# Dynamic Service Placement and Load Distribution in Edge Computing

Adyson Magalhaes Maia, Yacine Ghamri-Doudane, Dario Vieira, Miguel Franklin de Castro

# Dynamic Service Placement and Load Distribution in Edge Computing

Adyson Magalhães Maia*†, Yacine Ghamri-Doudane†, Dario Vieira‡, Miguel Franklin de Castro*
*Federal University of Ceará (UFC), GREat Lab, Fortaleza, Brazil
†La Rochelle University, L3i lab, La Rochelle, France
‡Engineering School of Information and Digital Technologies (EFREI), Villejuif, France
Email: adysonmaia@great.ufc.br, yacine.ghamri@univ-lr.fr, dario.vieira@efrei.fr, miguel@great.ufc.br

***Abstract*—Edge computing enables a wide variety of application services for the Internet of Things, including those with performance-critical requirements. To achieve this, it brings cloud computing capabilities to network edges. A key challenge therein is to decide where and when to place or migrate application services considering their load variation and seeking the optimization of multiple performance objectives. In this paper, we address this optimal service placement issue by further considering how to distribute the load of an application placed in different locations. By estimating the performance-cost trade-off of services migration, we propose a dynamic service placement and load distribution strategy that uses limited look-ahead prediction to handle load fluctuations. Evaluation analysis demonstrates that our proposal outperforms other benchmarks solutions in terms of multiple conflicting objectives.**

***Index Terms*—edge computing, internet of things, service placement, service migration, load distribution.**

## I. Introduction

Edge Computing (EC) bridges the gap between Cloud Computing (CC) and end-user devices by enabling computing, storage, networking, and data management on EC nodes (e.g., cellular base stations, routers, wireless access points, and mini data centers) within close vicinity of these end-user devices [1]–[3]. This proximity characteristic allows EC to support time-sensitive applications such as those made possible by the Internet of Things (IoT) (e.g., patient monitoring, real-time manufacturing, and self-driving cars). Indeed, CC with its distant data centers cannot satisfy the low latency requirements of such applications [1], [2].

Although EC brings various benefits to many of nowadays applications, it also faces some challenges to deliver their services due to $(i)$ the resource constraints in the edge network, $(ii)$ the geo-dispersed EC nodes, $(iii)$ the heterogeneity of application requirements, and $(iv)$ the dynamic services demands [1], [2]. In this context, an important issue is deciding where to place multiple applications, i.e., selecting EC nodes with hosting capability to deploy and run applications according to the demands, constraints, and performance criteria. As an EC node may not have sufficient resources to handle all user-generated loads for a specific application, another related challenge is how to efficiently distribute these loads among multiples nodes [1]. Indeed, an application (service) placement decision can also affect the load distribution (and vice versa). For instance, it is only possible to distribute requests to nodes hosting the requested application. Thus, an optimal decision strategy requires a joint optimization of these two aspects [4].

Speaking about an application load, this one might vary in both spatial and temporal domains due to, for instance, user mobility. This dynamic load implies re-evaluating, over time, the placement and distribution decisions to maintain satisfactory service performance. However, decision adjustments could also lead to additional operational or performance costs. For example, an application can be migrated or replicated to a new location to keep low latency. Nevertheless, excessive reallocation may result in network overload or even latency degradation due to the migration operation itself, especially in the case of large applications [5], [6]. As a result, a dynamic service placement and load distribution strategy is needed, but it should consider the benefits and costs of reallocations.

Existing studies on service placement in EC do not address some of the above-discussed complexities, such as $(i)$ the dynamic load [7], [8]; $(ii)$ the load distribution [6], [9]; $(iii)$ the constrained resource capacity [5], [10]; $(iv)$ the applications heterogeneity, including the time-sensitive vs. time-tolerant feature [6], [11]; $(v)$ the optimization of multiple and possibly conflicting objectives [9], [12]; or $(vi)$ the impact of service migration on latency/response time [4], [12]. Therefore, we intend in this paper to propose and validate a solution covering all these aspects. Our proactive service placement and load distribution strategy uses the Limited Look-ahead Control (LLC) [13], [14] to predict and handle the fluctuations on the request loads. Our main contributions are as follows:

- We go far beyond our previous works [15], [16] by supporting dynamic request loads of end-user devices on an EC system model. This model considers different attributes for nodes (e.g., resource capacity and cost) and applications (e.g., response deadline, and resource demand) to place multiple application replicas within the network and distribute requests among these replicas.
- Using the LLC concept, we estimate how the modeled EC system evolves under controllable (placement and distribution) decisions and uncontrollable (user-generated load) events. Then, we jointly formulate the service placement and load distribution as an optimization problem of multiple performance criteria over a look-ahead prediction window while satisfying a set of constraints.
- We use a genetic algorithm to solve the formulated

problem in a discrete prediction window with length $H$. We first propose a chromosome representation and a decoder algorithm to obtain a single and valid control decision when $H = 1$. Then, two heuristics extending this genetic solution are proposed to solve the problem when $H > 1$. The first heuristic generates simple sequences of control decisions over the prediction window, while the second one produces more general control sequences.

The remainder of this paper is organized as follows: we review related work in Section II. Section III overviews the LLC concept. In Section IV, we describe our system model and formulate the targeted problem. Then, Section V presents our proposed heuristics. We conduct performance evaluations in Section VI, and Section VII concludes this paper.

## II. Related Work

There are studies in the literature that jointly address service placement and load distribution problems in the context of EC to minimize resource cost [7], response time [8], Quality of Service (QoS) violations [15], or multiple performance-related objectives [16]. However, these mentioned works do not handle spatial and temporal changes in applications loads.

Some works use a Follow-me Cloud/Edge approach to handle dynamic loads caused by user mobility. In this approach, each user is associated with a dedicated application to execute its offloaded tasks. Moreover, an application may be migrated to another EC node to follow user mobility and maintain satisfactory service performance. This approach, however, such as in [5], [6], [9], [10], usually ignores that several users may request the same application, and multiple replicas of this application can be in the system. As a result, this approach does not consider load distribution.

