Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

Mobility-Aware Registry Migration for Containerized Applications on Edge Computing Infrastructures

Daniel Chaves Temp[a,b], Paulo Silas Severo de Souza[c], Arthur Francisco Lorenzon[b], Marcelo Caggiani Luizelli[b], Fábio Diniz Rossi[a,b,*]

[a] Federal Institute Farroupilha, Alegrete-RS, Brazil
[b] Federal University of Pampa, Alegrete-RS, Brazil
[c] Pontifical Catholic University of Rio Grande do Sul, Porto Alegre-RS, Brazil

ARTICLE INFO

Keywords:
Edge computing
Containers
Migration
Registry

ABSTRACT

Small footprints and fast provisioning times have promoted container adoption for deploying and managing applications on edge computing environments. As keeping all container images locally would quickly saturate the storage of resource-constrained edge servers, application provisioning consists of pulling container images from external repositories, called registries, located in specific locations in the infrastructure. Existing research reduces deployment time on edge infrastructures by defining the location of container registries. Although such an approach yields positive results in specific scenarios, it overlooks that the demand for container images in certain regions can vary over time according to users' mobility. This work presents a novel strategy that provisions container registries dynamically based on users' mobility, spinning up new registries when application provisioning times start growing excessively and deprovisioning registries far away from users. Experimental results demonstrate that our approach reduces the application provisioning time issues by 33.19% on average compared to strategies that allocate container registries statically.

1. Introduction

The popularization of mobile and real-time applications such as Augmented Reality and Unmanned Aerial Vehicle (UAV) support systems challenged the dominance of Cloud Computing as the standard architecture for hosting applications, as not even modern networking infrastructures could alleviate the noticeable end-to-end delay rendered by the physical distance between data centers and end devices (Shi and Dustdar, 2016). Such limitations paved the way for a new paradigm, called Edge Computing (Satyanarayanan et al., 2009), that brings data processing to the network's edge, in proximity to data sources, to couple the real-time requirements of modern applications.

Among the features that edge computing inherits from the cloud is virtualization, which enables multitenancy and improves the resource management process. Once edge resources are virtualized, applications can be relocated on the fly across the edge infrastructure to deliver low latency for end-users while they move across the environment.

While virtualization is transparent to end-users, under the hood, it generally takes place in two possible ways: Virtual Machines (VMs) and containers. On the one hand, VM-based applications sit on top of their own operating system, increasing the environment's isolation at the cost of considerable overhead. On the other hand, container-based applications share libraries and binaries from the host's operating system, reducing isolation compared to VMs but yielding greater agility and lower resource usage (Xavier et al., 2013).

VMs and containers are assembled based on images, which are templates containing binaries and dependencies used by the applications. While VM images are usually single-layer, container images are based on layered file systems, where the top layer hosts user-writable content and the other layers ship the container dependencies. As underlying layers are read-only, containers running on the same host can share these layers' images, reducing resource usage and shortening the time needed to provision containers on hosts with cached images.

Given the significant differences between VMs and containers, applications are migrated differently depending on which type of virtualization they rely upon. VM-based applications are relocated through cold and live migration techniques, which typically transfer the application's data from its origin to its destination host. Conversely, container-based applications are usually spawned in the target host while its underlying container images are pulled from nearby container image repositories called container registries.

* Corresponding author at: Federal Institute Farroupilha, Alegrete-RS, Brazil.
E-mail address: fabio.rossi@iffarroupilha.edu.br (F.D. Rossi).

https://doi.org/10.1016/j.jnca.2023.103676
Received 29 March 2022; Received in revised form 12 March 2023; Accepted 31 May 2023
Available online 9 June 2023

Despite the increased isolation granted by VMs, containers have been emerging as the prime option for deploying applications at the edge (Pallewatta et al., 2019). In addition to presenting a smaller footprint than VMs, containers reduce provisioning times from minutes to seconds compared to VMs, which meets the flexibility and disposability requirements of modern software architectures (Gannon et al., 2017).

In such a scenario, whenever an application needs to be reprovisioned, a container is spawned in the target location, and the application's binaries and dependencies are pulled from centralized repositories called container registries. Accordingly, defining registries locations in the infrastructure represents a pivotal decision to ensure that applications are timely provisioned (Knob et al., 2021).

There has been considerable prior work in reducing the provisioning time of containers in edge infrastructures by managing container registries. Overall, existing solutions concentrate either on (i) optimizing internal operations on container registries (Anwar et al., 2018; Harter et al., 2016; Chen et al., 2022), (ii) employing peer-to-peer protocols to alleviate the network demand between container registries and edge servers (Nathan et al., 2017; Becker et al., 2021; Ahmed and Pierre, 2019), or (iii) defining optimized placement policies for container registries within the infrastructure (Knob et al., 2021). Despite their contributions, state-of-the-art approaches fall short in addressing the dynamic needs of edge environments, adopting static placement schemes for registries in the infrastructure, which neglect that the demand for container images in certain regions can change over time depending on user mobility.

This paper presents a novel strategy that provisions container registries dynamically in the edge infrastructure based on users' mobility based on a threshold-based approach. Whenever our approach detects that provisioning times are growing excessively, it spins up new registries nearby the users. Conversely, idle registries are deprovisioned to avoid resource wastage. Simulations demonstrate that our approach can reduce provisioning time issues by 33.19% compared to existing approaches that allocate registries statically.

The remaining of this paper is organized as follows. Section 2 details the related work. Sections 3 and 4 describe the approached scenario and our proposed strategy. Section 5 present the evaluation used to validate our approach against existing strategies. Finally, Section 6 concludes the paper.

2. Related work

Container registries play a vital role in the container ecosystem as they host and serve container images across the network. At the same time, inefficient functioning of registries can dramatically increase application provisioning times or even cause service disruption if they stop working. Based on that, many efforts have proposed optimizations in how registries distribute container images to hosts.

Anwar et al. (2018) presented an extensive analysis of Docker registry workloads from IBM cloud data centers. The authors also introduced a Docker trace replayer tool that allows extracting insights from realistic workloads, understanding, for example, how they affect the quality of service of applications in large-scale infrastructures. Based on their analysis, they proposed two registry design improvements. The first improvement explores a multi-layer caching scheme that reduces cache misses by keeping popular container layers in memory or on SSD disks. The second improvement leverages a prefetching algorithm to proactively provision popular container layers before they are requested by applications to reduce provisioning times.

