there exist numerous ways of selecting which edge device will host a replica. These different ways are called *service placements*. Considering the p-th such possible service placement ω_p^s, it is a set of K server devices, corresponding to the edge devices hosting a replica for this service. In other words, $\omega_p^s = \{\delta_1, \delta_2, ..., \delta_K\}$, $\delta_i \in \Delta$. Where ambiguity does not arise, we do not use the service and placement index in the rest of the paper, but ω denotes the currently considered placement out of all the possible ones for a given service. Among the devices included in a given placement, a mix of spare and dedicated devices is considered.

2.6 Resource model

The first resource type is computational resources. An edge device is resource-constrained and therefore cannot handle more than a certain number of tasks per unit of time. The resources are split among the different services running on the edge device at one time. Details about allocation are out of the scope of this paper but different schemes can be envisioned such as each service getting an equal share or some critical services getting a greater share of the computational resources.

The second resource type is communication resources. For modeling those, the latency between edge devices is calculated, proportionally to the geographical distance between two nodes. This calculation can later be extended to account for bandwidth constraints or queuing time.

2.7 Cost model

Our cost model is based on the timeline of operation which is divided into *cycles* of identical duration. Edge devices can only join or leave the orchestration area at the end of a cycle.

The cost of using an edge device is divided into three parts:

- The *activation* cost (C_a), for adding the device into the orchestration area.
- The *replica* cost (C_r), for creating and starting-up an application replica on the device.
- The *utilization* cost (C_u), for using a device during the current cycle.

Dedicated devices are a part of the edge infrastructure that is always ready to handle edge load, i.e. they are active all the time and considered a permanent part of the orchestration area. Their activation cost $C_a^\delta(\delta \in \Delta^D)$ can therefore be ignored during the orchestration, as these nodes are only added when the area is created. The utilization cost of dedicated devices $C_u^\delta(\delta \in \Delta^D)$ consists in electricity and maintenance cost for example.

Spare devices do not necessarily belong to the edge infrastructure provider (even if they can), and are not part of the orchestration area at all times. There is therefore a cost associated to activating these devices so that they become part of the orchestration area (so we have $C_a^{\delta_1} > C_a^{\delta_2}$ for $\delta_1 \in \Delta^S$ and $\delta_2 \in \Delta^D$). This cost can be seen as a fee paid to be able to use the spare device by connecting it to your orchestration area. This cost only applies during the cycle when the device is added.

Regarding the utilization cost, in addition to the components representing maintenance and energy costs, the spare devices also have an additional component, namely device temporary acquisition costs representing a compensation for the fact that the resources on this device cannot be used for another task or have to be rented. Therefore, it is assumed that, for equivalent devices δ_1 and δ_2, $C_u^{\delta_1} > C_u^{\delta_2}$ for $\delta_1 \in \Delta^S$ and $\delta_2 \in \Delta^D$.

Therefore, the per cycle cost C_d of having a device δ active in the orchestration area can be expressed as:

$$C_d^\delta = \begin{cases} C_u^\delta & \text{if } \delta \in \Delta^D \text{ or } \delta \in \Delta^S \text{ with } \delta \in \omega \\ & \text{the previous cycle} \\ C_u^\delta + C_a^\delta & \text{if } \delta \in \Delta^S \text{ with } \delta \notin \omega \text{ the previous cycle} \end{cases} \tag{1}$$

And the per cycle cost of a service placement ω containing K devices is:

$$C^\omega = \sum_{i=1}^{K} C_d^\delta + C_r^{\delta_i} \tag{2}$$

where C_r^δ varies over cycles:

$$C_r^\delta = \begin{cases} c_r^\delta & \text{if no application replica on } \delta \text{ in the previous cycle} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where c_r^δ is the cost of creating and starting-up an application replica. An application replica is started up when created and runs until it is removed, i.e. it is not restarted every cycle.

This model is thought to include incentives for adding a spare node or creating a replica only when it is necessary for the QoS, as these actions introduce some set-up overhead during which the resources are used but cannot handle tasks. The activation and replica costs account for this.

This is a generic model that allows fine-grained business models varying over device classes and providers. However, for simplicity, in the rest of the paper we will assume that the utilization costs are the same for all edge devices of the same type, i.e. $\forall \delta_i^D, C_u^{\delta_i} = c_u^D$ and $\forall \delta_i^S, C_u^{\delta_i} = c_u^S$. We also assume that the activation cost of all spare devices is the same, i.e. $\forall i, C_a^{\delta_i^S} = c_a$, and that the cost of creating an application replica c_r^δ is the same for all edge devices, so $\forall i, c_r^{\delta_i} = c_r$.

2.8 Quality of service

For latency-sensitive services, high QoS requires that a certain proportion of tasks needed to deliver the service can be handled fast enough to meet the latency constraints. Thus, tasks should be sent for execution to a device 1) that is in close proximity of where the task enters the systems (to avoid long transmission delays) and 2) that has sufficient resources to handle it immediately (to avoid having queuing delays or offloading delays to another device).

However, some tasks complete faster than others. In our system, since we assume soft real-time guarantees, we need the proportion of tasks that do not satisfy their timing constraints to be below a certain threshold. For example, virtual reality frames should typically be rendered in less than 15 ms to avoid motion sickness [13]. Hence, we refer to a task that is **not** fast enough for satisfying the time constraints as a *slow* task, i.e. a task that missed its (soft) deadline. This happens when the network communication latency between the receiving and the executing edge devices is too high and/or the task is sent for execution to an overloaded replica (i.e. a replica without free resources).

The QoS of the system $\mathcal{E}^\omega$ is defined as the percentage of slow tasks among all received tasks in the system when using a given placement. The relevance of this indicator was shown when

evaluating the Voilà stack [18]. The calculation is therefore:

$$\mathcal{E}^{\omega} = 100 * \frac{\text{\# Slow tasks}}{\text{\# Received tasks}} \tag{4}$$

For a given service, the service provider and the edge system provider agree on a QoS threshold denoted $\mathcal{E}_0$. A service and edge placement ω is then acceptable if $\mathcal{E}^{\omega} < \mathcal{E}_0$.

2.9 Placement fitness

In order to determine how well a given placement balances the need for high QoS and the wish for a low cost, the notion of placement fitness is introduced. To achieve high fitness, the elements characterizing a bad placement, with regards to QoS (Section 2.8) and cost (Section 2.7), should be minimized. Those are therefore gathered in what is called the bad-fit score.

The bad-fit score of a placement $\mathcal{BF}^{\omega} \in \mathbb{R}$ expresses a trade-off between quality and cost. A low bad-fit score indicates a good fitness of the placement (low percentage of slow tasks and low costs).

The bad-fit score is expressed as:

$$\mathcal{BF}^{\omega} = \alpha * \mathcal{E}^{\omega} + (1 - \alpha) * \left(\frac{C^{\omega}}{C_{max}}\right) \tag{5}$$

where $\alpha \in [0, 1]$ is a weight parameter enabling the area orchestrator to favor performance (high value of α) or cost efficiency, and C_{max} is the maximum possible cost of a placement, corresponding to including all available dedicated and spare devices in a placement.

3 RESOURCE PROVISIONING WITH VIOLINN

We now introduce VioLinn[2], a system with orchestration policies for a complex infrastructure including multi-provider cooperation. Then, we present the algorithms used to instantiate the VioLinn components through two orchestration levels and to provide a QoS-aware and cost-aware service over geo-distributed devices with a load varying in time and space.

3.1 Overview

The VioLinn system is shown in Figure 2. It is organized as follows: three strings (the vertical blue bars) corresponding to the three subproblems of Section 2.2 and a monitoring bow (the horizontal green bar) which can be used over all the three strings. On the body of the VioLinn (purple part), we place management components which are the basic blocks upon which the other parts are built. The specific contributions of this paper reside in the right-hand side and are highlighted with darker color tones. For the rest of the components, we reuse existing works (the pink ovals).

The first string handles the task placement subproblem. It comprises a *task characterizer* which analyzes the incoming task and characterizes it according to what is important for the orchestration system, and a *task placer* which decides on which device the incoming task should be executed based on its characteristics. VioLinn relies on Proxy-mity [16] to handle this sub-problem.

The second string is about the service placement subproblem. Its two components, the *service placer* and the *autoscaler* determine on which devices the service instances should be deployed (the placer part) as well as how many of them are necessary for serving the incoming load (the autoscaler part). VioLinn relies on Voilà [18] to handle this sub-problem.

The third and last string handles the edge device provisioning subproblem, i.e. adding or removing spare devices to the orchestration area. Its first component, the *edge device broker* is in charge of defining the contracting terms under which the spare devices can be included (e.g. what Service

[2]The name is a contraction of the musical instrument used as an analogy for our system and of the Swedish name Linn, one of the given names of the "production baby" of this paper.

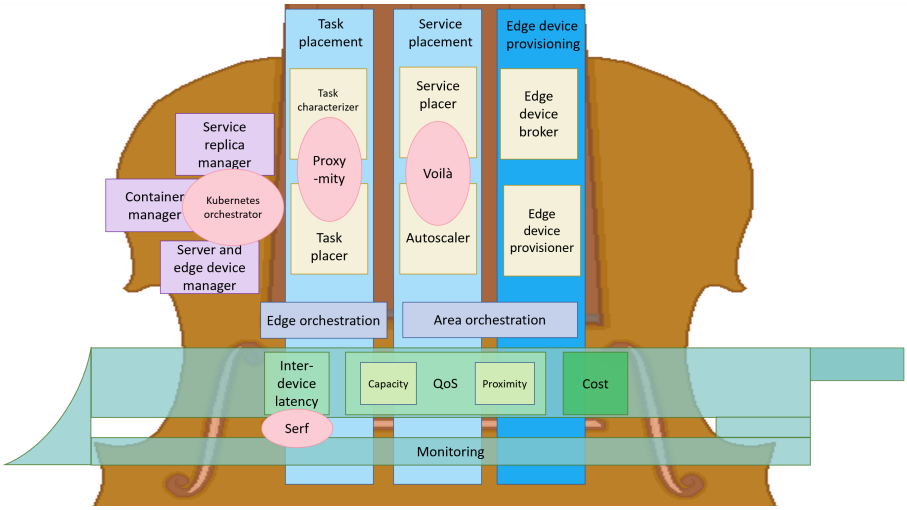


Fig. 2. The VioLinn system.

Level Agreement is concluded). The second component, the *edge device provisioner* decides when it is necessary to add/remove a spare device and which one to select.

Over the three strings is the monitoring bow, which keeps track of the attributes needed by the string components. Here, each edge device keeps track of the latency between itself and the other edge devices (using Serf [21]) while placement QoS and cost are calculated at the system level, in the area orchestrator.

When playing the VioLinn, two different orchestrations take place, depicted as horizontal rectangles over the strings in Figure 2. The **edge orchestration** is fully distributed as it takes place at every edge device, every time a task is received. It is about handling the incoming tasks. The **area orchestration** is distributed per orchestration area: in each of these areas one edge device is selected to take the role of the area orchestrator. The aim of this part of the orchestration is to deal with how many edge devices and service replicas are needed for handling the current load.

3.2 Edge orchestration

Edge orchestration is illustrated by Algorithm 1. It can be split into two parts: one that executes every time a task comes to one of the edge devices; and one that executes periodically in order to gather relevant information about the other edge devices present in the orchestration area.