Few studies deal with dynamic service placement and load distribution problems in EC. In [11], the authors jointly model these problems as a Markov Decision Process (MDP) to optimize transmission and migration costs while providing maximum delay guarantees. Due to the computational complexity, the work relaxed and decoupled the problem into independent sub-problems. However, this relaxation replaces the maximum delay constraints by queue stability constraints, which only provides worst-case delay guarantees. Authors in [12] study dynamic Virtual Machine (VM) placement and request distribution to minimize network traffic from requests data and VMs migration. They propose a heuristic algorithm that places first VMs for serving the most critical flows, which are flows with higher bandwidth requirements or with a larger number of requests. The work in [4] addresses service placement and request scheduling for data-intensive applications, but decisions for these problems are separated in different time scales. Service placement happens on a larger scale to prevent system instability, while requests are scheduled on a smaller scale to support real-time services. It also imposes a budget constraint to control service migration costs. However, neither [12] nor [4] studies the impact of placement, migration, and distribution decisions on the application response time.

In this work, we overcome the limitations identified in the above-discussed studies when investigating dynamic service placement and load distribution in EC.

## III. Limited Look-ahead Control Overview

The Limited Look-ahead Control (LLC) [13], [14] describes the continuous dynamics of a system by the following discrete-time state-space equation:

$$s(t+1) = \phi\left(s(t), c(t), e(t)\right) \qquad (1)$$

where $t$ is the discrete-time index, $s(t)$ is the system state or output, $c(t)$ denotes the control input or decision variable, and $e(t)$ represents the environment input or disturbance at time step $t$. In general, environmental inputs (e.g., system incoming load) are uncontrollable, but they can be estimated using well-known forecasting techniques, such as AutoRegressive Integrated Moving Average (ARIMA). The system dynamics model $\phi(\cdot)$ captures the relationship between a system state and its (control and environment) inputs.

In LLC, an online controller estimates relevant environment parameters to be used by the system model $\phi(\cdot)$ to forecast future system behavior over a limited look-ahead prediction horizon. The controller optimizes the forecast behavior for a specified performance criteria by selecting the best control input to apply to the system. More specifically, at the beginning of a time step $t$, the LLC constructs a set of future states from the observed state $s(t)$ up to a prediction horizon $H$. It selects within this horizon a sequence $\pi_{c^*} = \{c^*(k) \mid k \in [t, t+H-1]\}$ of control decisions that optimizes the system performance while satisfying both state and inputs constraints. Then, the controller applies the first control input $c^*(t)$ of this sequence into the system. This process is repeated at time step $t+1$ when the new measured system state $s(t+1)$ is available.

Such predictive approach is well adapted to the problem we address in this paper. We show in the following how this can be combined with our targeted optimization problem.

## IV. System Model and Problem Formulation

### A. System Model

Based on our previous works [15], [16], we consider an EC system consisting of an Infrastructure Provider (InP), various Application Service Providers (ASPs), and end-user devices.

*1) InP:* An InP owns the EC infrastructure and provides a set $\mathcal{R}$ of different types of virtual resources to ASPs, such as $\mathcal{R} = \{\mathrm{CPU}, \mathrm{RAM}, \mathrm{DISK}\}$. We model the EC infrastructure as a unidirectional connected graph $G = (\mathcal{V}, \mathcal{E})$ of nodes $\mathcal{V}$ and links $\mathcal{E}$ that are geographically dispersed among end-users and a remote centralized cloud data center. Fig. 1 illustrates an EC system for a mobile network where nodes are located on the Radio Access Network (RAN), Core Network, and Cloud regions of the network.

In the graph $G$, a node can represent a (mini) data center, a wireless access point, a network router, or all of them at the same time if they are co-located. Each node $n \in \mathcal{V}$ has the following properties:

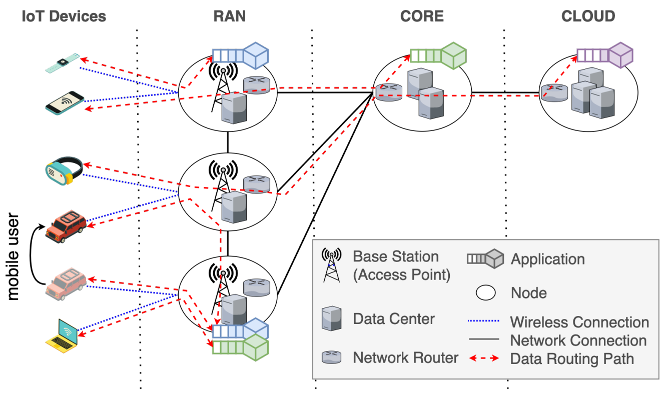


Fig. 1. Proposed Edge Computing system model for a mobile network.

- **Resource capacity** $N_{n,r}^{cap}$ is the total capacity for resource $r \in \mathcal{R}$ on node $n$. We assume that the cloud node has unlimited resources due to the capacity difference between such a system and a mini data center close to the users.
- **Usage cost** $N_{n,r}^{cost}(x)$ is a (monetary) cost function of allocating a specific amount $x$ of resource $r \in \mathcal{R}$ to an application on node $n$.

A link $l = (m, n) \in \mathcal{E}$ corresponds to a (physical or virtual) network connection between nodes $m$ and $n$, and it has the following attributes:

- **Bandwidth** $L_l^{bw}$ is the average transmission rate between end-points of link $l$.
- **Propagation delay** $L_l^{pd}$ is the time required for bits to reach the other end of link $l$.

*2) ASPs:* An ASP offers application services to end-user devices by renting on-demand resources from the InP to deploy its application on EC nodes through virtualization technologies, such as VMs or containers. However, ASPs do not directly determine where to deploy their applications. Instead, the InP is responsible for selecting the place where to deploy an application based on the requirements specified by each ASP. Let $\mathcal{A}$ be the set of all applications to be placed over the EC infrastructure. Then, each application $a \in \mathcal{A}$ has the following requirements:

- **Response deadline** $A_a^{rd}$ is the maximum time (i.e., deadline) allowed for application $a$ responding to a request.
- **Maximum number of replicas** $A_a^{max}$ of application $a$ that can be placed independently on different nodes.
- **Resource demand** $A_a^r(\lambda)$ is a function expressing the amount of resources $r \in \mathcal{R}$ that must be allocated to handle a load $\lambda$. Load $\lambda_a^n$ is defined as the arrival rate of requests for a replica of application $a$ placed on node $n$.
- **Work size** $A_a^{work}$ is the amount of processing required to get a response to a request for $a$. It is measured by the number of CPU instructions or clock cycles.
- **Data length** $A_a^{data}$ is the amount of data (in bits or bytes) in a request for application $a$ sent over the network.
- **Request rate** $A_a^{req}$ is the average request generation rate of an end-user device requesting application $a$. It follows a Poisson distribution.