Harter et al. (2016) also used empirical experiments to find insights for resource management decisions. First, the authors analyzed the provisioning time of 57 containerized applications to identify the most impactful factors. Their analysis showed that pulling software packages accounts for 76% of application provisioning times, while only 6.4% of downloaded data is required for containers to start. Based on such findings, the authors introduced Slacker, a novel Docker storage driver that leverages a lazy load approach that allows hosts to fetch images with the minimum data necessary for applications to start working while the rest of the data is downloaded later.

Chen et al. (2022) proposed Starlight, a solution for accelerating the provisioning time of containerized applications through design improvements on worker nodes and container registries. Starlight's motivation is mainly related to the inefficiencies presented by the layer-based structure used by most container solutions. For example, layer dependency causes updates to layers low in the layer stack of your images to force updates to other layers even though their content is mostly identical. Starlight addresses existing layer-related limitations by reconstructing the whole provisioning pipeline through optimizations all along the way, from compute workers to container registries. Provided optimizations include a push-based approach where workers can specify what files they already have to avoid unnecessary network transfers and a set of tools that allow containers to leverage lazy starts while their data is transferred in the background.

Nathan et al. (2017) discussed the challenges of provisioning containerized applications on resource-constrained infrastructures. Whereas resource bounds limit the number of co-hosted applications that can take advantage of image sharing on a given server, pulling images from registries can quickly produce network bottlenecks. Based on that, the authors presented CoMICon, a solution that allows cooperative management of Docker images across a pool of servers. In summary, all servers host a container registry and are eligible to serve container images with their neighbors. A similar approach was followed by other works like Becker et al. (2021) and Ahmed and Pierre (2019), which leverage peer-to-peer network protocols and shared file systems to allow neighboring servers to exchange container images.

Knob et al. (2021) highlighted the significant burden of meeting provisioning time expectations of containerized applications on edge infrastructures, where containerization challenges are summed to high server dispersion and network performance constraints. In such a scenario, inadequate positioning of container registries in the infrastructure can significantly increase application provisioning times due to network saturations. To overcome this challenge, the authors presented a placement strategy leveraging a fluid community algorithm to define where container registries should be placed to avoid excessive network competition in specific links and prolonged application provisioning times.

2.1. Discussion

Most existing research efforts have focused on improving internal operations in container registries or leveraging peer-to-peer protocols to reduce the time needed to transfer container images to hosts. Despite the technical particularities among proposed solutions, they assume that container registries are close enough to the hosts asking for container images. If such an assumption does not hold, the long network distance between registries and target hosts could partially nullify their optimization gains.

Our work complements existing approaches through a novel resource management policy that dynamically spins up new registries when application provisioning times start growing excessively and deprovisions underutilized registries. The closest work to ours is Knob et al. (2021), which presents a registry placement policy. Nevertheless, they assume that the optimal number of registries is known and that such value stays the same over time, which does not necessarily hold in edge infrastructures serving mobile users.

Table 1 presents a high-level comparison of the related work and our approach. To the best of our knowledge, we are the first to reduce the provisioning time of containerized applications on edge infrastructures by migrating container registries dynamically according to user mobility.

Table 1
Comparison between this work and related approaches.

Work	Approach	Mobility awareness	Paradigm
Anwar et al. (2018)	Registry design optimizations	×	Cloud Computing
Harter et al. (2016)	Registry design optimizations	×	Cloud Computing
Chen et al. (2022)	Registry design optimizations	×	Edge Computing
Nathan et al. (2017)	Peer-to-peer image distribution	×	Cloud Computing
Becker et al. (2021)	Peer-to-peer image distribution	×	Edge Computing
Ahmed and Pierre (2019)	Peer-to-peer image distribution	×	Edge Computing
Knob et al. (2021)	Registry placement	×	Edge Computing
This work	**Registry migration**	✓	**Edge Computing**

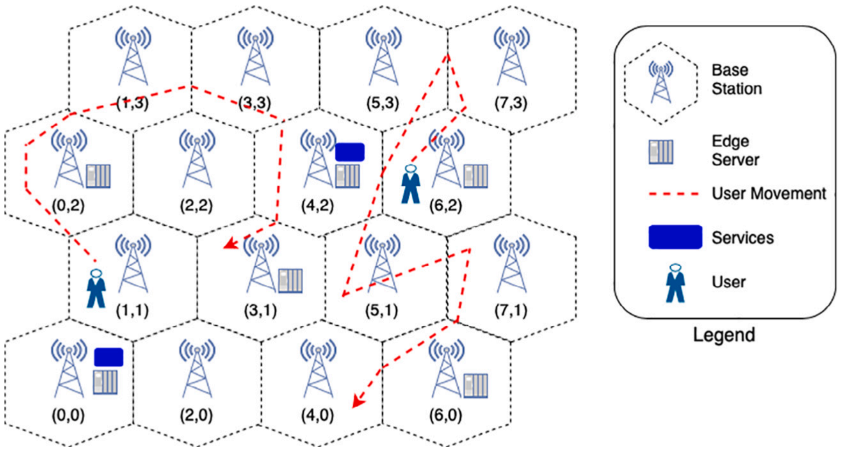


Fig. 1. Sample edge computing scenario.

3. System model

This section presents the edge computing scenario approached in this work. First, we describe the environment, including the edge infrastructure. Then, we depict the allocation decisions, including the relocation of applications and registries. Table 2 summarizes the notations.

The environment is represented as a set of hexagonal cells that divide the map, as in the model by Aral et al. (2021). The network infrastructure comprises a set of base stations $\mathcal{B}$ interconnected by a set of links $\mathcal{L}$. In this setting, a base station is represented as $\mathcal{B}_o = (w_o)$, where attribute w_o represents $\mathcal{B}_o$'s wireless delay, and a network link is represented as $\mathcal{L}_v = (d_v, b_v)$, where attributes d_v and b_v denote $\mathcal{L}_v$'s delay and bandwidth capacity.

We assume that each hexagonal cell has a base station and, optionally, an edge server. In our scenario, the infrastructure serves a set of users $\mathcal{U}$, which access a set of applications $\mathcal{A}$. While base stations provide wireless connectivity to the users located in their cells, edge servers host the applications. Fig. 1 illustrates such a scenario.

An application is represented as $\mathcal{A}_j = (\mu_j, \sigma_j, \beta_j, \gamma_j)$. We assume a scenario where applications are containerized. As such, $\mathcal{A}_j$ is built on top of a container image composed of one or multiple container layers $\mu_j \in \mathcal{I}$, where each of these layers is represented as $\mathcal{I}_k = (q_k)$, with attribute q_k representing the layer size. The overall capacity demand of $\mathcal{A}_j$, denoted as σ_j, is the aggregated size of all of its layers, i.e., $\sigma_j = \sum_{k \in \mu_j} q_k$.