Lines 1-4 correspond to the leftmost string in the VioLinn system. Upon receiving a task, it is first characterized (line 2). In this work, the focus is on latency and all tasks carry information about a relative deadline compared to their arrival time (i.e., a latency threshold) that should not be exceeded when handling the task. The task are therefore tagged in a straightforward manner with this information. Then, the orchestrator decides where the task should be placed/routed (line 3) before it is sent for execution (line 4).

Algorithm 1: Edge orchestration

```
// Runs at each edge device e
parameter: T_d // Monitoring period (cycle)
state      : L =<< n, l >> // List of
             monitored latency l for each
             reachable n
1 upon receiving task r do
      // Task characterizer
2     tag ← Characterize(r);
      // Task placer
3     device ←PlaceTask(r, tag, L);
4     sendForExecution(r, device)
5 every T_d do // Monitoring
6     N ← GetReachableEdgeDevices(e);
7     foreach n ∈ N do
8         Update l for n with GetLatency(n);
```

Algorithm 2: Area orchestration

```
parameter: T_d // Monitoring period (cycle)
state      : I // Provisioning information
             ω // Service placement
// Runs at each area orchestrator o
1  every T_d do
2      <actionNeeded, I> ← MonitorLoad(o);
3      if actionNeeded then
4          ω_c ← GetCurrentPlacement(o);
5          if I == UnderProvisioning then
6              ω_n ← FixUnderprovisioning(ω_c);
7          if I == OverProvisioning then
8              ω_n ←
                 CheckOverprovisioning(ω_c);
9          Deploy the new placement;
10         ω ← ω_n;
```

The part of the monitoring bow which executes on each device is presented on lines 5-8. In this work, the focus is on task completion latency by prioritizing the placement on the closest devices, where closeness is measured in terms of communication latency. Therefore the monitoring bow estimates the latency from the current device to all other devices in the orchestration area.

3.3 Area orchestration

Area orchestration is illustrated by Algorithm 2. This orchestration takes place periodically and has two parts. When the information from the monitoring bow indicates a need for an action (due to under/overprovisioning of edge resources), the orchestrator proposes a solution for the service placement and edge device provisioning subproblems of the problem presented in Section 2.2 based on which edge devices are available at this moment. This new placement is then executed.

3.3.1 Monitoring. We expect that the load coming into the system is dynamic and can change from one cycle to another. It is therefore important to monitor it in order to detect when the edge orchestration is overprovisioning or underprovisioning. Monitoring takes place on line 2 of Algorithm 2 and Function 3 presents details of it.

On the one hand, the load is monitored for underprovisioning (line 6-10 of Function 3) through the percentage of slow tasks among all received tasks for the current service placement ($\mathcal{E}^{\omega}$), as described in Section 2.8. A load which requires action due to underprovisioning is detected if the percentage of slow tasks exceeds a pre-defined threshold representing the QoS that the edge infrastructure provider aims to provide (line 6 of Function 3).

On the other hand, the load is monitored for overprovisioning (line 11-14 of Function 3). When the slow task threshold is not exceeded for a fixed period of time (t_o detection periods, on line 11 of Function 3), the system should investigate if resources (especially from spare devices) are not being kept in use without load to serve.

3.3.2 Action mechanisms. Once the monitoring bow has detected an over/underprovisioning situation that should be taken care of, the four components of the service and edge device strings collaborate to find a new service placement on a possibly new set of edge devices that will serve the new load with the chosen QoS-cost trade-off level. To do that, three alternatives are possible:

- **Replica replacement**: Keeping the current set of edge devices and the same number of replicas; some replicas are placed on other edge devices.
- **Replica autoscaling**: Keeping the current set of edge devices; the number of replicas is increased/decreased and a new service placement is chosen.
- **Edge autoscaling**: The set of edge devices is changed by increasing/decreasing the number of spare devices included (together with the corresponding application replicas).

Function 3: Monitoring for under/over provisioning.

```
parameter: E_o // Slow tasks % threshold
           t_o // Overprovisioning threshold
input    : o // Area orchestrator
output   : actionNeeded // Monitoring result
           I // Provisioning information
state    : τ // Overprovisioning counter (initially 0)
1  Function MonitorLoad(o)
2      ω ← GetCurrentPlacement(o);
3      E^ω ← CalculatePlacementQuality(ω);
4      I ← ∅;
5      actionNeeded ← false;
       // Check for underprovisioning
6      if E > E_o then
7          actionNeeded ← true;
8          I ← UnderProvisioning;
9      else
10         τ ← τ +1;
       // Check for overprovisioning
11     if τ == t_o then
12         τ ← 0;
13         actionNeeded ← true;
14         I ← OverProvisioning;
```

Function 4: Handling underprovisioning

```
input  : ω_c // Current service placement
output : ω_n // New service placement
1  Function FixUnderprovisioning(ω_c)
2      E^ω_c ← CalculatePlacementQuality(ω_c);
3      while E^ω_c > E_0 and not timeout do
           // Try replacement
4          Ω_P ← CreateNewCombinationsReplace();
5          if ∃Ω_S ⊆ Ω_P, ∀ω ∈ Ω_S, E^ω < E_0 then
6              ω_c, E^ω_c ← GetBestPlacement(Ω_S);
7          else
               // Try scale up with dedicated
8              Ω_P ← CreateNewCombosScUpDedicated();
9              if ∃Ω_S ⊆ Ω_P, ∀ω ∈ Ω_S, E^ω < E_0 then
10                 ω_c, E^ω_c ← GetBestPlacement(Ω_S);
11             else
                   // Try scale up with spare
12                 Ω_P ← CreateNewCombosScUpSpare();
13                 if ∃Ω_S ⊆ Ω_P, ∀ω ∈ Ω_S, E^ω < E_0
                   then
14                     ω_c, E^ω_c ← GetBestPlacement(Ω_S);
           // No acceptable QoS improvement
           // Pick the placement ω with lowest E^ω as ω_c for the next round
15         ω_c, E^ω_c ← GetHighestQoSPlacement(Ω_P);
16     ω_n ← ω_c;
```

The service placer and autoscaler (middle string in Figure 2) of VioLinn are responsible for the first two alternatives, respectively. Edge autoscaling is handled by the edge device broker and edge device provisioner (rightmost string in Figure 2) together. In this work, we opt for the simplest edge broker possible, i.e. assume that we have a pool of available spare edge devices that are available at all times, with a fixed price.

In Algorithm 2, handling over and underprovisioning is performed on lines 3-10. It uses two different helper functions depending on whether underprovisioning or overprovisioning was detected. The logic in both (detailed in Sections 3.3.3 and 3.3.4) is similar, the first one taking care of moving and adding replicas or spare devices while the second takes care of moving and removing replicas or spare devices.

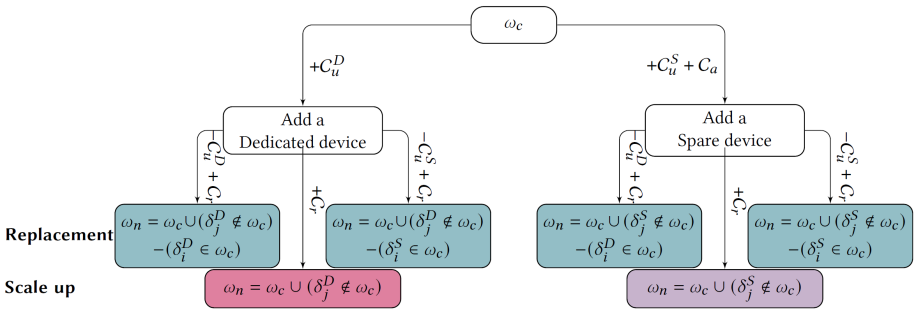


Fig. 3. Underprovisioning: creating combinations and calculating their costs.

3.3.3 Handling underprovisioning. For handling underprovisioning (Function 4), the algorithm works incrementally, trying to change the service placement and edge device provisioning by considering a one-step change per iteration (i.e., moving/adding one replica or adding one edge device, then two, etc..). The main idea is to generate possible placements (i.e. combinations of edge devices on which replicas will be deployed) according to the three possible alternatives taken in the following order: replica replacement (line 4), replica autoscaling (line 8) and edge autoscaling (line 12). The reason for choosing this order is due to the cost model chosen: this order ranks the alternatives from the cheapest to the most expensive one. In each set of possible combinations for one alternative (denoted as Ω_P), we first compute the subset of possible combinations satisfying the QoS requirements Ω_S (lines 5, 9 and 13). Then, the most suitable one (i.e., best) is selected according to the wanted QoS-cost trade-off level (lines 6, 10 or 14). Different variants for this trade-off level are described in Section 5.4. If no suitable edge placement can be found during one iteration (i.e., the QoS threshold is not reached), the best placement with regards to QoS is selected as the base placement for the next iteration (line 15). This continues until a solution is found or a timeout is reached (line 3).

Figure 3 illustrates how the combinations are generated and how the cost of a new placement (ω_n) is calculated compared to the current one (ω_c) in the case of underprovisioning. The combinations are generated by iterating index i over the edge devices present in ω_c and iterating index j over all other edge devices not currently included in ω_c. The cost of a leaf is the sum of all the costs on the branches leading to this leaf. Finally, a color code shows which of the three alternatives described in Section 3.3.2 a leaf corresponds to. First, the upper line (in blue) is for replica replacement, which happens when the number of replicas is constant but the location of a replica is changed, either from a dedicated device to another dedicated device, from a spare device to a dedicated device, from a dedicated device to a spare device or from a spare device to another spare device. On the lower line, we have the leaves corresponding to scaling up: first a red one corresponding to replica autoscaling, i.e. a new replica is created on a dedicated device that did not have one yet and then a purple one corresponding to edge autoscaling, i.e. where a spare device is added to the orchestration area together with the corresponding replica.

3.3.4 Handling overprovisioning. For handling overprovisioning (Function 5), the algorithm also works incrementally with the objective of removing spare devices (and the corresponding replicas) and/or decrease the number of replicas without violating the QoS requirements. First, we calculate how large the overprovisioning is (line 3) based on the current load level and the number of edge devices handling load. The algorithm will target to remove this overprovisioning. The first step is to create combinations that remove replicas from spare devices (line 5) as these are the most

expensive to use. If this is not possible, i.e. such combinations lead to QoS violations, combinations removing replicas on dedicated devices (line 9) are tested. The first suitable combination is used as the basis for the next recursion and the recursion stops when either the calculated overprovisioning is reached or if it is not possible to scale down (line 4).

Function 5: Checking if overprovisioning and handling it if it is the case

```
   input :ω_c // Current service placement
   output:ω_n // New service placement
1  Function CheckOverprovisioning(ω_c)
2      E^{ω_c} ← CalculatePlacementQuality(ω_c);
3      scdownObj ← CalculateScaleDownObjective(ω_c);
4      while E^{ω_c} < E_0 and scdownObj not reached do
5          Ω_P ← CreateNewCombosScDownSpare();
6          if ∃Ω_S ⊆ Ω_P, ∀ω ∈ Ω_S, E^ω<E_0 then
7              ω_c, E^{ω_c} ← GetFirstPlacement(Ω_S);
8          else
9              Ω_P ← CreateNewCombosScDownDedicated();
10             if ∃Ω_S ⊆ Ω_P, ∀ω ∈ Ω_S, E^ω<E_0 then
11                 ω_c, E^{ω_c} ← GetFirstPlacement(Ω_S);
12     ω_n ← ω_c;
```

Figure 4 illustrate how the combinations are generated and how the cost of a new placement (ω_n) is calculated compared to the current one (ω_c) for the case of overprovisioning. It works in the same way as Figure 3. A first difference is that for replica replacement (the blue leaves), the location is changed from a dedicated device to another dedicated device, from a dedicated device to a spare device, from a spare device to a dedicated device or from a spare device to another spare device (i.e. the two middle ones are inverted compared to underprovisioning case). Moreover, the lower line is this time for scale down, with a red one (replica autoscaling), when a new replica is removed from a dedicated device and then a purple one (edge autoscaling), when a spare device is removed to the orchestration area together with the corresponding replica.