*3) End-User Devices:* As shown in Fig. 1, a device is attached to a wireless access point node where it sends requests to be processed by an application. The EC system routes a request to a node hosting a replica of the required application based on a load distribution decision. This replica then puts the task in its waiting queue for processing where the result is sent back to the device as a response. In this way, users do not know where their requests are handled as multiple replicas of an application can be placed on the system, and their placement locations may change over time. Besides that and unlike our previous works [15], [16], users may move or be inactive without sending requests in this work, thereby changing the number of devices attached to each access point node over time. Hence, we model this dynamic by defining $u_a^n(t)$ as the number of active users connected to node $n$ requesting application $a$ at a time $t$.

### B. System Dynamics

We adopt the LLC concepts introduced in Section III to model the EC system behavior under dynamic loads. The following adjustable system parameters are designed as control inputs for $c(t) = (\rho(t), \delta(t))$:

- **Application placement** $\rho(t) = \{\rho_a^n(t)\}$ is a set of binary variables, where $\rho_a^n(t) \in \{0, 1\}$ indicates whether or not a replica of application $a$ should be placed on a node $n$ at time step $t$.
- **Load distribution** $\delta(t) = \{\delta_a^{m,n}(t)\}$ is a set of real variables, where $\delta_a^{m,n}(t) \in [0, 1]$ establishes the fraction of requests for an application $a$ that should be distributed from a node $m$ to another node $n$ at time $t$.

A relevant performance metric to observe in EC is the application response time. Therefore, we define a system state $s(t) = (d(t), q(t))$ as follows:

- **Response time** $d(t) = \{d_a^{m,n}(t)\}$ is the average response time of each request flow $\mathcal{F}_a^{m,n}(t)$ at the beginning of time step $t$. We define a request flow $\mathcal{F}_a^{m,n}(t)$ as the requests rate for application $a$ from node $m$ (source node) being handled by a replica of $a$ on node $n$ (target node).
- **Queue length** $q(t) = \{q_a^n(t)\}$ is the number of requests waiting to be processed on each application replica deployed on the system at the beginning of time step $t$.

Here, the environment input $e(t) = \{Q_a^n(t)\}$ represents the rate of requests generated by users, where $Q_a^n(t) = A_a^{req} u_a^n(t)$ is the observed request generation rate from all users of an application $a$ attached to a node $n$ at the current time step $t$. As the actual values for an environment input within the horizon window cannot be measured until the next sampling instants, we propose the use of a forecasting technique to predict the environment input $\widehat{Q}_a^n(k)$ for each time step $k$ along the prediction horizon.

Regarding a system state $s(k + 1),\ k \in [t, t + H]$, we formulate in (2) its response time $d(k+1)$ as the combination of network $d_{net}$, request processing $d_{proc}$, and application initialization $d_{init}$ delays. These delays and the queue length $q(k+1)$ are estimated in the remainder of this subsection.

$$d_a^{m,n}(k+1) = d_{net}^{a,m,n}(k+1)+d_{proc}^{a,n}(k+1)+d_{init}^{a,n}(k+1) \tag{2}$$

*1) Network Delay:* Eq. (3) expresses the average network delay of a request flow $\mathcal{F}_a^{m,n}(k+1)$ as the average time to send requests from the source node $m$ to the target node $n$, where $\mathcal{P}_{m,n}$ contains the links in a routing path from $m$ to $n$. This path can be obtained a priori by some shortest routing path algorithm, such as Dijkstra or Floyd–Warshall algorithms.

$$d_{net}^{a,m,n}(k+1) = \begin{cases} 0 & \text{if } m = n \\ \displaystyle\sum_{l\in\mathcal{P}_{m,n}} \frac{A_a^{data}}{L_l^{bw}} + L_l^{pd} & \text{otherwise} \end{cases} \tag{3}$$

*2) Processing Delay:* For a replica of application $a$ running on a node $n$, we model its processing as an M/M/1 queue. Moreover, let $\Lambda_a^n(k)$ and $\lambda_a^n(k)$ be the request arrival rate before and after load distribution at time step $k$, respectively. In (4), the arrival rate $\Lambda_a^n(k)$ is given by the predicted environment input $\widehat{Q}_a^n(k)$ plus the estimated queue length, which is converted to a rate value using $T_s$ as the sampling period, i.e., the time step duration. Meanwhile, the control input $\delta(t)$ regulates the arrival rate $\lambda_a^n(k)$, as shown in (5).

$$\Lambda_a^n(k) = \widehat{Q}_a^n(k) + \frac{q_a^n(k)}{T_s}\rho_a^n(k-1) \tag{4}$$

$$\lambda_a^n(k) = \sum_{m\in\mathcal{V}} \delta_a^{m,n}(k)\Lambda_a^m(k) \tag{5}$$

In (6), the average processing rate $\mu_a^n(k)$ is determined by the CPU speed $A_a^{CPU}(\cdot)$ allocated to the application replica, and the amount of CPU work $A_a^{work}$ necessary to process a request for this application.

$$\frac{1}{\mu_a^n(k)} = \frac{A_a^{work}}{A_a^{CPU}\left(\lambda_a^n(k)\right)} \tag{6}$$

According to the M/M/1 queueing model, (7) and (8) estimate the average queue length $q_a^n(k+1)$ and the average processing delay $d_{proc}^{a,n}(k+1)$, respectively.

$$q_a^n(k+1) = \frac{\lambda_a^n(k)}{\mu_a^n(k)-\lambda_a^n(k)} - \frac{\lambda_a^n(k)}{\mu_a^n(k)} \tag{7}$$

$$d_{proc}^{a,n}(k+1) = \frac{1}{\mu_a^n(k)-\lambda_a^n(k)} \tag{8}$$

*3) Initialization Delay:* In order to initialize the placement of an application on a selected node, an EC system migrates or replicates an instance of this application over the network from another node already hosting it to the selected node. Meanwhile, requests for this application arriving at the selected node need to wait for the migration/replication conclusion before they can be processed. Thus, the application migration delay is an impact factor to the response time. We can determine this migration delay as the time to transfer the application state, which includes the disk and RAM contents, from the nearest node hosting the application, as shown in (9).