The delay of an application $\mathcal{A}_j$ at a time step $\mathcal{T}_t \in \mathcal{T}$ is given by $\eta(\mathcal{T}_t, \mathcal{A}_j)$, as in Eq. (1), considering the wireless delay of $\mathcal{B}_o$ (which represents the base station used by $\mathcal{A}_j$'s user) and the aggregated delay of a set of network links $\xi(\mathcal{T}_t, \mathcal{A}_j)$, used to route $\mathcal{A}_j$'s data from its base station to its user's base station. In our modeling, $\xi(\mathcal{T}_t, \mathcal{A}_j)$ is found through the Dijkstra shortest path algorithm (Dijkstra et al., 1959) (link delays as weight). In this context, $\mathcal{A}_j$'s delay SLA is violated whenever its delay $\eta(\mathcal{T}_t, \mathcal{A}_j)$ exceeds its delay SLA, given by β_j.

$$\eta(\mathcal{T}_t, \mathcal{A}_j) = w_o + \sum_{v \in \xi(\mathcal{T}_t, \mathcal{A}_j)} d_v \tag{1}$$

Table 2
List of notations used in this paper.

Symbol	Description
w_o	Wireless delay of base station $\mathcal{B}_o$
d_v	Delay of link $\mathcal{L}_v$
b_v	Bandwidth capacity of link $\mathcal{L}_v$
c_i	Capacity of edge server $\mathcal{S}_i$
μ_j	List of layers that compose the image of application $\mathcal{A}_j$
σ_j	Capacity demand of application $\mathcal{A}_j$
β_j	Delay SLA of application $\mathcal{A}_j$
γ_j	Provisioning time SLA of application $\mathcal{A}_j$
q_k	Size of container layer $\mathcal{I}_k$
g_u	List of container layers hosted on registry $\mathcal{R}_u$
h_u	Demand of registry $\mathcal{R}_u$
$x_{e,i,j}$	Applications' placement
$z_{e,i,u}$	Registries' placement
$\eta(\mathcal{T}_t, \mathcal{A}_j)$	Delay of application $\mathcal{A}_j$ at time step $\mathcal{T}_t$
$\xi(\mathcal{T}_t, \mathcal{A}_j)$	Links used to route the data of application $\mathcal{A}_j$ at time step $\mathcal{T}_t$
$\rho(\mathcal{T}_t, \mathcal{S}_i)$	Demand of edge server $\mathcal{S}_i$ at time step $\mathcal{T}_t$

In addition to hosting applications, edge servers accommodate a set of container registries $\mathcal{R}$, which hold the container layers that compose the container images used by applications. A container registry is represented by $\mathcal{R}_u = (g_u, h_u)$, where attributes g_u and h_u represent $\mathcal{R}_u$'s hosted layers and $\mathcal{R}_u$'s demand (sum of the size of all layers hosted by $\mathcal{R}_u$). Whenever an application $\mathcal{A}_j$ needs to be provisioned or relocated, the layers that compose its image must be fetched from a container registry available in the infrastructure. This application provisioning is detailed in Algorithm 1.

Previous studies show that downloading layers in parallel can increase the provisioning time as a layer is only decompressed and extracted after the previous layer has been downloaded (Ahmed and Pierre, 2018). So, in our modeling, layers are downloaded sequentially (Algorithm 1, lines 2–7). For each layer that composes $\mathcal{A}_j$'s container image, we iterate over the list of registries, looking for the closest registry that hosts that layer. Once the closest registry hosting the layers is found, it starts being downloaded. This process is repeated until the

Algorithm 1: Application provisioning process

```
Function provisionApp($\mathcal{A}_j$, $\mathcal{S}_i$):
    foreach $\mathcal{I}_k \in \mu_j$ do
        $\mathcal{R}'$ = Registries sorted by distance (asc.) to $\mathcal{S}_i$
        foreach $\mathcal{R}'_u \in \mathcal{R}'$ do
            if $\mathcal{R}'_u$ hosts layer $\mathcal{I}_k$ then
                Download layer $\mathcal{I}_k$ from $\mathcal{R}'_u$ to $\mathcal{S}_i$
                break
    return Time taken to provision $\mathcal{A}_j$
```

entire container image is downloaded. The provisioning time SLA of $\mathcal{A}_j$ is violated whenever the result of Algorithm 1 exceeds its threshold γ_j.

An edge server is represented as $\mathcal{S}_i = (c_i)$, where c_i denotes $\mathcal{S}_i$'s capacity. The application placement is represented by $x_{e,i,j}$, which receives 1 if edge server $\mathcal{S}_i$ hosts application $\mathcal{A}_j$ at time step $\mathcal{T}_t$, and 0 otherwise. Similarly, the registry placement is given by $z_{e,i,u}$, which receives 1 if edge server $\mathcal{S}_i$ hosts registry $\mathcal{R}_u$ at time step $\mathcal{T}_t$, and 0 otherwise. The overall demand of edge server $\mathcal{S}_i$ at time step $\mathcal{T}_t$ is given by $\rho(\mathcal{T}_t, \mathcal{S}_i)$, as in Eq. (2).

$$\rho(\mathcal{T}_t, \mathcal{S}_i) = \sum_{j=1}^{|\mathcal{A}|} \sigma_j \cdot x_{e,i,j} + \sum_{u=1}^{|\mathcal{R}|} h_u \cdot z_{e,i,u} \tag{2}$$

4. Proposed strategy

This section presents our resource management strategy, which employs a threshold-based approach to proactively relocate applications and registries based on user mobility, avoiding delay and provisioning time SLA violations.