4 TESTBED IMPLEMENTATION

VioLinn is implemented in Kubernetes, a mainstream cloud orchestration engine, and a potential one for future edge and fog orchestration [48, 50, 58, 60]. VioLinn relies on a modified version of the Voilà stack [15, 18] which creates a bridge from theory to practice.

In particular, VioLinn leverages the following features of Voilà: pairwise inter-node latency approximation using Vivaldi coordinates [11], a monitor of the load coming from the edge devices receiving tasks and redirected towards one of the replicas, latency-aware routing using Proxy-mity [16], tail-latency-aware placement, tail-latency-aware autoscaling, and finally health check mechanisms to make sure the current placement is achieving the promised QoS. For example, the edge orchestration algorithm is implemented using Proxy-mity and the Voilà monitoring part.

VioLinn orchestration presents novel challenges that require several extensions to the Voilà stack to integrate additional support of edge devices.

4.1 Kubernetes

Kubernetes is an open-source orchestration engine which automates the deployment, scaling and management of containerized applications [60]. As shown in Figure 5a, a Kubernetes cluster consists

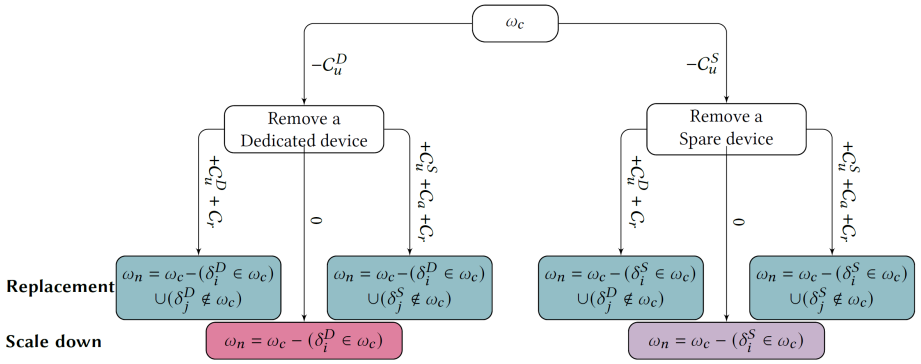
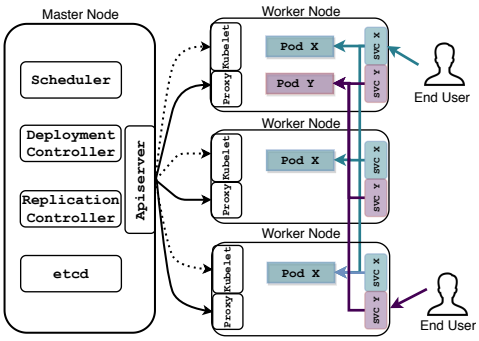
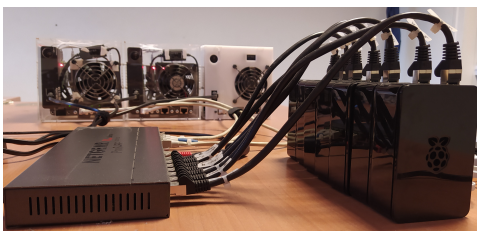


Fig. 4. Overprovisioning: creating combinations and calculating their costs.



(a) Kubernetes architecture.



(b) The research testbed used for evaluation.

Fig. 5. Kubernetes architecture and testbed.

of a master node responsible for the monitoring and the management of the deployed applications (including resource discovery using etcd), and any number of worker nodes that constitute the system's computing, network, and storage resources. Each worker node has two daemons: the *Kubelet* daemon which periodically checks the scheduler decision and deploys any pods assigned to it, and the *kube-proxy* daemon which takes care of local IP routing.

In Kubernetes, an application deployment is composed of a set of *pods*, defined as a set of logically-related containers and data volumes to be deployed on a single machine. Application replication is achieved by deploying multiple identical pods. These pods can then be exposed to external end users as a single entity by creating a *service* which exposes a single IP address to the end users and acts as a front end that routes requests to one of the corresponding pods.

The Kubernetes scheduler is in charge of determining which pod will be placed in which worker node. By default Kubernetes uses latency-*un*aware filtering/scoring algorithms. To realize the functions described in Section 3, we need to design a new scheduler which controls both the number of replicas and their placement within the fog computing infrastructure. This provides QoS guarantees across large variations in the end-user request workload.

4.2 Extending Voilà

Voilà is implemented as an extension of Kubernetes, and it also provides a simulation tool to evaluate large-scale clusters, thus, ensuring the scalability of the solution. As a result, we also implement and evaluate VioLinn in a Kubernetes cluster and the simulator tool.

Since Voilà only caters for a single device type, we need to extend all of its components to allow making decisions based on the distinction between the dedicated and spare devices. The first step is to add the capability of labeling the devices as dedicated and spare devices, and then to add functions to detect the device type in order to make placement and autoscaling decisions.

Since cost calculation and optimization is not included in Voilà, and this work tries to optimize the network latency as well as deployment cost, new cost estimation functions are implemented. These provide the placement and autoscaling algorithms with an estimation for various service size and placement configurations. The cost functions calculate the cost based on the number of devices and their types, according to the cost model presented in Section 2.7. The system administrator is capable of changing these configuration parameters according to the cluster characteristics.

The availability of the two types of devices and the consideration of cost adds a new level of complexity to be addressed by the placement and autoscaling algorithms. Consequently, three new algorithms (instances of the algorithms described in Section 3.3.3) are implemented to take full advantage of the spare devices to optimize the QoS while maintaining low cost (see Section 5.4).

The code corresponding to the implementation used in this work is available in Gitlab[3].

5 EVALUATION SETUP

To evaluate how VioLinn handles the mentioned challenges, two sets of experiments are performed: the first one in a real-world experimental testbed and the second in a simulator. The first set of experiments focuses on understanding how an implementation using the mature Kubernetes container orchestrator can provide a solution for the QoS vs. cost trade-off; the second set of experiments evaluates the algorithms' scalability with greater number of edge devices.

5.1 Experimental environment

The first experiment environment (shown in Figure 5b) is a cluster of 22 Raspberry Pi 3B+ devices running HypriotOS v1.9.0, Linux kernel 4.14.34, Docker v18.04.0 and Kubernetes v.1.9.3.

The simulation experiments are conducted in a custom Python simulator[4] which replicates the behavior of the physical testbed. Using a simulator facilitates experiments with larger numbers of devices (up to 500), which was not possible to do in the testbed. The simulations run on a HP Elitebook 840 G5 laptop with Ubuntu 20.04, 16 GB RAM and an Intel Core i7-8550U CPU @ 1.80 GHz x 8, except for the scalability analysis which was run on Dell Inspiron 15 7000 laptop with Ubuntu 20.10 with 16 GB RAM and Intel Core i7-10750H CPU @ 2.60GHz × 12.

The extended Voilà stack is a complex ecosystem that supports a large range of configurations to cover multiple scenarios. Table 1 summarizes the used parameter values (both in the testbed and in the simulator) for the generic parameters appearing in all scenarios.

The configurable cost values are chosen relative to the utilization cost of a dedicated device during a cycle (c_u^D), set to 1. Using a spare device (c_u^S) is considered three times more expensive as these are scarce resources and should be profitable for providers while being costly enough so that it is not a viable option to permanently overprovision the system. Replicas are running in lightweight Kubernetes containers that are not inexpensive to create, so c_r is chosen as 0.2. Finally, c_a is the fee for including a spare device. In this work, including a device does not include reserving

[3] https://gitlab.liu.se/ida-rtslab/public-code/2022_VioLinn
[4] Available at https://gitlab.liu.se/ida-rtslab/public-code/2022_VioLinn_sim

Parameter	Notation	Value	Parameter	Notation	Value
Utilization cost of a dedicated device per cycle	c_u^D	1	Utilization cost of a spare device per cycle	c_u^S	3
Activation cost of a device	c_a	0.5	Replica cost	c_r	0.2
Peak radius		1*L0 ms	Peak strength	PS	4 (testbed) 6 (simulator)
Latency threshold	L0	20 ms	# dedicated edge devices	$\|\Delta^D\|$	15 (testbed) 70 (simulator)
Cycle length	T_d	1h	# spare edge devices	$\|\Delta^S\|$	7 (testbed) 30 (simulator)
Weight parameter between QoS and cost	α	0.5	# edge devices receiving tasks		12 (testbed) 50 (simulator)
QoS threshold	$\mathcal{E}_0$	1%			

Table 1. Evaluation parameters.

resources on it (i.e. there is no guarantee that you can use resources on it) so the cost is lower than the utilization cost in a cycle. A sensibility analysis of how varying some of these costs impact the results is provided in Section 6.5.

Regarding the other parameters, the length of a cycle (T_d) is taken as 1 hour, which allows for capturing load spikes that are lasting long enough at the scale of a day while avoiding unwanted fluctuations caused by short load spikes. A placement is considered satisfactory when 99% of the tasks meet their latency requirements, hence the choice of $\mathcal{E}_0$ as 1%. Finally, the proximity of edge devices is governed by the L0 latency threshold. Here it should not take more than 20 ms round-trip latency to another device in order for it to be suitable for executing tasks. This is in line with the fact that current latency-sensitive applications require end-to-end latency below 10 to 20 ms [2, 13].

5.2 Performance metrics

We evaluate VioLinn using several performance metrics: the QoS abstracted as the percentage of slow tasks for a given edge application placement ($\mathcal{E}^\omega$) as presented in Section 2.8, the total cost of the placement (C^ω) as presented in Section 2.7, the number of edge devices included in the selected placement (i.e. the deployment size), as well as the number of spare devices in the selected placement (i.e. the number of pods deployed on spare devices).

5.3 Service and load scenarios

Our work focuses on the orchestration algorithms and not on the service provided by the edge devices. The aim is to evaluate how spare devices can contribute to handle load spikes (in time and space) with satisfactory QoS from the edge provider perspective, independently from the performance of the services themselves. Hence, deploying a realistic application service in the testbed could actually interfere with the evaluation of the infrastructure (orchestration) services by burdening the system in a less controlled manner. To create repeatable experiments we test with a service that takes roughly the same duration on every request. Hence, the service considered for the evaluations is a straightforward one where the application task simply asks for the IP address of the chosen application replica for executing the task. Such a service is very lightweight and overload situations are instead created by a large number of such tasks sent into the orchestration system. In our illustrative Open-RAN use case, the RAN elements that are deployed at the edge server are basic operations [26], which the implemented service can be assimilated to.