$$d_{mig}^{a,n}(k) = \left(1-\rho_a^n(k-1)\right)\rho_a^n(k) \times \min_{m\in\mathcal{V}}\left\{d_{mig}^{a,m,n}(k)\rho_a^m(k-1)\right\} \tag{9a}$$

$$d_{mig}^{a,m,n}(k) = \sum_{l\in\mathcal{P}_{m,n}} \frac{A_a^{D+R}\left(\lambda_a^m(k-1)\right)}{L_l^{bw}} + L_l^{pd} \tag{9b}$$

$$A_a^{D+R}(\lambda) = A_a^{DISK}(\lambda) + A_a^{RAM}(\lambda) \tag{9c}$$

Assuming that a migration process starts and finishes at the same time step, then, not all requests and their response time are impacted by this process. Hence, let us define an application initialization delay as the impact of a migration process in the response time. In a M/M/1 queue, a node $n$ receives $\lambda_a^n(k)\Delta t$ requests on average for an application $a$ during a time interval $\Delta t$. We can split the migration delay into $M = \lceil d_{mig}^{a,m,n}(k)\rceil$ consecutive intervals of one unit of time (e.g., $\Delta t = 1$s). Then, during the $i$-th migration interval, $\lambda_a^n(k)$ requests arrive and wait for $M-i+1$ units of time until the migration is complete. We calculate $d_{init}^{a,n}(k+1)$ as a weighted average by using the arrived requests of each migration interval against all requests during sampling period $T_s$. We then approximate it by setting $M \approx d_{mig}^{a,m,n}(k)$, as shown in (10).

$$M = \lceil d_{mig}^{a,m,n}(k)\rceil \tag{10a}$$

$$d_{init}^{a,n}(k+1) = \frac{\sum_{i=1}^{M}\lambda_a^n(k)\left(M-i+1\right)}{\lambda_a^n(k)T_s} \approx \frac{d_{mig}^{a,n}(k)\left(d_{mig}^{a,n}(k)+1\right)}{2T_s} \tag{10b}$$

### C. Optimization Formulation

Let $F = \left(f_1,\dots,f_i,\dots,f_{|F|}\right)$ be a list of functions and $f_i$ a function that associates some performance for reaching and maintaining a system state. Then, at each time step $t$, the LLC controller aims to optimize the following problem:

$$\min_{c(k)\in\mathcal{C}} \sum_{k=t}^{t+H-1} F\left(s(k+1),c(k),e(k)\right) \tag{11a}$$

$$\text{s.t.} \quad s(k+1) = \phi\left(s(k),c(k),e(k)\right) \tag{11b}$$

$$1 \le \sum_{n\in\mathcal{V}} \rho_a^n(k) \le A_a^{max} \qquad \forall a\in\mathcal{A} \tag{11c}$$

$$\delta_a^{m,n}(k) \le \rho_a^n(k) \qquad \forall a\in\mathcal{A}, \forall m,n\in\mathcal{V} \tag{11d}$$

$$\sum_{i\in\mathcal{V}} \delta_a^{n,i}(k)\Lambda_a^n(k) = \Lambda_a^n(k) \qquad \forall a\in\mathcal{A}, \forall n\in\mathcal{V} \tag{11e}$$

$$\sum_{a\in\mathcal{A}} \rho_a^n(k)A_a^r\left(\lambda_a^n(k)\right) \le N_{n,r}^{cap} \quad \forall r\in\mathcal{R}, \forall n\in\mathcal{V} \tag{11f}$$

$$\lambda_a^n(k) < \mu_a^n(k) \qquad \forall a,n\left(\rho_a^n(k)=1\right) \tag{11g}$$

where $\mathcal{C}$ is the set of all possible control inputs and (11c) constraints the allowed number of application replicas placed in the system. Eq. (11d) restricts load distribution to nodes that host the requested application, while (11e) ensures the distribution of all loads. Constraint (11f) assures that the amount of resources allocated on a node does not exceed its capacity, and (11g) guarantees the processing queue stability. Then, a feasible solution for problem (11) is associated to a sequence of control decisions that satisfies all these constraints within the prediction horizon $H$.

For an optimization problem with multiple conflicting objectives, we can use the Pareto dominance concept to find a set of best trade-off solutions that cannot be improved in any objective without degrading other objectives. Formally, $x_1 \prec x_2$ expresses that a solution $x_1$ dominates and is better than another solution $x_2$ when:

$$\begin{aligned} x_1 \prec x_2 \text{ if } f_i(x_1) \leq f_i(x_2) \qquad & \forall i \in \{1, 2, \ldots, |F|\} \\ \text{and } f_j(x_1) < f_j(x_2) \qquad & \exists j \in \{1, 2, \ldots, |F|\} \end{aligned} \tag{12}$$

Feasible solutions that are not dominated by any other feasible solution are equally good/optimal if there is no additional preference information. However, for our optimization problem (11), a preference can be the improvement of time-sensitive applications. Hence, we define a dominance operator that prioritizes a selected function $f_i$ as follows:

$$\begin{aligned} x_1 \prec_i x_2 \text{ if } & f_i(x_1) < f_i(x_2) \\ \text{or } & (f_i(x_1) = f_i(x_2) \text{ and } x_1 \prec x_2) \end{aligned} \tag{13}$$

That is, it is sufficient that $f_i(x_1) < f_i(x_2)$ in order for $x_1$ to dominate $x_2$. Otherwise, if they have equal values for $f_i$, then the traditional Pareto dominance operator is used instead.

## V. Online Controller Algorithms

An LLC algorithm that exhaustively evaluates all possible control inputs presents an exponential increase in worst-case complexity with an increasing number of control inputs and longer prediction horizons [13]. Furthermore, not all control inputs produce a feasible solution satisfying all problem (11) constraints. A way to alleviate this complexity issue is by using (meta-)heuristic algorithms that find sub-optimal solutions in a reasonable time, and thus, trading optimality for speed. Hence, in this section, we propose some heuristic algorithms to solve (11). First, we present an algorithm that solves the problem for a single time step in Section V-A. Then, we extend this algorithm by taking into account a prediction horizon $H > 1$ in Section V-B.

### A. One Time Step Algorithm

In order to obtain feasible solutions for a multi-objective problem, we use our proposed genetic algorithm *BRKGA+NSGA-II* [16], which is a combination of two genetic algorithms: $(i)$ Biased Random-Key Genetic Algorithm (BRKGA) [17], and $(ii)$ Non-dominated Sorting Genetic Algorithm II (NSGA-II) [18]. *BRKGA+NSGA-II* evolves a population of individuals toward better solutions over several generations. Each individual has a corresponding chromosome represented by a vector of real numbers in the interval $[0, 1]$. A deterministic algorithm, named decoder, takes any chromosome as input and associates it with a feasible solution. Then, the genetic algorithm ranks these solutions according to a multi-objective dominance and diversity operators. Individuals associated with best-ranked solutions are kept in the next generation as an elite group. The next generation also comprises offspring resulting from a crossover between elite and non-elite parents, and mutant individuals randomly generated.