Algorithm 2: Proposed resource management strategy.

```
1  foreach time step $\mathcal{T}_t \in \mathcal{T}$ do
2      $A$ = Applications in $\mathcal{A}$ sorted by Eq. (3) (asc.)
3      $\phi = \{\}$
4      foreach $\mathcal{A}_j \in A$ do
5          if $\eta(\mathcal{T}_t, \mathcal{A}_j) > \beta_j \cdot \lambda$ then
6              $\varkappa$ = User that accesses $\mathcal{A}_j$
7              $S$ = Edge servers sorted by delay to $\varkappa$ (asc.)
8              foreach $\mathcal{S}_i \in S$ do
9                  if $\mathcal{A}_j$ is already hosted by $\mathcal{S}_i$ then
10                     break
11                 else
12                     if $c_i - \rho(\mathcal{T}_t, \mathcal{S}_i) \geq \sigma_j$ then
13                         $pt = provisionApp()$
14                         if $pt > \gamma_j \cdot \rho$ then
15                             $\phi = \phi \cup \{\varkappa\}$
16                         break
17     Deprovision container registries distant from users
18     $\psi$ = List of servers with capacity to host a registry
19     while $|\phi| > 0$ and $|\psi| > 0$ do
20         $\theta = \{\}$
21         foreach $\mathcal{S}_i \in \psi$ do
22             foreach $\mathcal{U}_e \in \phi$ do
23                 $\mathcal{A}_j$ = Application accessed by $\mathcal{U}_e$
24                 $b$ = Bandwidth between $\mathcal{S}_i$ and $\mathcal{U}_e$
25                 if $\frac{\sigma_j}{b} \leq \gamma_j \cdot \rho$ then
26                     $\theta_i = \theta_i \cup \{\mathcal{U}_e\}$
27         $\mathcal{S}_i$ = Edge server in $\psi$ with the largest $\theta$
28         if $|\theta_i| > 0$ then
29             Provision a container registry on $\mathcal{S}_i$
30             $\psi = \psi - \{\mathcal{S}_i\}$
31             $\phi = \phi - \theta_i$
32         else
33             $\psi = \{\}$
```

The proposed strategy initially arranges the list of applications according to a score function ∂ (Eq. (3)), which sorts them based on how much their actual delay exceeds their delay threshold (Algorithm 2, line 2). In that way, applications with the most intense delay issues are migrated first.

$$\partial(\mathcal{T}_t, \mathcal{A}_j) = \beta_j - \eta(\mathcal{T}_t, \mathcal{A}_j) \tag{3}$$

Rather than migrating all applications, the proposed algorithm tries to migrate only those whose delay is close enough to their delay SLAs.

Table 3
Overall settings of the three scenarios considered in the evaluation.

Configuration	Number of users/Applications	Infrastructure demand
Scenario 1	55	25%
Scenario 2	235	50%
Scenario 3	415	75%

The proximity limit between the actual delay and the SLA delay is defined through a λ threshold (Algorithm 2, line 5). When migrating an application $\mathcal{A}_j$, the proposed algorithm gathers the list of edge servers candidates for hosting it, sorting these servers by their delay to $\mathcal{A}_j$'s user (Algorithm 2, line 7). If the closest edge server already hosts $\mathcal{A}_j$, no migration is performed (Algorithm 2, lines 9–10). Otherwise, $\mathcal{A}_j$ is migrated to the edge server closest to its user (Algorithm 2, lines 12–16). If $\mathcal{A}_j$'s migration time exceeds a threshold ρ, $\mathcal{A}_j$'s user is added to a list ϕ, which corresponds to the list of users accessing applications whose provisioning time SLAs are near to be violated (Algorithm 2, line 15).

After performing application migrations, our algorithm deprovisions all registries that are not the closest to any of the users in the environment (Algorithm 2, line 17). Then, it starts making decisions to provision new registries to prevent provisioning time SLA violations from happening. In such a process, the algorithm starts gathering a list of edge servers eligible for hosting a container registry (Algorithm 2, line 18). Then, it starts an iterative process that seeks to find suitable registry locations (Algorithm 2, lines 19–33). Such iterative process repeats until registries are provisioned close enough to all users in ϕ, or no edge server in the infrastructure has enough resources to host a registry.

Without loss of generality, our strategy assumes the existence of network QoS and computing resource reservation policies within the infrastructure, as in Souza et al. (2022a,b). As such, there is no room for conflict over access to resources shared between co-hosted applications and registries. In addition, ongoing migrations (either from registries or applications) do not affect each other's bandwidth or the network delay of running applications.

When choosing edge servers to host registries, the algorithm stores a list θ_i for each server $\mathcal{S}_i \in \psi$, with the number of users whose provisioning time SLA violations could be avoided if $\mathcal{S}_i$ hosted a registry (Algorithm 2, lines 21–26). Once θ is obtained for each $\mathcal{S}_i \in \psi$, the server with the largest θ (i.e., the one that could avoid the largest number of upcoming provisioning time SLA violations) is picked for hosting a registry (Algorithm 2, line 29).

5. Performance evaluation

This section details the experiments conducted to validate the proposed heuristic against baseline migration strategies on edge computing environments. First, we describe the adopted experimental method (Section 5.1) and a sensitivity analysis that defines the proposed heuristic's thresholds (Section 5.2). Finally, we present the achieved results in three scenarios (Sections 5.3–5.5).

5.1. Experiments description

We consider three edge computing scenarios with different occupation rates during the evaluation: 25%, 50%, and 75%. The main difference in the parameters of these scenarios that yield the different occupation rates is the number of users and applications in the environment. A detailed description of the chosen parameters is presented in Table 3.

In the three scenarios, the edge infrastructure comprises 50 edge servers with two capacity configurations (700 and 1400) distributed uniformly and wireless antennas with delay = 10. The network topology interconnecting the edge servers is created with the Barabási–Albert model (Barabási and Albert, 1999), with links configured with delay = {5 ms, 10 ms} and bandwidth = {1, 2}, distributed uniformly. The initial applications' placement is defined with the First-Fit heuristic described in Algorithm 3.

Algorithm 3: Initial application placement heuristic.

```
1 $\mathcal{A}' \leftarrow$ List of applications in $\mathcal{A}$
2 $\mathcal{S}' \leftarrow$ List of edge servers in $\mathcal{S}$
3 foreach $\mathcal{A}'_j \in \mathcal{A}'$ do
4     foreach edge server $\mathcal{S}'_i \in \mathcal{S}'$ do
5         if $\mathcal{S}'_i$ has capacity to host $\mathcal{A}'_j$ then
6             Host application $\mathcal{A}'_j$ on edge server $\mathcal{S}'_i$
7             break
```

Without loss of generality, we assume an one-to-one relationship between applications and users. In addition, users move within the map according to the Pathway mobility model (Bai and Helmy, 2004). Unlike random mobility models, which only represent open areas where entities can move freely, the Pathway model considers geographic restrictions when defining the mobility of simulated entities (Khider et al., 2007). This allows a more accurate representation of scenarios like cities, where constructions limit the possible routes within the map (Ahmed et al., 2010).

Several works have adopted the Pathway model in Wireless Sensor Networks (Taleb et al., 2013) and Edge Computing scenarios (Mahmud et al., 2022; Souza et al., 2022a), typically representing the map as a graph whose edges denote the streets and freeways that link the vertices, which indicate strategic points that affect mobility (e.g., buildings and street corners). In such a setting, mobile entities (e.g., users with mobile devices and connected cars) can only move from one vertex to another throughout the edges.