The load scenarios are based on a trace of geo-distributed Internet requests in the province of Trentino in Italy [4]. Requests are gathered within 1 km^2 cells. We emulate load in the testbed and the simulator by randomly selecting as many cells from the trace as there are dedicated devices receiving tasks, and replaying the number of requests as the tasks that the system should handle. This workload presents a classic day/night pattern, with most of the requests being issued between

9am and 11pm. Moreover, different types of geographical areas (e.g., commercial districts and residential areas) have their load peak(s) at different times of the day.

Using spare devices should remain an exceptional solution for dealing with abnormal load in an area. Using spare devices in normal situations would signal that the system is under-dimensioned for this area. We use the workload trace as the "normal" load for the orchestration area. Additional local load spikes are introduced to simulate an overload limited in time and geographical location, for example created by a large concert happening. We select a geographical location determined by a peak center (i.e. a randomly-selected dedicated edge device) and a peak radius (defined in terms of latency around the peak center). During a specified time period the load of the dedicated devices located within the concerned area will be multiplied by a coefficient (called the peak strength) to reflect an area that was impacted by a sudden load increase.

5.4 Algorithm alternatives

The evaluation is performed in two steps. First, a solution without spare devices available (*WithoutS*) is created as a baseline to compare with a scenario where spare devices are made available to handle load spikes (using one of three scenarios below). This is done in order to showcase the need for extra resources in the case of localized load spikes.

Then, three alternative algorithms for the area orchestration are compared to one another. The edge orchestration part is the same for all three alternatives.

QoSOnly: The first alternative uses spare edge devices but focuses on QoS only (ignoring costs). When the choice has to be made of selecting the best placement in the edge device provisioner of the area orchestrator (using Functions 4 and 5), the one with highest QoS, i.e., the lowest $\mathcal{E}$, is selected. The cost of the different placement alternatives is ignored.

QoSMinCost: The second alternative using spare edge devices selects the placement that provides the minimum cost if $\mathcal{E}$ is below the threshold. In case of ties, the one with lowest $\mathcal{E}$ is selected.

QoSFitness: The third and last alternative selects the best placement, i.e., the one with the highest fitness (i.e., lowest bad-fit score from Section 2.9) if the QoS requirements are satisfied (i.e., $\mathcal{E}$ is below the threshold). Depending on the value of α, this alternative will favor solutions that are more (or less) cost-effective at the expense of lower (higher) QoS. In the evaluation, this alternative is considered with $\alpha = 0.5$, meaning a balanced trade-off level between QoS and cost.

To the best of our knowledge and according to a recent survey [9], there is no other work dealing with the edge device provisioning subproblem and hence no other alternative algorithms.

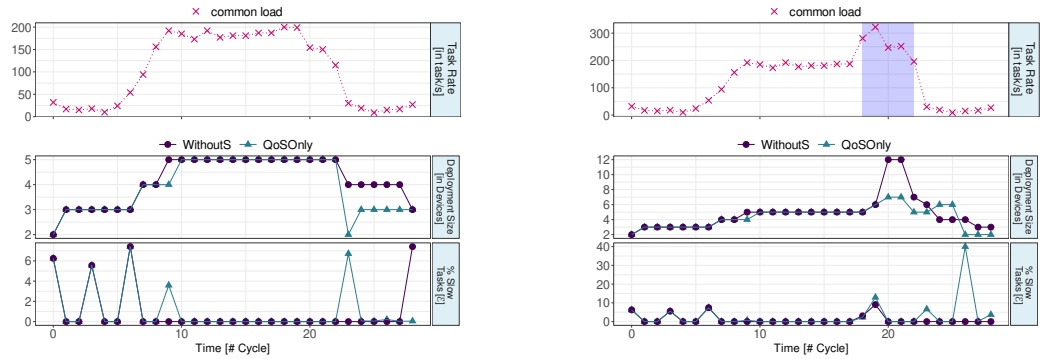
6 RESULTS AND DISCUSSIONS

In this section, the outcome of the evaluations described in the previous section is presented. Sections 6.1 and 6.2 use the physical testbed whereas Sections 6.3 to 6.6 are based on simulations.

6.1 Dealing with load changes using spare devices

We begin by showing how a typical change in load over time is treated by the WithoutS and QoSOnly alternatives respectively. Figure 6a shows an example run of a load during 28h. In this run, the testbed is loaded with a sub-trace of the Trentino workload restricted to 12 dedicated edge devices receiving tasks, with no added local spike.

The top part of the graph shows the total number of tasks coming to the area. There is a clear time aspect where, in the early morning, there is low level of activity. The number of tasks increases from 8am and then decreases around 10pm. The lower part of the graphs shows the violations, i.e., the percentage of slow tasks. Most of them appear during the early part of the day, when the system is trying first to find a good set of devices for the load and then has to adapt to the increase in number of tasks. Then, some violations appear again towards the end of the day, when the system



(a) Without a load spike

(b) With a localized load spike

Fig. 6. Handling a 28-hour workload trace (testbed).

scales down as the number of tasks decreases (the number of devices used is visible in the middle part of the figure). In the middle of the day, when the load is stable, there is no violation and the number of replicas used for the two algorithms remains stable, showing a similar performance.

In order to achieve high QoS, it is important to be able to handle not only load fluctuations in time but also in space. Therefore, the previous experiment is repeated with the introduction of a load spike in one geographical part of the system during a few hours (in this case between 6 and 10 pm). Such a load spike can correspond to a real scenario with a sport event or a demonstration for example where an unusual number of persons gather in a specific location. Figure 6b shows the system behaviour where the regular workload trace as in Figure 6a is now modified to include a load spike (with a peak strength of 4), highlighted in area shaded blue in the figure.

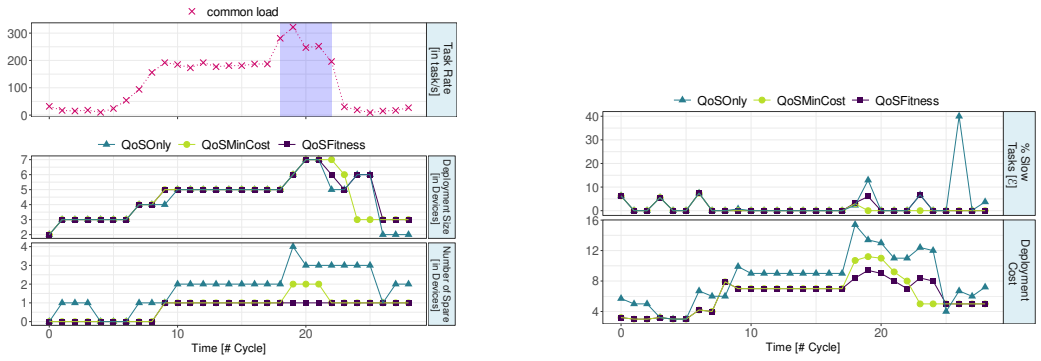
As visible in the middle graph of Figure 6b, the load spike is burdening the system heavily, since it requires to use 12 of the 15 dedicated devices in the WithoutS scenario (purple curve). When spare devices are introduced (blue curve), it is possible to handle the spike with 7 devices only (of which 4 are spare devices) for a similar performance, as the violation level is almost the same for both alternatives (see the bottom graph). This is because the algorithm focuses on getting devices with the lowest latency, no matter whether they are spare or dedicated.

This experiment shows the potential for scaling up the system using spare devices when the dedicated ones are not sufficient in the presence of localized load spikes.

6.2 Balancing quality of service and cost

In practice, it is often not feasible to add spare devices without any further thoughts as these come at a higher cost for the edge provider. As discussed in Section 2.7, the costs considered in this paper are activation, replica and utilization costs. Therefore, the next step of the study consists of executing the three alternatives for device selection described in Section 5.4 on the same load. The results of this evaluation are shown in Figure 7a and Figure 7b.

Figure 7a shows the number of devices used by the different algorithm alternatives. The overall deployment size (middle graph) is similar for the three alternatives during most of the trace execution. However, in the lower part of the figure the number of spare devices in the deployment varies between the three alternatives. As expected, the QoSOnly alternative is using the greatest number of spare devices as this algorithm focuses on getting the best devices (in our case the ones offering the lowest latency), no matter their cost. The two other cost-aware alternatives use a greater number of dedicated devices instead, which are cheaper according to the cost model.



(a) Size

(b) Performance and cost

Fig. 7. Comparison of the three algorithm alternatives when handling a 28-hour workload trace (testbed).

Details about the performance and cost results are shown in Figure 7b. Before the load spike, at time point 8 to 18 of the trace, the three algorithms have a similar performance with regards to violations. However, they do not select the same devices, which is reflected in the cost part (QoSOnly selects more spare devices than the other two so it is more expensive). During the spike, all algorithms manage to handle the violations that appeared after two cycles.

Since different devices get selected by the different algorithms, we see that: (1) QoSOnly presents a greater percentage of slow tasks at the start of the load spike, and also after the load spike despite the fact that this alternative focuses on QoS; (2) QoSMinCost is at the beginning of the load spike actually more costly than QoSFitness. This may seem counter-intuitive but it is an effect of the small scale of the testbed, where selecting a different device can have an important impact.

About the first observation, the greater fraction of slow tasks for QoSOnly at the start of the load spike is due to the chosen devices becoming overloaded. Indeed, the load increases during several cycles at the start of the spike, so a device that was selected during the first spike cycle can in fact be insufficient to handle the additional load during the second spike cycle. Regarding the high percentage of slow tasks after the load spike has ended (cycle 27), it happens as the load is very low and VioLinn therefore scale downs the number of devices handling load. During this scale down, four devices are removed (see Figure 7a) which leads to six tasks becoming slow. However, the load being very low (15), these few tasks yield a comparatively large percentage of the total load, i.e., a high $\mathcal{E}$. We can see in Figure 7a that replacing one of the dedicated devices with a spare one drastically reduces the percentage of slow tasks in the next cycle, for a comparable load.

About the second observation, towards the end of the load spike, the QoSMinCost alternative changes the set of selected devices in order to pick the least expensive. Indeed, the priority is always given to QoS in all alternatives. When a load spike starts, the focus is on securing acceptable QoS first before looking at the cost trade-off. The algorithm only trades QoS against cost when the QoS is under the threshold (see lines 5, 9 and 13 of Function 4). Moreover, the small scale of the testbed does not enable the alternatives QoSMinCost and QoSFitness to be clearly differentiated so there is a need to perform further evaluations with a larger scale.

In conclusion, QoSFitness represents a good compromise with regards to both performance and cost. We will delve further into the trade-off by looking at larger experiments next.

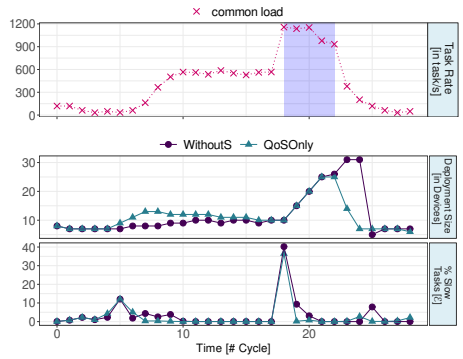


Fig. 8. Handling a 28-hour workload trace with a localized load spike (simulator).

6.3 Scaling up using simulations

We now evaluate VioLinn in the simulator described in Section 5.1, with a random subset of 100 devices taken from the Trentino workload trace. Of those, 30 are spare devices.