*1) Chromosome Representation:* In *BRKGA+NSGA-II*, the chromosome representation and decoder algorithm play essential roles as the problem-dependent portion of the algorithm. However, [16] only considers a static load, and thus, its problem-dependent part cannot be applied in a dynamic load case. Therefore, we need to propose a new chromosome representation and decoder algorithm to produce valid control inputs that satisfy problem (11) constraints for any system state and load values at a single time step within the prediction horizon. Then, the following chromosome encode is proposed:

$$\begin{aligned} C = \Big[ & C_{\text{I}}^{1}, C_{\text{I}}^{2}, \ldots, C_{\text{I}}^{|\mathcal{A}|}, \\ & C_{\text{II}}^{1,1}, C_{\text{II}}^{1,2}, \ldots, C_{\text{II}}^{1,|\mathcal{V}|}, \ldots, C_{\text{II}}^{|\mathcal{A}|,1}, C_{\text{II}}^{|\mathcal{A}|,2}, \ldots, C_{\text{II}}^{|\mathcal{A}|,|\mathcal{V}|}, \\ & C_{\text{II}}^{1,1}, C_{\text{III}}^{1,2}, \ldots, C_{\text{III}}^{1,|\mathcal{V}|}, \ldots, C_{\text{III}}^{|\mathcal{A}|,1}, C_{\text{III}}^{|\mathcal{A}|,2}, \ldots, C_{\text{III}}^{|\mathcal{A}|,|\mathcal{V}|} \Big] \end{aligned}$$

where

- $C_{\text{I}}^{a}$ is the fraction of nodes to be candidates for hosting application $a$.
- $C_{\text{II}}^{a,n}$ is the priority to place application $a$ in node $n$.
- $C_{\text{III}}^{a,m}$ is the priority to distribute requests for application $a$ from a (source) node $m$.

*2) Chromosome Decoder:* Algorithm 1 decodes a chromosome with the above representation into a valid control input. This algorithm operates in three stages: $(i)$ nodes selection, $(ii)$ load distribution, and $(iii)$ local search. First, it selects nodes as potential placement locations for each application (lines 3 to 5). For this, the first part of the chromosome ($C_{\text{I}}$) delimits the number of nodes to be selected on line 4. Then, the next line selects nodes with high values in the second part of the chromosome ($C_{\text{II}}$) as host candidates.

The second stage (lines 6 to 18) of Algorithm 1 is related to load distribution. It first orders all possible load sources according to the third part of the chromosome ($C_{\text{III}}$) on line 6. Following this order, it distributes the total loads $\Lambda_a^m(k)$ of a source in chunks $\lambda^*$ among the nodes selected in the first stage plus the cloud node. The cloud node addition ensures there are enough resources to deploy at least one replica of each application. Note that the chunk size $\lambda^* = \Lambda_a^m(k)\lambda_{\%}$ can be an algorithm input by setting parameter $\lambda_{\%} \in (0, 1]$. Line 10 orders the selected nodes by the estimated response time. Then, while there are still loads to be dispatched, the decoder searches in the sorted nodes for one with sufficient resources to receive an additional chunk of load. When the first envisioned target node is found on line 13, it sets to place a replica of the requested application on this node and assigns an additional chunk to the replica.

In the last stage, Algorithm 1 performs a local search around the placement decision from line 19 to 26. If the number of replicas deployed by previous stages exceeds the maximum allowed, it replaces surplus replicas with one on the cloud node. Otherwise, the decoder tries to place an application

**Algorithm 1:** Chromosome Decoder.

```
Data: $C = [C_{\text{I}}, C_{\text{II}}, C_{\text{III}}]$, $s(k)$, $e(k)$
Result: $c(k) = (\rho(k), \delta(k))$
$\rho_a^n(k)$, $\delta_a^{m,n}(k)$, $\lambda_a^n(k) \leftarrow 0$;
$\lambda_{\%} \leftarrow 0.25$;                                    // 25%
forall $a \in \mathcal{A}$ do
    $x \leftarrow \min(|\mathcal{V}|, \lceil C_{\text{I}}^a A_a^{max} \rceil)$;
    $V_a \leftarrow$ select $x$ nodes with higher $C_{\text{II}}^{a,n}$, $n \in \mathcal{V}$;
$P \leftarrow$ list of pairs $(a, m) \in \mathcal{A} \times \mathcal{V}$ sorted by $C_{\text{III}}^{a,m}$;
forall $(a, m) \in P$ do
    $l \leftarrow \Lambda_a^m(k); \lambda^* \leftarrow \Lambda_a^m(k)\lambda_{\%}$;
    $V'_a \leftarrow V_a \cup \{\text{cloud}\}$;
    sort nodes $n \in V'_a$ by $d_a^{m,n}(k+1)$ in (2);
    while $l > 0$ do
        forall $n \in V'_a$ do
            if $\lambda_a^n(k) + \lambda^*$ respects (11f) $\wedge$ (11g) then
                $\rho_a^n(k) \leftarrow 1$;
                $\delta_a^{m,n}(k) \leftarrow \delta_a^{m,n}(k) + \lambda^* / \Lambda_a^n(k)$;
                $\lambda_a^n(k) \leftarrow \lambda_a^n(k) + \lambda^*$;
                $l \leftarrow l - \lambda^*; \lambda^* \leftarrow \min\{l, \lambda^*\}$;
                break;
forall $a \in \mathcal{A}$ do
    $x \leftarrow A_a^{max} - \sum_{i \in \mathcal{V}} \rho_a^i(k)$;
    if $x > 0$ then
        replace $x + 1$ replicas of $a$ with one on cloud;
    else
        forall $n \in V_a$ do
            if $\lambda_a^n(k)$ respects (11f) and (11g) then
                $\rho_a^n(k) \leftarrow 1$;
```

replica on each node selected by the first stage. This former case allows the pre-deployment of an application that may be requested in the next time steps, avoiding future migration burden on response time.