Our evaluation considers two classes of applications based on representative use cases for the edge: (i) remote UAV control applications for farm machinery (McEnroe et al., 2022) and (ii) augmented reality applications for remote healthcare (e.g., monitoring and medical procedure guidance) (Hartmann et al., 2022). As these classes of applications can be implemented on top of multiple software stacks (e.g., operating systems, programming languages, libraries, etc.), they are represented in our datasets through multiple container image specifications to illustrate the heterogeneous software dependencies of large-scale infrastructures.

The delay SLA values used in our dataset (30 ms and 60 ms) are real requirements of UAV and augmented reality applications specified in the 3rd Generation Partnership Project (3GPP) (3GPP, 2022). As we could not find real provisioning time SLAs, as they greatly vary based on scenario-specific constraints, provisioning time SLA values used in our dataset (80 s and 160 s) are set arbitrarily based on the size of container images and the network bandwidth specifications within the datasets. Delay and provisioning time SLA values are uniformly distributed across applications.

At the beginning of the simulation, we assume that there are 15 registries distributed randomly across the edge servers. We consider three types of container layers with different possible sizes: operating system (35, 40, and 45), runtime (20, 25, 30), and provided service (5, 10, 15). In this setting, the container image of each application is built based on three layers, one of each type (operating system, runtime, and provided service). For simplicity, server capacities and image sizes are denoted through abstract size units that could comprehend multiple resource specifications (e.g., CPU, RAM, and disk).

There is no comparison parameter with our strategy in the literature, as no other approach allocates registries dynamically in edge infrastructures. Thus, we compare our strategy against two naive migration strategies presented by Yao et al. (2015), called Never Follow and Follow User.

Although Never Follow and Follow User perform no registry allocation decisions based on users' mobility, they allow us to outline the impact of provisioning decisions on the availability and performance of applications. Never Follow does not migrate applications regardless of users' mobility. As such, it serves as the lower bound for migration decisions, allowing us to identify how much mobility-aware policies reduce

Table 4
Sensitivity analysis of delay (α) and provisioning time (ρ) thresholds.

Configuration	Cost (α)		
	Scenario 1	Scenario 2	Scenario 3
1. $\lambda = 0.7; \rho = 0.7$	**14**	79.5	**145.5**
2. $\lambda = 0.7; \rho = 0.8$	**14**	**79**	**145.5**
3. $\lambda = 0.7; \rho = 0.9$	14.5	81.5	150
4. $\lambda = 0.7; \rho = 1.0$	15	81.5	149
5. $\lambda = 0.8; \rho = 0.7$	**14**	**79**	146
6. $\lambda = 0.8; \rho = 0.8$	**14**	79.5	146
7. $\lambda = 0.8; \rho = 0.9$	14.5	81	146
8. $\lambda = 0.8; \rho = 1.0$	15	82.5	148.5
9. $\lambda = 0.9; \rho = 0.7$	24	109.5	201
10. $\lambda = 0.9; \rho = 0.8$	24	109.5	193.5
11. $\lambda = 0.9; \rho = 0.9$	24.5	110	196
12. $\lambda = 0.9; \rho = 1.0$	23.5	111.5	197.5
13. $\lambda = 1.0; \rho = 0.7$	34.5	166	269.5
14. $\lambda = 1.0; \rho = 0.8$	34.5	166	268
15. $\lambda = 1.0; \rho = 0.9$	35	167.5	270.5
16. $\lambda = 1.0; \rho = 1.0$	35	168.5	271.5

delay-related issues against a baseline scenario without migrations. Follow User complements the evaluation with the opposite approach to Never Follow. Follow User saturates the infrastructure by unnecessarily migrating applications when their users are still close enough to them, and their delay SLA is still being respected.

Whereas Never Follow is a lower bound for migration decisions, highlighting their improvements regarding reductions in delay issues against static placements, Follow User is an upper bound for that goal as it always keeps applications close to their users regardless of the number of unnecessary migrations performed. Accordingly, comparing our strategy with Never Follow and Follow User allows us to understand how well our proposal improves the trade-off between the number of migrations and the reduction of application delay issues.

Our experiments evaluate the compared strategies regarding the number of SLA violations (delay and provisioning time), number of migrations, average migration time, number of provisioned registries, and number of images provisioned inside the registries. The tests were conducted on a Linux machine with Ubuntu 20.04.3 LTS containing 8 CPU cores and 16 GB of RAM, configured with Python 3.8.10 (GCC 9.3.0). The experiments assets, including the source code of our simulator and the dataset, are publicly available at our GitHub repository.[1]

5.2. Sensitivity analysis

This section presents the sensitivity analysis of the proposed heuristic's delay (λ) and provisioning time (ρ) thresholds. Without loss of generality, we consider four values (0.7, 0.8, 0.9, and 1) for the λ and ρ thresholds. As a comparison parameter, we use a cost function α that considers the arithmetic mean between the number of delay and provisioning time SLA violations achieved by the proposed algorithm in each threshold configuration. Accordingly, the configurations that achieve a lower α are chosen for each of the three scenarios. Table 4 presents the sensitivity analysis results.

Lower provisioning thresholds (i.e., configurations 1, 2, 5, and 6) produced the best results in the evaluated scenarios. Such results demonstrate that relocating applications and registries as soon as application delays and provisioning times start to grow yields positive results, especially when the infrastructure has few available resources (e.g., scenario 3). As multiple configurations tied with the best results, we chose the best configurations with the lowest thresholds for the three scenarios (configurations 1, 2, and 1, respectively).

[1] Experimental assets: https://github.com/paulosevero/registry_migration.