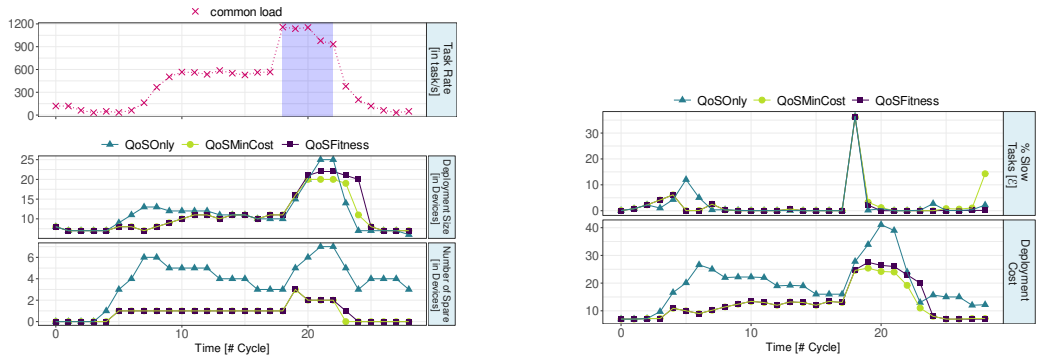
6.3.1 Single trace study. As in the testbed evaluation, a load spike is introduced in a random 28-hour workload trace, at a random location, between cycle 18 and 22. Figure 8 shows the result for a load spike of peak strength 6. The chosen peak strength is different from in the testbed experiment as the scale of the system differs. Indeed, a larger system has a greater inherent capacity for dealing with increased load so the load spike needs to be adapted according to the system for the evaluation to be relevant. Moreover, the number of dedicated devices receiving tasks is increased from the testbed experiment so that a similar share of devices receive tasks. This leads to an increase of the task rate as visible on the top part of Figure 8.

In this figure, the same behaviour as in the testbed can be observed. When the spike starts, both the WithoutS and the QoSOnly alternative start adding replicas, but the QoSOnly alternative finds a stable state (using 25 replicas) before scaling down, while WithoutS continues adding replicas (up to 31) until the end of the spike, which is a sign that the system is struggling to handle the load and cannot find a placement solution satisfying the QoS requirements. When looking at the percentage of slow tasks, it can be seen that QoSOnly is able to find a solution which decreases the percentage of slow tasks to a pre-spike level after only one cycle into the load spike, while it takes two additional cycles for the baseline to achieve a similar result.

Moving on to comparing the different algorithm alternatives when using spare devices, Figures 9a and 9b show how these are performing with regards to size, performance and cost.

With regards to size, the middle graph of Figure 9a shows that the total number of replicas is rather similar for the different alternatives but when looking at the bottom part, there is a clear difference in the number of replicas deployed on spare devices. When focusing more on QoS (alternative QoSOnly), the total number of replicas is first the greatest and then the lowest, but it always includes a higher number of replicas deployed on spare devices. When introducing the cost trade-off (alternatives QoSMinCost and QoSFitness), fewer spare devices are used (a maximum of 3 instead of a maximum of 7). This is because these alternatives only introduce a spare device when absolutely necessary for meeting the QoS requirements.

With regards to performance and cost, Figure 9b shows that the three alternatives manage to handle the violations caused by the load spike in a comparable way, with a slight advantage for the QoSOnly alternative. Since the deployment size and number of spare devices is different as



(a) Size (b) Performance and cost

Fig. 9. Comparison of the three algorithm alternatives when handling a 28-hour workload trace (simulator).

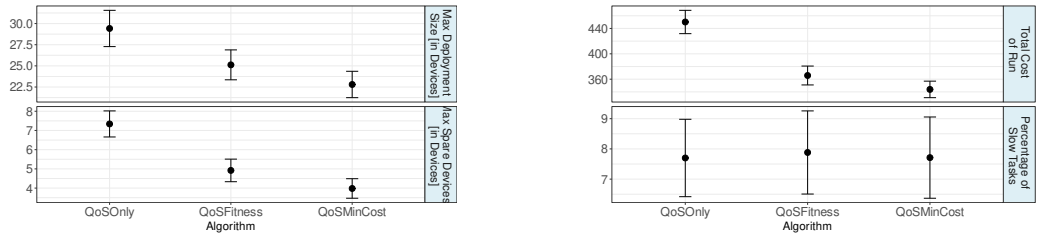


Fig. 10. Analysis of the chosen indicators for a 28-hour workload trace (simulator).

seen in Figure 9a, the cost is also different, with QoSOnly being the most expensive solution and QoSMinCost the least expensive one. This is directly related to the number of spare devices used as these are the expensive part in the cost model used in this study.

6.3.2 Multiple traces with variation in time and space. The above analysis is repeated for 50 different random loads that introduce variation in the 28h workload trace used (i.e., different days and devices included) and in the locality of the load spike. The average value for different indicators is computed and presented in the remainder of this section.

The first indicator is the maximum deployment size for a 28-hour workload trace. This corresponds to the maximum number of devices used to handle the different load spikes. Taking the maximum value and not the average enables to see to which extent the system is pushed due to the spike. As all spikes only last 4 h out of the 28 contained in the trace, using an average measure could hide the effect of the load spike by the rest of the load. Figure 10 shows at the top the average of this maximum deployment size over the 50 simulator runs. It can be seen than on average, QoSOnly uses a greater maximum number of devices than QoSFitness that on average uses more devices than QoSMinCost. However, the standard deviation indicates that specific runs can exhibit other behaviours, especially for QosMinCost and QoSFitness.

The second indicator is the maximum number of spare devices used for a 28-hour workload trace. This is interesting is order to see whether the devices deployed during the load spike are mostly spare or dedicated devices. Figure 10 shows the average number of spare devices over the simulation runs in its second chart from the top. It shows a similar pattern, with QoSOnly having the greatest maximum number of spare devices, followed by QoSFitness and finally QoSMinCost.

This is inline with how the algorithms are expected to behave, since the cost aspect is driven by the spare devices in the model used.

The third indicator is the total cost in terms of activation, replica and utilization costs for a 28-hour workload trace. It should reflect the higher cost of spare devices that are used during the load spike but even other factors. Figure 10, third chart from the top, shows the average total workload cost over the 50 simulation runs. QoSOnly clearly exhibits a greater cost, and among the two alternatives QoSFitness and QoSMinCost we find an advantage (i.e. lower cost) for QoSMinCost.

Finally, the bottom chart depicts the percentage of slow tasks in the 50 28-hour workload traces. This indicator shows how the different algorithms are performing with regards to QoS. The goal of any algorithm is to keep the total percentage of slow tasks as low as possible. The three algorithms trigger a reactive approach to fix the QoS degradation once the value of $\mathcal{E}$ has exceeded a threshold ($\mathcal{E}_0$) defined as 1%. For this indicator, the pattern is different, i.e. all three algorithms apparently have a simmilar percentage of slow tasks (around 7-8%). This is due to the reactive nature of the algorithms and will be studied in more detail in Section 6.4.

This study confirms the lower cost due to using fewer spare devices when handling a load spike. The location of the spare devices with regards to the location of the spike greatly influences performance. If a spike happens in a place where no spare devices are located, these cannot help handling the load. With regards to the three algorithms alternatives, the simulation study highlights QoSMinCost as being both cheap and performing well in terms of percentage of slow tasks.

6.4 Peak load impact on the QoS

In this section, we study the effect of the peak load on the QoS, by depicting how the algorithm reacts to the peak. We start by an experiment similar to the one in Section 6.3: i.e. select 50 traces that are each run 7 times while varying the value of Peak strength (PS) between 1 and 6.

In Figure 11 we compute the cumulative distribution function for the percentage of slow tasks:

$$CDF(Cyle_i) = \frac{\text{Cumulative \# of slow tasks until } Cycle_i}{\text{Total \# of tasks}} \tag{6}$$

For conciseness, we only include in the figure the charts for PS values of 1, 2, 4, and 6. This shows us how much each cycle of the simulation has contributed in the final percentage of slow tasks. For $PS = 1$ we see that all the algorithms were capable of achieving a low percentage of less than 1% slow tasks. The main increase of the slow tasks occurs during the 8th cycle where we expect an increase in tasks due to day-night load pattern. Similarly, for $PS = 2$, the percentage of slow tasks remains a little above 1% even though the load suddenly doubled in the peak region. The important thing to notice here is the fact that all algorithms require only one cycle to repair the QoS after an unexpected load peak, as can be seen when the CDF reaches a constant value in cycle 19 even though the peak continues for 4 cycles. Similar patterns are observed for $PS = 4$ and $PS = 6$: when the peak occurs, the system reacts quickly to provide extra replicas and eliminate the slow tasks. After one cycle in the case of $PS = 4$ and two cycles for $PS = 6$ the system reaches a stable point even with an extreme load change where the load is multiplied 6 times.

We can derive two conclusions: first, most slow tasks occur only in the first cycle, thanks to the reactive nature of the algorithms. Second, the algorithms deliver on their promises of providing an excellent quality of experience by being able to stabilize the system rapidly, within a few cycles.

6.5 Sensitivity analysis

This work uses a number of parameters that may influence the evaluation results in one way or another. We now show how changing them impacts the evaluation metrics that were used in the previous subsection. This sensitivity analysis is conducted using the custom simulator in a load

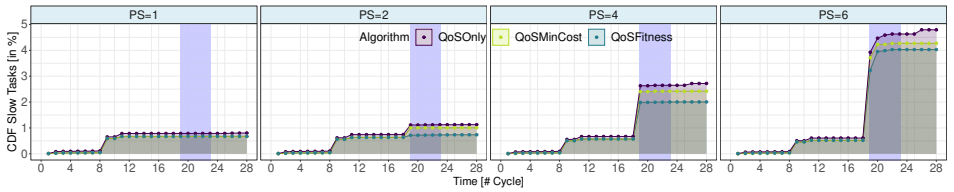


Fig. 11. The impact of different Peak Strength on the overall percentage of slow tasks.

scenario with a peak strength of 6, using 50 different runs for the QoSFitness algorithm alternative. Each run consists of a separate load (i.e. different number of tasks coming to each edge devices and different locations of the load spike). All other parameters from Table 1 are kept constant. For all the parameters except number of edge devices receiving tasks and number of spare devices, the different parameter values are evaluated on the same 50 different runs. However, changing the number of edge devices receiving tasks or spare devices impacts the load generation process so these evaluations were conducted on 50 different loads for each of the parameter values.

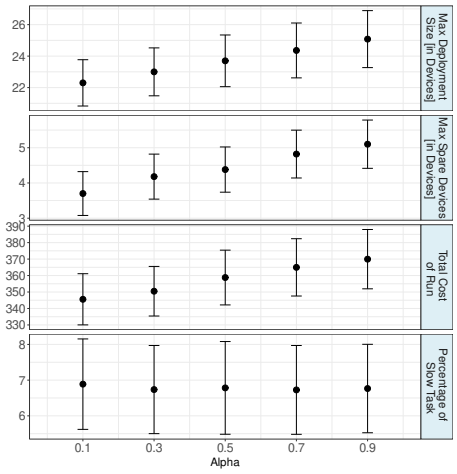
The first parameter we vary in this study is α, the weight parameter from the bad-fit score introduced in Section 2.9. This parameter varies from 0 to 1, where a high value gives more importance to performance over cost once QoS requirements are satisfied. Figure 12a shows the evaluation results with regards to average maximum size, average maximum number of spare devices used, average percentage of slow tasks and average total cost for a 28h trace. The maximum number of total and spare devices used is greater for a high value of α. This is reflected by a greater cost for greater values of α, which is expected as spare devices are more expensive. When looking at the percentage of slow tasks, there is no noticeable difference for the different values of α. This is expected as the chosen $\mathcal{E}_0$, i.e. the QoS requirement threshold, is already enforcing strict QoS requirements (with 1% of slow tasks), leaving little possibility to increase QoS when varying α.