*3) Initial Population:* It is composed of random-generated individuals, elite members from the previous time step, and individuals generated by the following heuristics:

- **Deadline**. One heuristic would be to prioritize requests for applications with shorter deadline requirements, which is encoded as:

$$C_{\text{I}}^a = 1,\, C_{\text{II}}^{a,n} = 0,\, C_{\text{III}}^{a,n} = 1 - \frac{A_a^{rd}}{\max_{i \in \mathcal{A}} A_i^{rd}}$$

- **Net Delay**. Another heuristic is to select nodes with the lowest network delay for all other nodes as candidates to host an application. We encode this heuristic as follows:

$$C_{\text{I}}^a = 1, C_{\text{II}}^{a,n} = 1 - \frac{\sum_{i \in \mathcal{V}} d_{net}^{a,i,n}(k+1)}{\max\limits_{j \in \mathcal{V}} \sum\limits_{i \in \mathcal{V}} d_{net}^{a,i,j}(k+1)}, C_{\text{III}}^{a,n} = 0$$

- **Combined Solution**. We can obtain a new individual by combining two or more solutions. For this, we add their chromosome vectors and divide the result by the number of added solutions.

**Algorithm 2:** Control Sequence Decoder.

```
Data: $\pi_C = \{C(t), \dots, C(t+H-1)\}$, $s(t)$, $e(t)$
Result: $\pi_c = \{c(t), \dots, c(t+H-1)\}$
forall $k \in \{t, t+1, \dots, t+H-1\}$ do
    $c(k) \leftarrow$ Algorithm 1 with $C(k), s(k), e(k)$;
    $s(k+1) \leftarrow \phi\,(s(k), c(k), e(k))$;
    $e(k+1) \leftarrow$ by a forecasting method;
```

### B. H-Steps Look Ahead Algorithms

Instead of directly establishing a control input sequence for a prediction horizon $H$, let $\pi_C = \{C(k) \mid k \in [t, t+H-1]\}$ be a sequence of chromosome vectors where $C(k)$ is the chromosome selected for time step $k$. Then, Algorithm 2 describes how we can obtain a control input sequence by applying the system dynamics of Section IV-B and Algorithm 1 in this $\pi_C$ sequence. Observe that a control input sequence generated by Algorithm 2 is a feasible solution for problem (11), as Algorithm 1 always decodes a chromosome to a control input that satisfies this problem constraints.

An evaluation of all possible $\pi_C$ sequences may present a similar computational complexity issue to the control input sequence case. Therefore, we propose two heuristics in the rest of this section to obtain chromosome sequences that are decoded to sub-optimal solutions for problem (11).

*1) Simple Sequence:* Given a chromosome vector $C$, this heuristic creates a simple sequence only containing this chromosome, i.e., $\pi_C = \{C(k) = C \mid k \in [t, t+H-1]\}$. In this way, we can adopt both the genetic algorithm and chromosome representation of Section V-A to solve (11), but, instead, using this simple sequence as an individual associated solution.

*2) General Sequence:* Given $|C|$ as the vector length of a chromosome presented in the previous subsection, we design a new chromosome representation $C'$ where $|C'| = H \times |C|$. Thus, we can obtain a sequence of former chromosome representation by splitting $C'$ into $H$ consecutive parts with length $|C|$. By using this new representation and Algorithm 2 as the decoder method, we can perform *BRKGA+NSGA-II* to find sub-optimal solutions for problem (11).

## VI. Performance Evaluation

### A. Evaluated Algorithms

In order to evaluate our proposed strategy the following algorithms are compared:

- ***Cloud***. It places all applications in the cloud node.
- ***N+D***. It is a combination of Net Delay and Deadline heuristics presented in Section V-A3.
- ***H1***. We apply the proposed one-time step algorithm when the prediction horizon $H = 1$.
- ***Static***. It performs the one-time step algorithm only in the first time step. Then, the resulted control decision is maintained almost without changes for the remaining time steps. Control changes happen when an application replica cannot handle a load increase due to a lack of resources, and then, this excess load is sent to the cloud.

TABLE I
PERFORMANCE EVALUATION PARAMETERS

| **Parameter** | **Value** |
|---|---|
| CPU (MIPS) | Cloud: $\infty$, Core: $2 \times 10^4$, BS: $10^4$ |
| DISK (GB) | Cloud: $\infty$, Core: 32, BS: 16 |
| RAM (GB) | Cloud: $\infty$, Core: 16, BS: 8 |
| Usage Cost for resource/second | $N_{n,r}^{cost}(x) = a \times 10^{-b}(x+1)$<br>$a =$ Cloud: 0.25, Core: 0.5, BS: 1<br>$b =$ CPU: 12, DISK: 18, RAM: 15 |
| User Proportion (%) | mMTC:70, eMBB:20, URLLC:10 |
| App. Proportion (%) | mMTC:34, eMBB:33, URLLC:33 |
| Bandwidth (Gbps) | BS↔BS: 0.1, BS↔Core: 1, Core↔Cloud: 10 |
| Propagation delay (ms) | BS↔BS: 1, BS↔Core: 1, Core↔Cloud: 10 |
| Max. Replicas $A_a^{max}$ | $[1, \lvert\mathcal{V}\rvert]$ |
| Deadline $A_a^{rd}$ (s) | mMTC:$[0.1, 1]$, eMBB:$[0.01, 0.1]$, URLLC:$[0.001, 0.01]$ |
| $\lambda_a$ (requests/s) | mMTC: $[0.1, 1]$, eMBB: $[1, 100]$, URLLC: $[1, 100]$ |
| Data $A_a^{data}$ (Kb) | mMTC: $[0.1, 1]$, eMBB: $[1, 10]$, URLLC: $[0.1, 1]$ |
| Work $A_a^{work}$ (CPU Instructions) | mMTC, URLLC: $[1, 5] \times 10^6$, eMBB: $[5, 10] \times 10^6$ |
| RAM, DISK Demand $A_a^r(\lambda) = b\lambda + c$ (MB) | $b =$ mMTC, URLLC: $[0.1, 1]$, eMBB:$[1, 10]$<br>$c =$ mMTC, URLLC: $[10, 100]$, eMBB: $[100, 1000]$ |
| CPU Demand (IPS) $A_a^r(\lambda) = b\lambda + c$ | $b = A_a^{work} + 1$, $c = \frac{A_a^{work}}{\alpha A_a^{rd}} + 1$,<br>$\alpha = [0.1, 0.5]$ |

- ***SS***. It is the proposed simple sequence heuristic with ARIMA as the forecasting method. We set $H = 2$, which allows pre-migrations exploration.
- ***GS***. Our proposed general sequence heuristic also using ARIMA as the forecasting method. We also set $H = 2$.