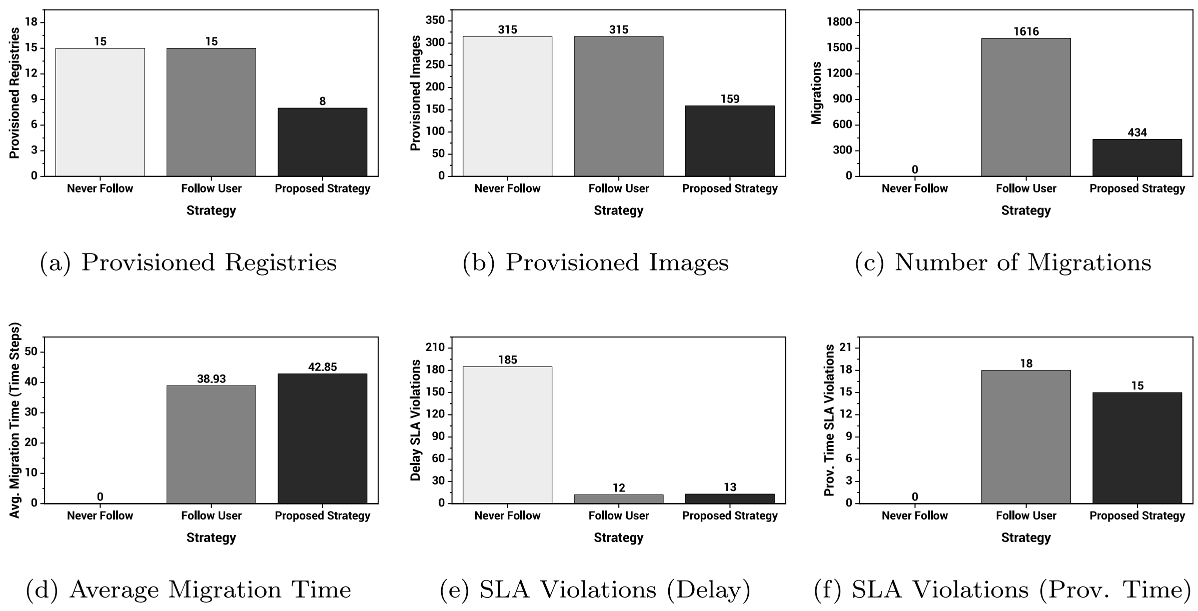


Fig. 2. Experimental results of the first evaluated scenario (low occupation).

5.3. Scenario 1

This section presents the results achieved in the first evaluation scenario, where the 50 edge servers in the infrastructure host 55 applications, resulting in an average infrastructure demand of 25%.

Figs. 2(a) and (b) show the average number of provisioned registries and container images, respectively. As static strategies, Never Follow and Follow User keep the set of resources defined in the initial placement during the whole simulation (15 registries hosting 315 images). Given the low number of users in the environment, the proposed strategy reduced the number of registries and container images by 46.67% and 49.52% on average, respectively.

Figs. 2(c) and (d) present the migration results. Whereas Never Follow performed no migration as expected, Follow User migrated services 1616 times during the simulation. As Follow User relocated services whenever users moved between the coverage area of different edge servers, it presented the lowest average migration time. In comparison to Follow User, the proposed strategy reduced the number of migrations by 73.14% while increasing the average migration time by 9.15%.

Figs. 2(e) and (f) depict the SLA violation results. Regarding the number of delay SLA violations, Never Follow presented the worst results by maintaining services in their original positions regardless of users' mobility. Follow User and the proposed strategy had similar results (12 and 13 delay SLA violations, respectively), reducing delay SLA violations by 93.51% and 92.97% compared to Never Follow. These results demonstrate the effectiveness of the proposed strategy, which reduced the number of delay SLA violations almost as much as Follow User, but with far fewer migrations.

As provisioning time SLA violations are counted according to migrations, Never Follow had no violations. At the same time, the proposed strategy presented the second-best results, reducing the number of provisioning time SLA violations from 18 to 15 compared to Follow User by relocating registries dynamically during the simulation.

5.4. Scenario 2

This section presents the results for the second scenario considered during our evaluation, where the 50 edge servers host 235 applications, resulting in an average infrastructure demand of 50%.

Figs. 3(a) and (b) present the resource usage results (i.e., the average number of provisioned registries and container images) achieved by the evaluated strategies. As this scenario is 76.6% more populated than the first one in terms of number of users, the proposed strategy needed to allocate 2 more registries than Never Follow and Follow User to ensure that applications were provisioned within a satisfactory time frame.

Figs. 3(c) and (d) show the number of migrations and average migration time of the compared strategies during the tests. The proposed strategy performed 71.96% fewer migrations than Follow User while slightly reducing the average migration time (from 42.50% to 42.16%). Comparing such results with the ones from the first scenario, we observe the increasing complexity of dynamically relocating registries based on users' mobility in more populated scenarios.

Figs. 3(e) and (f) present the SLA violation results. As the infrastructure has a higher occupation rate in this scenario, the timely application relocations performed by the proposed strategy produced the best results, reducing the number of delay SLA violations by 4.29% compared to Follow User, which was the second-best strategy.

The proposed strategy also reduced the number of provisioning time SLA violations by 50.81% compared to Follow User, thanks to its dynamic registry relocation approach. When comparing these results with those of the first scenario, we observe an increasing reduction in the provisioning time SLA violations obtained by the proposed strategy, highlighting the criticality of dynamically relocating registries in providing sufficiently low application provisioning times in more populated scenarios.

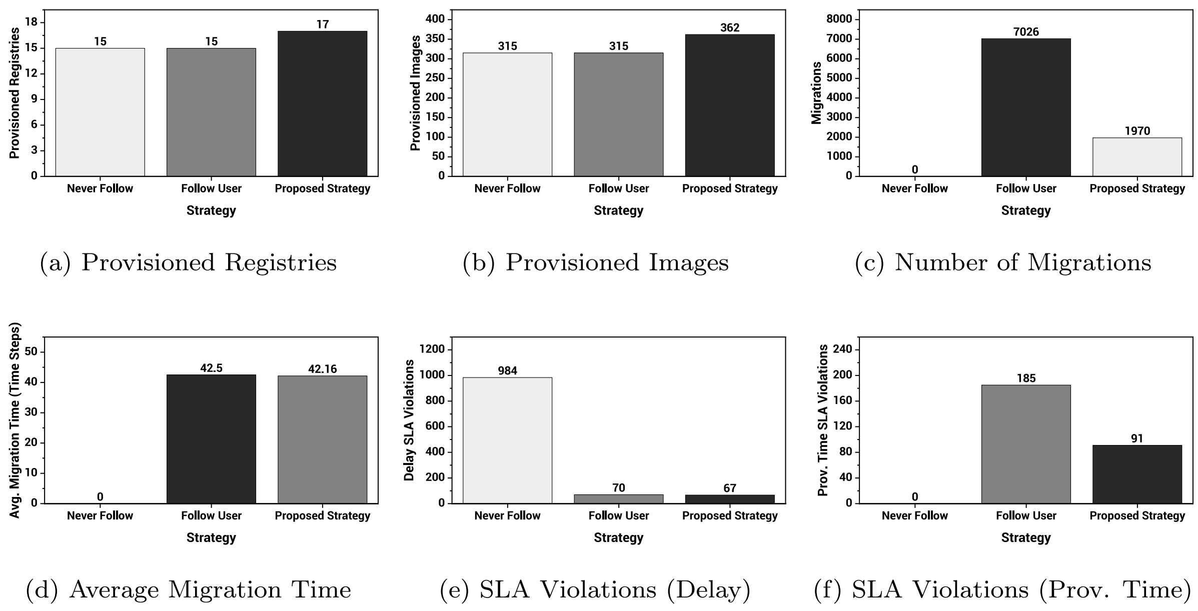
5.5. Scenario 3

This section presents the results achieved in the third evaluation scenario, where the 50 edge servers in the infrastructure host 415 applications, resulting in an average infrastructure demand of 75%.