The second parameter we vary is c_u^S, the utilization cost of a spare device. Figure 12b shows the evaluation with regards to the same indicators as before. When c_u^S increases, the total cost increases as well, as the load still requires using spare devices to maintain a good performance (as shown on the bottom chart with a stable percentage of slow tasks). It is also visible that, since the algorithm tries to find a good balance between performance and cost, the number of spare devices (i.e., the expensive part of the system) decreases when their cost increases.

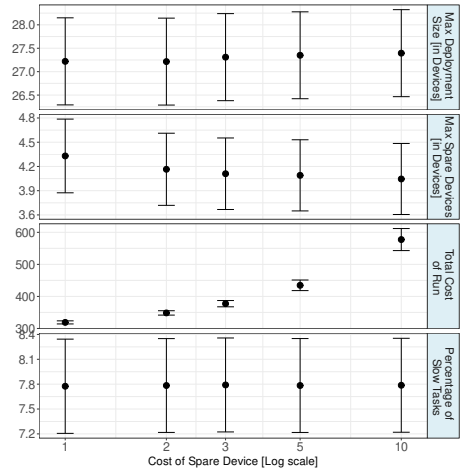
The third parameter is L0 which defines the devices that are considered nearby. We see in Figure 12c that L0 has a very visible impact on the average maximum size of the system. A high value of L0 leads to a decrease in terms of size (from a max average of 33.65 to an average of 26.72). It also impacts performance with an increase of the percentage of slow tasks. The rate of this increase however slows down for greater values of L0. Finally, a greater latency threshold leads to lower costs, as enlarging the scope of nearby devices allows using cheaper dedicated devices.

The fourth parameter is the number of edge devices receiving end-user tasks. Figure 12d shows that increasing the number of edge devices receiving tasks leads to an increased value of the maximum size, maximum number of spare devices used, total cost and percentage of slow tasks. This is reasonable as increasing the share of dedicated devices that is receiving tasks will mechanically increase the total load the system has to handle, so more resources are needed.

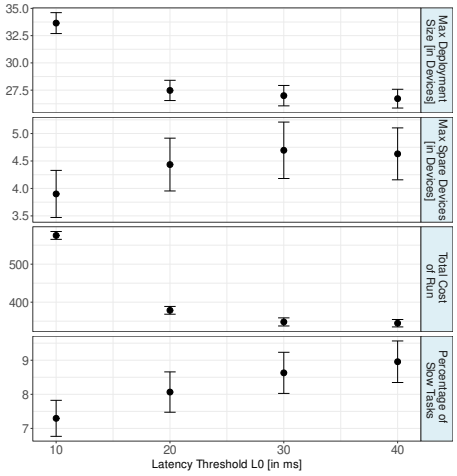
The last parameter is the number of spare devices (out of a total of 100 devices) in the orchestration area, see Figure 13. The spare devices are chosen randomly for each run among the available devices. On average, the size of the placement required for handling the load is similar regardless of the number of spare devices available (which is logical since this is related to the computing capacity



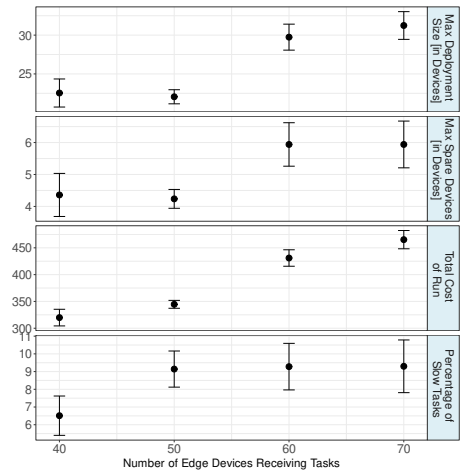
(a) Varying α



(b) Varying c_u^S.

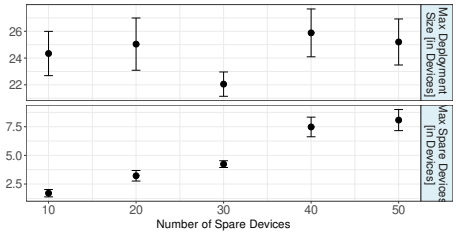


(c) Varying $L0$.



(d) Varying # edge devices receiving tasks.

Fig. 12. Sensitivity analysis.



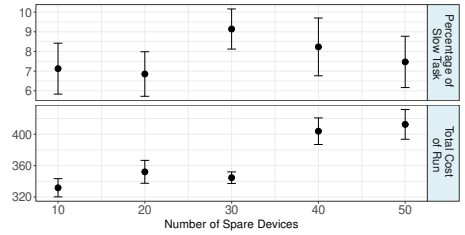
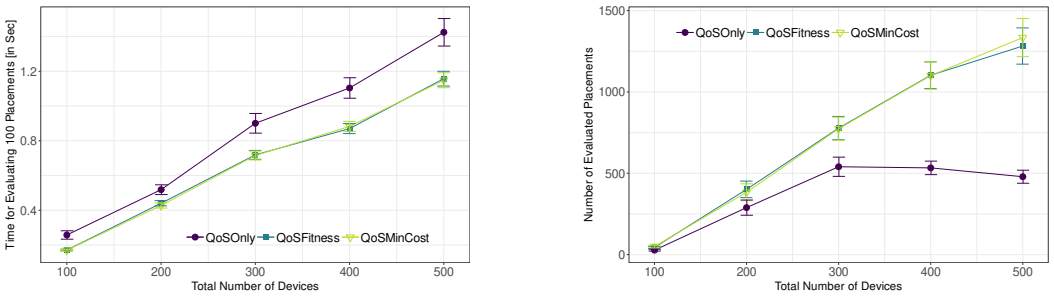


Fig. 13. Sensitivity analysis - Varying # spare devices.



(a) (b)

Fig. 14. Scalability analysis.

of the devices). However, the share of spare devices among the devices used increases when the number of spare devices increases, showing that spare devices are chosen instead of dedicated devices when they are available. The percentage of slow tasks varies between the different values of the number of spare devices with no specific pattern. Each value of the parameter is tested with a different load, which can explain the differences. Finally, the greater number of spare devices, the greater total cost, which is again expected as the spare devices are the expensive ones.

6.6 Scalability analysis

We finally study the execution time for various system sizes. We use the parameter values from Table 1, except for $c_u^S = 2$, $|\Delta^S| = 30\% \times |\Delta^D|$, and # edge devices receiving tasks= $30\% \times |\Delta^D|$. We use different system sizes $|\Delta^D| \in \{100, 200, 300, 400, 500\}$. We look at execution time for the evaluation of different placements according to QoS and cost (when appropriate). This corresponds to lines 6, 10 and 14 in Function 4 and lines 7 and 11 in Function 5.

Figure 14a displays the time needed to evaluate 100 placements as a function of the system size. As expected, the time needed to evaluate the placement increases when the system size increases. The curve's slope indicates that the placement evaluation time increase is proportional to the increase in number of devices. However, even for a system of 500 devices it is still possible to evaluate 100 placements in a little over one second. Another conclusion that can be drawn from Figure 14a is the improvement on the placement evaluation algorithms achieved in VioLinn, where QoSMinCost and QoSFitness (whose curves are on top of each other) are $\approx 20\%$ faster than QoSOnly.

Figure 14b shows how many placements need to be evaluated until a solution is found as a function of different system sizes and algorithms. For this metric, the slope of the curves indicates first a proportional increase that then flattens. This is clearly visible for QoSOnly (between sizes 300 and 500) but only a beginning of flattening is visible for QoSFitness and QoSMinCost. Since QoSOnly considers the value of $\mathcal{E}_0$ as the only metric, finding a solution requires fewer placements to be evaluated. In comparison, QoSFitness and QoSMinCost have two metrics to achieve a solution. As a result they need to evaluate more placements. All in all, both algorithms were still able to find solutions in less than 15 seconds for large system sizes of 500 devices which indicates the ability of these systems to cope with large scales. Recall that this placement evaluation is performed once per cycle when a load change requires it and ensures that services can get a service time satisfying their QoS requirements (e.g. in the ms range) during the rest of the cycle.

7 RELATED WORKS

7.1 Task and service placement

Placing tasks optimally is crucial for the successful deployment of edge computing. Since this work considers that only a subset of the edge devices have an application instance to process tasks, it is also important to consider how to place the service instances (also called application instances or replicas). This service placement problem is being studied since the creation of the first geo-distributed environments such as content delivery networks [25, 56].

In the edge/fog computing field, the survey [61] devotes a section to recent works tackling the task placement problem, while one section of [44] discusses service placement, and one of the fog computing resource management issues discussed in [20] is application placement. More surveys gather works about application placement. For example, [6] focuses on analyzing the algorithms used. There exists several recent application placement taxonomies, e.g. [34] which uses eight dimensions including the resource type, placement metric, and offloading approach. Another classification is found in [49] including control plan design and mobility support. The authors also discuss optimization strategies and evaluation environments. A recent resource scheduling survey [33] includes a resource provisioning part gathering task/service placement studies.

Many works presented in these surveys consider that all the task placement requests are received by a central entity. Such a centralized approach enables formulating the task placement problem as an optimization problem, solved using optimization techniques [35, 42, 53, 66] or heuristics [55, 68]. In this work, the task placement is distributed, i.e. each edge device places the tasks it receives.

Table 2 presents relevant recent publications on task and service placement in fog/edge computing. There are only a few works proposing both a dynamic placement (i.e. an update of the placement when the workload changes) and the placement of a replica set. Yu et al. [70] study replicated VMs placement. Their algorithm considers end user-edge proximity but not edge-edge proximity. Aral et al. [3] and Shao et al. [51] propose dynamic replica placement algorithms for data services. Similarly, Li et al. [30] present a replica placement algorithm to enhance data availability. When evaluating their algorithms, these three works use mean latency as a metric and do not consider performance issues coming from uneven load distribution. ElasticFog [38] modifies Kubernetes to consider the network traffic status of each node and accounts for it when placing replicas. However, contrary to what the name may suggest, they do not consider adding/removing replicas. Our previous works Hona [17] and Voilà [18] are the first dynamic replica placement algorithms aiming to maintain the tail latency within pre-defined bounds. Hona finds a solution based on tail latency and load balancing to ensure a fair load distribution over a fixed-size replica set. Voilà has in addition the capability to scale the replica set size according to tail latency and saturation.

To our knowledge, this work is the first to consider resource elasticity with dynamic placement of tasks and replicas. It also uses tail latency in order to ensure high QoS for most of the tasks.

7.2 Autoscaling

In an edge environment, the workload that the system has to handle will vary according to time and space. Therefore, one must scale the resource to adapt for load changes, i.e., use autoscaling. Despite an important body of work studying autoscaling in the past decades, it has not been widely studied in the context of fog/edge computing. This is probably due to the prevalent use of vertical offloading instead of autoscaling [61], i.e., offloading parts of the requests to the cloud. While this is a valid approach for many applications, it is not feasible for latency-sensitive applications.