### *B. Performance Evaluation Metrics*

We define the following metrics as the optimization objectives upon which we evaluate the algorithms:

- **Deadline Violation**. An important metric is to keep an application response time below its deadline. Thus, we would like to minimize potential deadline violations. Eq. (14) expresses the deadline violation among all requests flow, where $[x]^+ = max(0, x)$. Each request flow contributes to this violation according to its request transmission rate as a weight. Furthermore, we select this metric as the priority objective described in Section IV-C.

$$\frac{\sum_{a \in \mathcal{A}} \sum_{m,n \in \mathcal{V}} [d_a^{m,n}(k+1) - A_a^{rd}]^+ \delta_a^{m,n}(k) \Lambda_a^m(k)}{\sum_{a \in \mathcal{A}} \sum_{m,n \in \mathcal{V}} \delta_a^{m,n}(k) \Lambda_a^m(k)} \tag{14}$$

- **Operational Cost**. Eq. (15) specifies the operational cost during a time step as the total cost of the resources allocation for all deployed application replicas.

$$\sum_{a \in \mathcal{A}} \sum_{n \in \mathcal{V}} \rho_a^n(k) T_s \sum_{r \in \mathcal{R}} N_{n,r}^{cost}\left(A_a^r\left(\lambda_a^n(k)\right)\right) \tag{15}$$

- **Migration Cost** is the ratio of application replicas migrated/replicated in the system at a time step, as shown in (16). The application state size is a weight in this metric as large applications may take longer to be transferred and consume more network resources.

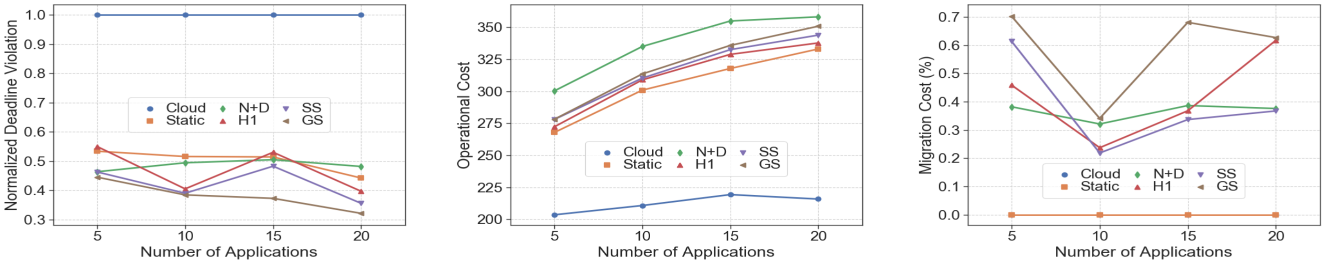
$$\frac{\sum_{a \in \mathcal{A}} \sum_{n \in \mathcal{V}} A_a^{D+R}\left(\lambda_a^n(k)\right) \rho_a^n(k) \left(1 - \rho_a^n(k-1)\right)}{\sum_{a \in \mathcal{A}} \sum_{n \in \mathcal{V}} A_a^{D+R}\left(\lambda_a^n(k)\right) \rho_a^n(k)} \tag{16}$$

### *C. Evaluation Setup*

We developed simulation experiments using Python to evaluate our proposed algorithms in a 5G network scenario with EC. In this scenario, we consider nine Base Stations (BSs), forming a $3 \times 3$ grid network. These BSs are also connected to a core node, which is connected to the cloud on the other side. All of these nodes (BSs, core, and cloud) have hosting capabilities, and their total resource capacities are lower as we descend from cloud to BSs. On the other hand, the resource allocation cost increases as nodes get closer to end-users due to resource scarcity. Besides, we assume that a node usage cost function $N_{n,r}^{cost}(\cdot)$ is linear.

We analyze the placement of the three types of applications specified for a 5G network [19], [20]. First, massive Machine Type Communications (mMTC) applications are characterized by low resource requirements, delay tolerance, and a quite large number of users. Then, enhanced Mobile Broadband (eMBB) applications have high resource demand, a medium deadline, and an intermediate number of users. Finally, Ultra Reliable Low Latency Communications (URLLC) applications have low resource usage, a strict deadline, and a small number of users. Based on these characteristics, we randomly assign the values for the application parameters. Note that for evaluation purposes, it is sufficient to use relatively different parameter values among diverse application types instead of applying more realistically accurate values. Moreover, application resource demands $A_a^r(\cdot)$ are considered to be linear functions. In the CPU demand case, the linear constants are selected based on queue stability and deadline requirements.

A test case of the evaluated scenario consists of 48 time steps with 30 minutes of duration each, totaling one day. In this way, it is possible to make, apply, and measure the result of a control decision in a single time step. Moreover, users generate requests with an unchanged average rate. However, the number of active users in the system can change over time, and consequently, varying the aggregate request arrival rate. Thus, we change the number of active users for each application in each node according to some workload patterns for cloud environments as described in [21]: Stable, Growing, Cycle/Bursting, and On-and-Off. A Stable workload is characterized by a constant number of requests per time unit. A growing pattern shows a load that increases due to, for instance, an application becoming popular. We also add the inverse of this pattern where a load decreases along the time. A Cyclic/Bursting may present regular periods or bursts of loads (e.g., daytime has more workload than nighttime). In the

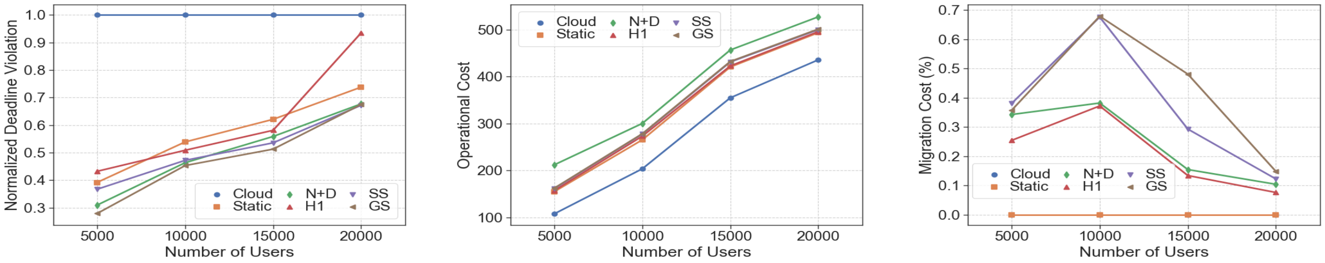


(a) Deadline Violation

(b) Operational Cost

(c) Migration Cost

Fig. 2. Average performance per time step with 10000 users.