Figs. 4(a) and (b) show the number of provisioned registries and container images in the infrastructure. Given that many users are present in the environment, the proposed strategy allocated 28.57% more registries than Never Follow and Follow User.

Figs. 4(c) and (d) present the migration results. As observed in the previous scenarios, while Never Follow performs no migration regardless of users' mobility, the proposed strategy reduced the number of migrations (67.45%) and the average migration time (2.02%) compared to Follow User.

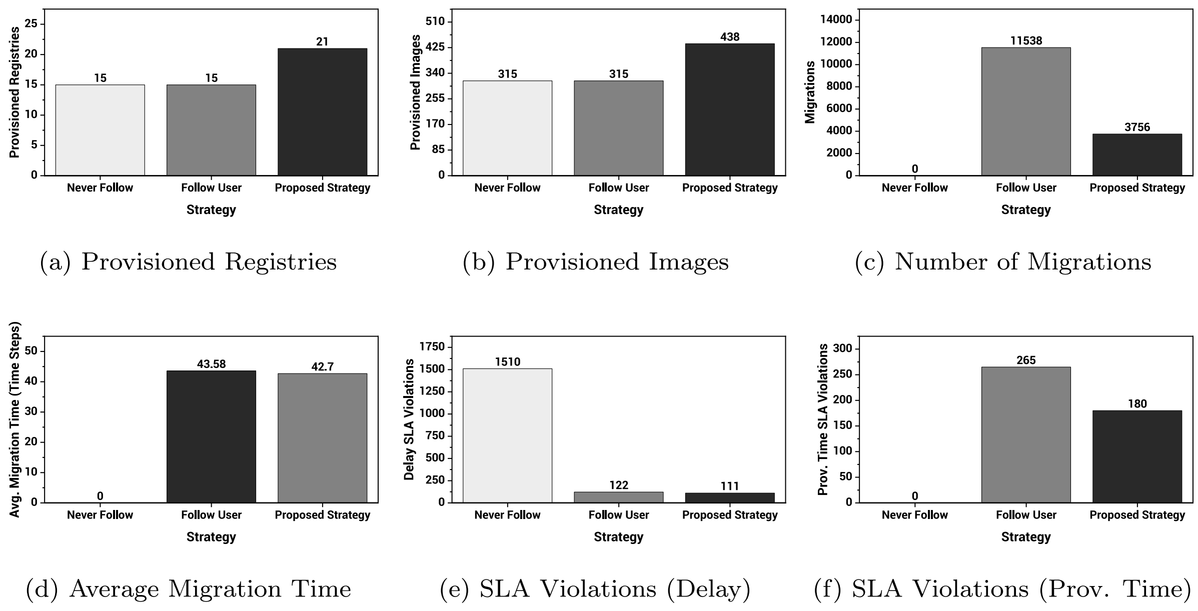
Figs. 4(e) and (f) show the SLA violation results. Regarding delay SLA violations, Never Follow got the worst results by performing



(a) Provisioned Registries (b) Provisioned Images (c) Number of Migrations

(d) Average Migration Time (e) SLA Violations (Delay) (f) SLA Violations (Prov. Time)

Fig. 3. Experimental results of the second evaluated scenario (medium occupation).



(a) Provisioned Registries (b) Provisioned Images (c) Number of Migrations

(d) Average Migration Time (e) SLA Violations (Delay) (f) SLA Violations (Prov. Time)

Fig. 4. Experimental results of the third evaluated scenario (high occupation).

no migrations during the simulation. At the same time, Follow User achieved the second-best results, reducing the number of delay SLA violations by 91.92% compared to Never Follow.

In this scenario, the proposed strategy reduced the number of delay SLA violations by 9.02% compared to Follow User, possibly because performing unnecessary migrations leads to more severe consequences as the infrastructure is more utilized. Regarding provisioning time SLA violations, the proposed strategy achieved the best results, reducing the number of violations by 32.08% compared to Follow User.

6. Conclusion and future work

Container-based virtualization has been acknowledged as a prime architecture for achieving scalability and better use of resources on edge computing infrastructures. Once applications are encapsulated inside containers, which have a smaller footprint than virtual machines, edge servers are less likely to be impacted by the virtualization overhead, which is pleasing given the resource constraints of edge infrastructures. At the same time, the agility of containers allows dynamically moving applications across the infrastructure as their users move across the environment.

However, managing containerized applications at the edge implies handling a significant set of challenges. For instance, timely provisioning of containerized applications according to users' mobility yields intense communication across the edge infrastructure, as container images must be transferred from container registries to wherever applications must be provisioned.

State-of-the-art placement strategies distribute registries in the infrastructure to mitigate potential hotspots that lead to prolonged provisioning times. However, as the number of users in the environment grows, more registries are provisioned to couple with the increasing demand, which may become inconvenient in resource-constrained edge infrastructures, as registries can saturate resources that could be better used to host applications.

This work envisions a novel approach based on thresholds that spins up new registries when application provisioning times are becoming too high while existing registries away from users are deprovisioned to avoid resource wastage. Experimental results demonstrate that our approach reduces the application provisioning time issues by 33.19% on average compared to strategies that allocate container registries statically. As future work, we highlight the following research opportunities:

- Leveraging predictive algorithms to forecast user mobility and make proactive application and container registry provisioning decisions.
- Characterizing and mitigating performance degradation issues caused by competing migrations in the network and co-hosted applications.
- Designing resource management algorithms that make optimized decisions in scenarios where multiple users in different locations access the same application.

CRediT authorship contribution statement

Daniel Chaves Temp: Conceptualization, Methodology, Software, Writing – original draft. **Paulo Silas Severo de Souza:** Conceptualization, Methodology, Software, Writing – original draft. **Arthur Francisco Lorenzon:** Conceptualization, Writing – original draft, Writing – review & editing. **Marcelo Caggiani Luizelli:** Conceptualization, Writing – original draft, Writing – review & editing. **Fábio Diniz Rossi:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing.