Nevertheless, the literature includes a wide range of autoscaling techniques in similar platforms. Autoscalers designed for general-purpose Kubernetes clusters aim at providing a seamless service for the application users [39, 47, 57]. Geo-distributed autoscalers aim at selecting the nearest available

Type	Reference	Dynamicity	Replication	Elasticity	Objective	Evaluation
Data	Lera [28]	✗	✗	✗	NU	Sim
	Liu [32]	✗	✗	✗	RT	Sim
	Naas [36]	✗	✗	✗	RT	Sim
	Aral [3]	✓	✓	✗	RT	Sim
	Shao [51]	✓	✓	✗	RT	Sim
VM	Li [31]	✗	✗	✗	NU	Sim
	Zhao [71]	✗	✓	✗	NU	Sim
	Yu [70]	✓	✓	✗	NU	Sim
Service Oriented	Hong [23]	✗	✓	✗	DT	Testbed
	Silvestro [52]	✗	✗	✗	PX,DT	Sim
	Xu [69]	✗	✗	✗	PX	Sim
	Skarlat [53]	✗	✗	✗	PX,RU	Sim
	Tang [59]	✓	✗	✗	PX,DT	Testbed
	Li [30]	✓	✓	✗	PX,RU	Testbed
	Wang [66]	✓	✗	✗	RU	Sim
	Mahmud [35]	✗	✓	✗	RT,RU	Sim
	Ouyang [42]	✓	✗	✗	RT	Sim
	Souza [55]	✗	✗	✗	RT,NU	Sim
	Xia [68]	✗	✗	✗	RT,NU	Sim
	Nguyen [38]	✓	✓	✗	DT,RT	Testbed
	Hona [17]	✓	✓	✗	PX,LB	Testbed+Sim
	Voilà [18]	✓	✓	✗	PX,ST	Testbed+Sim
	VioLinn	✓	✓	✓	PX,ST	Testbed+Sim

Table 2. Literature review for task and service placement. Response Time (RT) represents the overall response latency including network and processing latency; Network Usage (NU) is the volume of backhaul traffic; Resource Utilization (RU) is the effective use of the available resources; Deployment Time (DT) is the time needed for the algorithm to find and deploy a solution; Proximity (PX) is the latency between end-user and the closest application instance; Load Balancing (LB) is the distribution of the load over the replicas; and Saturation (ST) is the performance degradation induced by overloaded replicas.

resource [46, 73]. Furthermore, countless autoscaling algorithms are available for cloud computing systems which have various objectives such as cost reduction and performance optimization [10].

Few papers propose autoscaling systems designed for fog computing platforms. Zheng et al. [72] propose to vary the number of replicas according to the load, but they do not consider pod placement nor efficient user request routing to the closest replica. ENORM [65] reduces the latency between the user and where the application is executed, as well as the network traffic to the cloud. However, the location of the computing resources is not considered and all devices are deemed equivalent to the others. Abdullah et al. [1] propose a predictive fog autoscaler for microservices. They build their prediction model using machine learning, based on observation from two applications. In our case, the load spikes are events outside of the "normal" operations, hence unsuitable for prediction.

Considering a fixed set of edge devices, the dynamic fog resource manager Voilà [18], which this work extends, considers autoscaling to adjust the number of application replicas upon any significant variation of the request workload, placement/replacement to choose where these replicas should execute, and routing of end-user request to nearby replicas.

7.3 Edge device provisioning

Edge device provisioning is about dealing with load spikes using all resources without satisfying the need. In other words, what to do when autoscaling on the available resources is not enough?

A straightforward solution would consist of increasing the size of the available edge resources. As an analogy, cellular networks are usually provisioned with enough resources to ensure an excellent

QoE even under peak traffic [12]. This however leads to a low energy efficiency and resource utilization during non-peak hours. For cellular companies like Vodafone Germany, this necessary overprovisioning has resulted in a 6% annual decrease in revenue during the period of 2000 to 2009, one of the main reasons being the energy cost [19]. In the context of a file storing application, Li et al. [29] address the resource problem by considering that it is possible to rent cloud nodes if necessary. In our case, this is not a viable alternative due to the latency-sensitive applications.

An alternative solution would be the capability of leveraging another source for spare resources, e.g. by renting nearby resources from another edge or fog infrastructure provider. Narayana et al. [37] present a cost model for renting edge resources. It includes a cost for renting a resource (similar to our utilization cost) and fetching the service (similar to our replica cost). However, their stochastic study is limited to one edge device and one cloud device and assumes that any load can be served. In some cases, like fog gaming where the users' Sega consoles are used as fog nodes [22], the platform can also rent resources from currently inactive users in order to process requests originating from nearby active users. Another alternative for finding spare resources is to use idle computing resources from (high-end) IoT devices, which are becoming increasingly powerful. For example, Kim et al. [27] propose a task scheduling scheme where edge and IoT devices collaborate to serve the tasks coming to the edge, while ensuring that local IoT tasks are not impacted. Finally, adding extra resources at a particular location can be achieved through the use of mobile edge devices (e.g. edge-equipped drones in a smart city) as advocated by ORCH [62].

7.4 Orchestration frameworks

Santoro et al. [50] propose the Foggy framework for workload association. The prototype is implemented by using OpenStack and Kubernetes. Orchestration is performed in a centralized way, which is different from our work, but the testbed implementation uses similar software tools.

Sonmez et al. [54] present a workload orchestrator using fuzzy logic. The orchestration is performed in a centralized way and considers only task placement, contrary to our work. Moreover, they only propose a simulation evaluation using the open-source EdgeCloudSim simulator.

Etemadi et al. [14] propose an ASRP (Automatic Scalable Resource Provisioning) framework that is composed of four modules: Control, Monitor, Prediction and Decision. They use neural networks for predicting the future workload. However, they only focus on time-varying workloads and not on time- and space-varying ones as in this paper. They also only provide simulation evaluations.

HYDRA [24] is a proof-of-concept of a fully distributed location-aware orchestrator. It scales up to 20 000 nodes, thus providing an alternative design with edge computing in mind to the centralized Kubernetes orchestrator. As HYDRA is a proof-of-concept, there is no real deployment of it and they do not deal with load spikes and their impact on the resource provisioning.

COSMOS [43] is an orchestration framework focusing on offloading requests about points of interest. The implementation is deployed in a square and uses network function virtualization. However, all requests are handled by a central entity that decides on where/if to handle them, whereas in our case, the requests are handled in a distributed way, at the node closest to the user.

Finally, ORCH [62] is a generic framework solving the task and mobile edge placement problems using a deadline-aware approach. While ORCH focused on mobile resources for providing the dynamic resources, in this work both stationary and mobile resources can be exploited. Also, this work includes service placement, while ORCH assumed all edge devices have the required service.

8 CONCLUSION

Orchestration of network resource utilization in presence of explicit quality of service and cost requirements is one of the challenges of edge networking systems. We presented the VioLinn system for tackling the joint problems of task placement, service placement and edge provisioning

to handle latency-sensitive tasks even during sudden and geographically-limited load spikes. Three different QoS-cost trade-off algorithms that exploit spare devices are implemented in a real testbed and compared. The implementation extends the Voilà software stack atop Kubernetes, and is extensively evaluated. The algorithms are also evaluated for scalability using a simulator. Spare devices are shown to be an effective way to handle localized load spikes. In particular, the QoSFitness and QoSMinCost algorithms allow for QoS to be achieved with a reduced cost hence showing that the proposed solution can maintain QoS while introducing different QoS-cost trade-off options, one focusing on reducing cost (QoSMinCost) and the other providing flexibility in the trade-off.

This approach relies on the edge infrastructure providing an appropriate amount of resources for the regular load. The spare device concept and the proposed algorithms were therefore not thought of as a way of handling regular load in the absence of spikes and may work orthogonally with alternative resource management regimes. Moreover, VioLinn relies on the availability of spare devices nearby (in time and space) in relation to the load spikes, which requires further work on how to make it easy to have devices capable of acting as spare devices.

A further step in this work would be to test the VioLinn system using the presented components with more specific load and deployment scenarios in order to define more precisely how it can be used in a real edge deployment. Another further step is to consider more elaborate alternatives for implementing the different components where simplified options were explored here. For example, a prediction scheme could be considered to predict the locations in the edge infrastructure that may be sensitive to a load spike and study how this may improve the overall system performance. An option is to explore leveraging data-driven machine learning techniques.

ACKNOWLEDGMENTS

The work at Linköping University was supported by CUGS national graduate school and ELLIIT strategic research area, and the IRISA collaboration by a mobility grant from Rennes Métropole.

REFERENCES

[1] Muhammad Abdullah, Waheed Iqbal, Arif Mahmood, Faisal Bukhari, and Abdelkarim Erradi. 2021. Predictive Autoscaling of Microservices Hosted in Fog Microdata Center. *IEEE Systems Journal* 15, 1 (2021), 1275–1286.

[2] Arif Ahmed, HamidReza Arkian, et al. 2019. Fog Computing Applications: Taxonomy and Requirements. *CoRR* abs/1907.11621 (2019). arXiv:arXiv:1907.11621

[3] Atakan Aral and Tolga Ovatman. 2018. A decentralized replica placement algorithm for edge computing. *IEEE transactions on network and service management* 15, 2 (2018), 516–529.

[4] Gianni Barlacchi, Marco De Nadai, et al. 2015. A multi-source dataset of urban life in the city of Milan and the Province of Trentino. *Scientific Data* 2, 150055 (2015).

[5] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (Helsinki, Finland). 13–16.

[6] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. 2020. How to place your apps in the fog: State of the art and open challenges. *Software: Practice and Experience* 50, 5 (2020), 719–740.

[7] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. 2017. Mobile Augmented Reality Survey: From Where We Are to Where We Go. *IEEE Access* 5 (April 2017), 6917–6950.

[8] Zhuo Chen, Wenlu Hu, et al. 2017. An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance. In *Proc. ACM/IEEE SEC* (San Jose, California).

[9] Breno Costa, Joao Bachiega, Leonardo Rebouças de Carvalho, and Aleteia P. F. Araujo. 2022. Orchestration in Fog Computing: A Comprehensive Survey. *ACM Comput. Surv.* 55, 2 (jan 2022). https://doi.org/10.1145/3486221

[10] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. 2015. Elasticity in cloud computing: a survey. *Ann. Telecommun.* 70, 7 (2015), 289–309.

[11] Frank Dabek, Russ Cox, et al. 2004. Vivaldi: A Decentralized Network Coordinate System. In *Proc. ACM SIGCOMM*.

[12] Ali El Amine. 2019. *Radio resource allocation in 5G cellular networks powered by the smart grid and renewable energies.* Ph.D. Dissertation. Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire.

[13] Mohammed S. Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. 2018. Toward Low-Latency and Ultra-Reliable Virtual Reality. *IEEE Network* 32, 2 (2018), 78–84.