(a) Deadline Violation

(b) Operational Cost

(c) Migration Cost

Fig. 3. Average performance per time step with 5 applications.

last pattern, On-and-Off, workloads are processed periodically or occasionally processed in batches.

The results shown in the following sub-section come from an average of 30 different random runs for each test case, and Table I summarizes the main evaluation parameters.

### *D. Results and Discussion*

Figures 2(a) and 3(a) show the normalized deadline violation per time step when increasing the number of applications and users, respectively. This normalization uses the *Cloud* results as the base. We can see in both figures that *GS* achieves lower violation levels than the other algorithms, and *SS* has better or similar results than *H1*, *Static*, and *N+D*. Moreover, *GS* reduces violations in Fig. 2(a) when there are more applications, but keeping the total number of users fixed. This drop can be an effect of having fewer users per application, especially for the URLLC type that is characterized by low user percentages and strict deadlines. On the other hand, violations rise when the number of users increases with a fixed number of applications in Fig. 3(a). This is caused by the growth of requests traffic to the remote cloud node when there is more competition for resources on the other nodes.

We observe an augmentation of operational costs by having more applications or users in Figs. 2(b) and 3(b), respectively. *Cloud* exhibits the lowest costs because only one replica of each application is placed in the system, and cloud resources are cheaper than in other locations. Meanwhile, *N+D* tends to place as much replicas as possible, having the highest cost. The other compared algorithms have similar costs.

Regarding migration costs in Figs. 2(c) and 3(c), *Cloud* and *Static* present no migration as expected. In both figures, *GS* has the highest costs, which is a trade-off of prioritizing deadline violation. In addition, migration costs for *H1*, *SS*, and *GS* initially decrease and then increase when there are more than ten applications in Fig. 2(c). This cost turning point happens because of two aspects that impact the volume of migration traffic: the total number of application replicas that increase with more applications, and their state size that shrinks when there are fewer users per application. In Fig. 3(a), migration costs fall when there are more than $10^4$ users for a fixed number of applications. As well as the rise of deadline violation in Fig. 3(a), this migration cost reduction is also a consequence of cloud traffic growth.

## VII. Conclusion and Future Works

In this paper, we studied a joint optimization of service placement and load distribution with dynamic request loads in EC. We then proposed a limited look-ahead prediction strategy to handle the impact of service migration on application response time while optimizing multiple performance-related objectives. In this strategy, we designed a genetic algorithm to solve the formulated problem for a single time step, and two extensions of this algorithm, *SS* and *GS*, when looking at more time steps in a prediction window. Evaluations showed that our proposed *SS* and *GS* algorithms outperform other benchmark algorithms in terms of deadline violations while having similar operational costs. A trade-off when our proposals prioritize the deadline violation minimization was the occurrence of more service migrations.

As for future work, we plan to study a distributed control strategy to make decisions more frequently in a scalable way.

# REFERENCES

[1] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan, “Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers,” *Computer Networks*, vol. 130, pp. 94 – 120, 2018.

[2] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, “All one needs to know about fog computing and related edge computing paradigms: A complete survey,” *Journal of Systems Architecture*, vol. 98, pp. 289 – 330, 2019.

[3] J. Gedeon, F. Brandherm, R. Egert, T. Grube, and M. Mühlhäuser, “What the fog? edge computing revisited: Promises, applications and future challenges,” *IEEE Access*, vol. 7, pp. 152 847–152 878, 2019.

[4] V. Farhadi, F. Mehmeti, T. He, T. L. Porta, H. Khamfroush, S. Wang, and K. S. Chan, “Service placement and request scheduling for data-intensive applications in edge clouds,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1279–1287.

[5] X. Yu, M. Guan, M. Liao, and X. Fan, “Pre-migration of vehicle to network services based on priority in mobile edge computing,” *IEEE Access*, vol. 7, pp. 3722–3730, 2019.

[6] B. Gao, Z. Zhou, F. Liu, and F. Xu, “Winning at the starting line: Joint network selection and service placement for mobile edge computing,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1459–1467.

[7] L. Gu, D. Zeng, S. Guo, A. Barnawi, and Y. Xiang, “Cost efficient resource management in fog computing supported medical cyber-physical system,” *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 1, pp. 108–119, 2017.

[8] L. Zhao and J. Liu, “Optimal placement of virtual machines for supporting multiple applications in mobile edge networks,” *IEEE Transactions on Vehicular Technology*, vol. 67, no. 7, pp. 6533–6545, 2018.

[9] X. Sun and N. Ansari, “Green cloudlet network: A sustainable platform for mobile cloud computing,” *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 180–192, 2020.

[10] T. Ouyang, Z. Zhou, and X. Chen, “Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[11] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leung, “Dynamic service migration and workload scheduling in edge-clouds,” *Performance Evaluation*, vol. 91, pp. 205 – 228, 2015, special Issue: Performance 2015.

[12] Y. Yu, T. Chiu, A. Pang, M. Chen, and J. Liu, “Virtual machine placement for backhaul traffic minimization in fog radio access networks,” in *2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–7.

[13] S. Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema, “Online control for self-management in computing systems,” in *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, 2004, pp. 368–375.

[14] N. Kandasamy, S. Abdelwahed, and M. Khandekar, “A hierarchical optimization framework for autonomic performance management of distributed computing systems,” in *26th IEEE International Conference on Distributed Computing Systems (ICDCS’06)*, 2006, pp. 9–9.

[15] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, “Optimized placement of scalable iot services in edge computing,” in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 189–197.

[16] ——, “A multi-objective service placement and load distribution in edge computing,” in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–7.

[17] J. F. Gonçalves and M. G. C. Resende, “Biased random-key genetic algorithms for combinatorial optimization,” *Journal of Heuristics*, vol. 17, no. 5, pp. 487–525, Oct 2011.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[19] N. Alliance, “5g white paper,” *Next generation mobile networks, white paper*, pp. 1–125, 2015.

[20] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel, A. Puschmann, A. Mitschele-Thiel, M. Muller, T. Elste, and M. Windisch, “Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture,” *IEEE Communications Magazine*, vol. 55, no. 2, pp. 70–78, 2017.

[21] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.