Declaration of competing interest

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

Data availability

We shared the data via GitHub and the link is in the paper.

Acknowledgments

This work was partially funded by National Council for Scientific and Technological Development (CNPq 404027/2021-0), Foundation for Research of the State of Sao Paulo (FAPESP 2021/06981-0, 2021/00199-8, 2020/05183-0), and Foundation for Research of the State of Rio Grande do Sul (21/2551-0000688-9).

References

3GPP, 2022. 5G; Service requirements for the 5G system (3GPP TS 22.261 version 16.16.0 Release 16). Technical specification (ts), 3rd Generation Partnership Project (3GPP), pp. 46–51, URL https://portal.etsi.org/webapp/workprogram/Report_WorkItem.asp?WKI_ID=64134.

Ahmed, S., Karmakar, G.C., Kamruzzaman, J., 2010. An environment-aware mobility model for wireless ad hoc network. Comput. Netw. 54 (9), 1470–1489.

Ahmed, A., Pierre, G., 2018. Docker container deployment in fog computing infrastructures. In: IEEE International Conference on Edge Computing. pp. 1–8.

Ahmed, A., Pierre, G., 2019. Docker image sharing in distributed fog infrastructures. In: International Conference on Cloud Computing Technology and Science. IEEE, pp. 135–142.

Anwar, A., Mohamed, M., Tarasov, V., Littley, M., Rupprecht, L., Cheng, Y., Zhao, N., Skourtis, D., Warke, A.S., Ludwig, H., et al., 2018. Improving docker registry design based on production workload analysis. In: USENIX Conference on File and Storage Technologies. USENIX, pp. 265–278.

Aral, A., Demaio, V., Brandic, I., 2021. ARES: Reliable and sustainable edge provisioning for wireless sensor networks. IEEE Trans. Sustain. Comput. 1–12.

Bai, F., Helmy, A., 2004. A survey of mobility models. In: Wireless Adhoc Networks, Vol. 206. University of Southern California, USA, p. 147.

Barabási, A.-L., Albert, R., 1999. Emergence of scaling in random networks. Science 286 (5439), 509–512.

Becker, S., Schmidt, F., Kao, O., 2021. EdgePier: P2P-based container image distribution in edge computing environments. In: International Performance, Computing, and Communications Conference. IEEE, pp. 1–8.

Chen, J.L., Liaqat, D., Gabel, M., de Lara, E., 2022. Starlight: Fast container provisioning on the edge and over the WAN. In: USENIX Symposium on Networked Systems Design and Implementation. USENIX, pp. 35–50.

Dijkstra, E.W., et al., 1959. A note on two problems in connexion with graphs. Numer. Math. 1 (1), 269–271.

Gannon, D., Barga, R., Sundaresan, N., 2017. Cloud-native applications. IEEE Cloud Comput. 4 (5), 16–21.

Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., 2016. Slacker: Fast distribution with lazy docker containers. In: USENIX Conference on File and Storage Technologies. USENIX, pp. 181–195.

Hartmann, M., Hashmi, U.S., Imran, A., 2022. Edge computing in smart health care systems: Review, challenges, and research directions. Trans. Emerg. Telecommun. Technol. 33 (3), e3710.

Khider, I., Furong, W., Hua, Y.W., et al., 2007. A survey of geographic restriction mobility models. J. Appl. Sci. 7 (3), 442–450.

Knob, L.A.D., Faticanti, F., Ferreto, T., Siracusa, D., 2021. Community-based placement of registries to speed up application deployment on edge computing. In: International Conference on Cloud Engineering. IEEE, pp. 147–153.

Mahmud, R., Pallewatta, S., Goudarzi, M., Buyya, R., 2022. iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments. J. Syst. Softw. 190, 111351.

McEnroe, P., Wang, S., Liyanage, M., 2022. A survey on the convergence of edge computing and AI for UAVs: Opportunities and challenges. IEEE Internet Things J. 9 (17), 15435–15459.

Nathan, S., Ghosh, R., Mukherjee, T., Narayanan, K., 2017. Comicon: A co-operative management system for docker container images. In: International Conference on Cloud Engineering. IEEE, pp. 116–126.

Pallewatta, S., Kostakos, V., Buyya, R., 2019. Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments. In: IEEE/ACM International Conference on Utility and Cloud Computing. pp. 71–81.

Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N., 2009. The case for vm-based cloudlets in mobile computing. IEEE Pervasive Comput. 8 (4), 14–23.

Shi, W., Dustdar, S., 2016. The promise of edge computing. Computer 49 (5), 78–81.

Souza, P., Crestani, v., Rubin, F., Ferreto, T., Rossi, F., 2022a. Latency-aware privacy-preserving service migration in federated edges. In: International Conference on Cloud Computing and Services Science. pp. 288–295.

Souza, P.S., Ferreto, T.C., Rossi, F.D., Calheiros, R.N., 2022b. Location-aware maintenance strategies for edge computing infrastructures. IEEE Commun. Lett. 26 (4), 848–852.

Taleb, A.A., Alhmiedat, T., Hassan, O.A.-H., Turab, N.M., 2013. A survey of sink mobility models for wireless sensor networks. J. Emerg. Trends Comput. Inf. Sci. 4 (9), 679–687.

Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A., 2013. Performance evaluation of container-based virtualization for high performance computing environments. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 233–240.

Yao, H., Bai, C., Zeng, D., Liang, Q., Fan, Y., 2015. Migrate or not? Exploring virtual machine migration in roadside cloudlet-based vehicular cloud. Concurr. Comput.: Pract. Exper. 27 (18), 5780–5792.

Daniel Chaves Temp is a Master student in electrical engineering at the Federal University of Pampa. His research interest focuses on resource allocation in cloud–edge infrastructures.

Paulo Silas Severo de Souza is a Ph.D. candidate in computer science at the Pontifical Catholic University of Rio Grande do Sul. His research interest lies primarily in the fields of resource management, Cloud Computing, Edge Computing, and algorithms.

Arthur Francisco Lorenzon is an Associate professor at the Federal University of Rio Grande do Sul. His areas of interest include parallelism exploitation aiming energy efficiency and the development of approaches to automate the TLP exploitation.

Marcelo Caggiani Luizelli is an Associate Professor at the Federal University of Pampa. His research interest broadly focuses on networking and combinatorial optimization, focusing on NFV, SDN, and PDPs.

Fábio Diniz Rossi is a Full Professor at the Federal Institute Farroupilha. His research interest is focused on resource allocation in cloud–edge infrastructures.