[14] Masoumeh Etemadi, Mostafa Ghobaei-Arani, and Ali Shahidinejad. 2020. A learning-based resource provisioning approach in the fog computing environment. *Journal of Experimental & Theoretical Artificial Intelligence* (2020), 1–24.
[15] Ali Fahs. 2020. *Proximity-Aware Replicas Management in Geo-Distributed Fog Computing Platforms.* Ph.D. Dissertation. Université de Rennes 1.
[16] Ali J. Fahs and Guillaume Pierre. 2019. Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms. In *Proc. IEEE/ACM CCGrid.*
[17] Ali J. Fahs and Guillaume Pierre. 2020. Tail-Latency-Aware Fog Application Replica Placement. In *Proc. ICSOC.*
[18] Ali J. Fahs, Guillaume Pierre, and Erik Elmroth. 2020. Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler. In *Proc. MASCOTS.*
[19] Albrecht Fehske, Gerhard Fettweis, Jens Malmodin, and Gergely Biczok. 2011. The global footprint of mobile communications: The ecological and economic perspective. *IEEE communications magazine* 49, 8 (2011), 55–62.
[20] Mostafa Ghobaei-Arani, Alireza Souri, and Ali A. Rahmanian. 2020. Resource Management Approaches in Fog Computing: a Comprehensive Review. *Journal of Grid Computing* 18 (2020), 1–42.
[21] HashiCorp. [n.d.]. *Serf: Decentralized cluster membership, failure detection,and orchestration.* https://www.serf.io/
[22] Kris Holt. 2020. *Sega wants to turn Japanese arcades into 'fog gaming' data centers.* Retrieved April 27, 2021 from https://engt.co/3cZfwaw
[23] Hua-Jun Hong, Pei-Hsuan Tsai, and Cheng-Hsin Hsu. 2016. Dynamic module deployment in a fog computing platform. In *18th Asia-Pacific Network Operations and Management Symposium (APNOMS).* IEEE, 1–6.
[24] Lara Lorna Jimenez and Olov Schelen. 2020. HYDRA: Decentralized Location-aware Orchestration of Containerized Applications. *IEEE Transactions on Cloud Computing* (2020), 1–1.
[25] Magnus Karlsson, Christos Karamanolis, and Mallik Mahalingam. 2002. *A framework for evaluating replica placement algorithms.* Technical Report. Research Report HPL-2002-219, HP Laboratories, Palo Alto, CA.
[26] Ferath Kherif, Robert Gdowski, Sebastian Geller, Ciro Formisano, and Adeliya Latypova. 2020. *Use cases definition and preparation.* EU Horizon 2020 project deliverable D6.3. MORPHEMIC. Grant 871643.
[27] Youngjin Kim, Chiwon Song, Hyuck Han, Hyungsoo Jung, and Sooyong Kang. 2020. Collaborative Task Scheduling for IoT-Assisted Edge Computing. *IEEE Access* 8 (2020), 216593–216606.
[28] Isaac Lera, Carlos Guerrero, and Carlos Juiz. 2018. Comparing centrality indices for network usage optimization of data placement policies in fog devices. In *Proc. FMEC.*
[29] Chunlin Li, Jingpan Bai, Yi Chen, and Youlong Luo. 2020. Resource and replica management strategy for optimizing financial cost and user experience in edge cloud computing system. *Information Sciences* 516 (2020), 33–55.
[30] Chunlin Li, YaPing Wang, Hengliang Tang, Yujiao Zhang, Yan Xin, and Youlong Luo. 2019. Flexible replica placement for enhancing the availability in edge computing environment. *Computer Communications* 146 (2019), 1–14.
[31] Kangkang Li and Jarek Nabrzyski. 2017. Traffic-aware virtual machine placement in cloudlet mesh with adaptive bandwidth. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE, 49–56.
[32] Juan Liu, Bo Bai, Jun Zhang, and Khaled B Letaief. 2017. Cache placement in Fog-RANs: From centralized to distributed algorithms. *IEEE Transactions on Wireless Communications* 16, 11 (2017), 7039–7051.
[33] Quyuan Luo, Shihong Hu, Changle Li, Guanghui Li, and Weisong Shi. 2021. Resource Scheduling in Edge Computing: A Survey. *IEEE Communications Surveys Tutorials* 23, 4 (2021), 2131–2165.
[34] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2020. Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions. *ACM Comput. Surv.* 53, 4, Article 88 (July 2020).
[35] Redowan Mahmud, Satish Narayana Srirama, et al. 2019. Quality of Experience (QoE)-aware placement of applications in Fog computing environments. *J. Parallel and Distrib. Comput.* (October 2019).
[36] Mohammed Islam Naas, Philippe Raipin Parvedy, Jalil Boukhobza, and Laurent Lemarchand. 2017. iFogStor: an IoT data placement strategy for fog infrastructure. In *Proc. ICFEC.*
[37] V.S. Ch Lakshmi Narayana, Sharayu Moharir, and Nikhil Karamchandani. 2021. On Renting Edge Resources for Service Hosting. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 6, 2, Article 8 (oct 2021), 30 pages.
[38] Nguyen Dinh Nguyen, Linh-An Phan, Dae-Heon Park, Sehan Kim, and Taehong Kim. 2020. ElasticFog: Elastic Resource Provisioning in Container-Based Fog Computing. *IEEE Access* 8 (2020), 183879–183890.
[39] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. 2020. Horizontal pod autoscaling in Kubernetes for elastic container orchestration. *Sensors* 20, 16 (2020), 4621.
[40] Shadi A. Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. 2020. The Emerging Landscape of Edge Computing. *GetMobile: Mobile Comp. and Comm.* 23, 4 (May 2020), 11–20.
[41] Afif Osseiran, Jose F. Monserrat, and Patrick Marsch. 2016. *5G Mobile and Wireless Communications Technology.* Cambridge University Press.
[42] Tao Ouyang, Zhi Zhou, and Xu Chen. 2018. Follow Me at the Edge: Mobility-Aware Dynamic Service Placement for Mobile Edge Computing. *IEEE Journal on Selected Areas in Communications* 36, 10 (2018), 2333–2345.

[43] George Papathanail, Ioakeim Fotoglou, et al. 2020. COSMOS: An Orchestration Framework for Smart Computation Offloading in Edge Clouds. In *Proc. IEEE/IFIP NOMS*.

[44] Kai Peng, Victor C. M. Leung, et al. 2018. A Survey on Mobile Edge Computing: Focusing on Service Adoption and Provision. *Wireless Communications and Mobile Computing* 2018 (2018).

[45] CLAudit project. 2017. *Planetary-scale cloud latency auditing platform.* http://claudit.feld.cvut.cz/

[46] Chenhao Qu. 2016. *Auto-scaling and deployment of web applications in distributed computing clouds.* Ph.D. Dissertation.

[47] Gourav Rattihalli, Madhusudhan Govindaraju, et al. 2019. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *Proc. IEEE CLOUD*. IEEE.

[48] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. 2020. Geo-distributed efficient deployment of containers with Kubernetes. *Computer Communications* 159 (2020), 161–174.

[49] Farah Aït Salaht, Frédéric Desprez, and Adrien Lebre. 2020. An Overview of Service Placement Problem in Fog and Edge Computing. *ACM Comput. Surv.* 53, 3, Article 65 (June 2020).

[50] Daniele Santoro, Daniel Zozin, et al. 2017. Foggy: A Platform for Workload Orchestration in a Fog Computing Environment. In *Proc. CloudCom*.

[51] Yanling Shao, Chunlin Li, and Hengliang Tang. 2019. A data replica placement strategy for IoT workflows in collaborative edge and cloud environments. *Computer Networks* 148 (2019), 46–59.

[52] Alessio Silvestro, Nitinder Mohan, Jussi Kangasharju, Fabian Schneider, and Xiaoming Fu. 2018. Mute: Multi-tier edge networks. In *Proceedings of the 5th Workshop on CrossCloud Infrastructures & Platforms*. 1–6.

[53] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. 2017. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications* 11, 4 (2017), 427–443.

[54] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. 2019. Fuzzy Workload Orchestration for Edge Computing. *IEEE Transactions on Network and Service Management* 16, 2 (2019), 769–782.

[55] Vitor B. Souza, Xavier Masip-Bruin, et al. 2018. Towards a proper service placement in combined Fog-to-Cloud (F2C) architectures. *Future Generation Computer Systems* 87 (October 2018).

[56] Michal Szymaniak, Guillaume Pierre, and Maarten Van Steen. 2006. Latency-driven replica placement. *IPSJ Digital Courier* 2 (2006), 561–572.

[57] Salman Taherizadeh and Marko Grobelnik. 2020. Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering Software* 140 (2020), 102734.

[58] Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. 2020. Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation. In *Prof. MASCOTS*.

[59] Hengliang Tang, Chunlin Li, et al. 2019. Dynamic resource allocation strategy for latency-critical and computation-intensive applications in cloud–edge environment. *Computer Communications* 134 (2019).

[60] The Kubernetes Authors. 2019. Kubernetes. https://kubernetes.io/.

[61] Klervie Toczé and Simin Nadjm-Tehrani. 2018. A Taxonomy for Management and Optimization of Multiple Resources in Edge Computing. *Wireless Communications and Mobile Computing* 2018 (2018).

[62] Klervie Toczé and Simin Nadjm-Tehrani. 2019. ORCH: Distributed Orchestration Framework using Mobile Edge Devices. In *Proceedings of the 2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*. 1–10.

[63] Rob van der Meulen. 2018. What Edge Computing Means for Infrastructure and Operations Leaders. Smarter with Gartner. https://gtnr.it/3euQbFh.

[64] Verizon. 2020. *What is the Latency of 5G?* Retrieved June 18, 2021 from https://vz.to/2EdT5Sa

[65] Nan Wang, Blesson Varghese, Michail Matthaiou, and Dimitrios S. Nikolopoulos. 2020. ENORM: A Framework For Edge NOde Resource Management. *IEEE Transactions on Services Computing* 13, 6 (2020), 1086–1099.

[66] Shiqiang Wang, Rahul Urgaonkar, et al. 2017. Dynamic Service Placement for Mobile Micro-Clouds with Predicted Future Costs. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2017).

[67] IS Wireless. 2022. *IS-Wireless5GMadeTogether.* Retrieved March 30, 2022 from https://www.is-wireless.com/

[68] Ye Xia, Xavier Etchevers, et al. 2018. Combining Heuristics to Optimize and Scale the Placement of IoT Applications in the Fog. In *Proc. IEEE/ACM UCC*.

[69] Jinlai Xu, Balaji Palanisamy, Heiko Ludwig, and Qingyang Wang. 2017. Zenith: Utility-aware resource allocation for edge computing. In *IEEE international conference on edge computing (EDGE)*. IEEE, 47–54.

[70] Ya-Ju Yu, Te-Chuan Chiu, Ai-Chun Pang, Ming-Fan Chen, and Jiajia Liu. 2017. Virtual machine placement for backhaul traffic minimization in fog radio access networks. In *IEEE International Conference on Communications (ICC)*. 1–7.

[71] Lei Zhao, Jiajia Liu, Yongpeng Shi, Wen Sun, and Hongzhi Guo. 2017. Optimal placement of virtual machines in mobile edge computing. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 1–6.

[72] Wei-Sheng Zheng and Li-Hsing Yen. 2018. Auto-scaling in Kubernetes-based fog computing platform. In *International Computer Symposium*. 338–345.

[73] Jieming Zhu, Zibin Zheng, Yangfan Zhou, and Michael R Lyu. 2013. Scaling service-oriented applications into geo-distributed clouds. In *IEEE Seventh International Symposium on Service-Oriented System Engineering*. 335–340.