Proceedings Paper:
Canete, A, Djemame, K orcid.org/0000-0001-5811-5263, Amor, M et al. (2 more authors) (2022) A proactive energy-aware auto-scaling solution for edge-based infrastructures. In: Proceedings of the 15th IEEE/ACM International Conference on Utility and Cloud Computing. 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC2022), 06-09 Dec 2022, Vancouver, Washington, USA. IEEE , pp. 240-247. ISBN 978-1-6654-6087-3

https://doi.org/10.1109/UCC56403.2022.00044

A proactive energy-aware auto-scaling solution for edge-based infrastructures

Angel Cañete, Karim Djemame, Mercedes Amor, Lidia Fuentes, Abdullah Aljulayfi .

***Abstract*—Proactive auto-scaling mechanisms in edge-based infrastructures can anticipate user service requests by allocating computing resources while supporting the quality of service needed by a vast range of applications requiring, e.g., a low latency or response time. However, managing the dynamic needs of user service requests is challenging due to the edge infrastructure's heterogeneity and dynamic nature. Also, minimizing global energy consumption is a must in today's systems, which should be addressed inherently as part of any resource scaling solution. This paper presents a proactive horizontal auto-scaling framework for edge infrastructures, which takes into account both the base (idle) and dynamic (due to application execution) energy consumption of edge nodes, as well as of the node scaling mechanism. Simulations were performed with the EdgeCloudSim simulator with a workload provided by Shanghai Telecom and the results show up to a 92.5% decrease in energy consumption, a failed request rate of up to 0%, and reasonable execution times of the auto-scaling process for different problem sizes.**

***Index Terms*—energy efficiency, proactive auto-scaling, horizontal auto-scaling, Edge Computing, B5G**

I. Introduction

In a near future, B5G networks are expected to support a variety of vertical services demanding ultra-low latency and high data rates. It is the responsibility of the network infrastructure to efficiently allocate the available (and limited) resources to the service requests of mobile users meeting certain Quality of Service (QoS) requirements while accounting for their energy footprint cost.

Edge computing, which delivers and distributes computing and storage to nodes close to the end user, can help significantly reduce service response times and increase reliability, security, and sustainability compared to cloud computing. To achieve this, it is required efficient management of the infrastructure resources. Auto-scaling allows management and improves infrastructure resource availability while optimizing energy consumption when the services demand or workload varies dynamically. Horizontal auto-scaling, which enables increasing/decreasing the number of nodes of edge infrastructures, allows reducing the energy consumption (EC) of the nodes themselves, where non-IT appliances such as cooling and lighting can consume around 33%-52% of the total energy of an infrastructure of up to 500 nodes [1]. In addition, for dynamic workloads that require frequent scaling up and down, reactive auto-scaling solutions can be costly and consume a significant amount of time, resources and so energy. Alternatively, predictive methods can perform better in minimizing response time, satisfying service latency objectives, and reducing the auto-scaling interventions, as they can forecast future workload and anticipate resource provision or de-provision. Proactive auto-scaling solutions for Edge Computing systems are scarce. The heterogeneity of a limited set of computing resources makes proactive auto-scaling more complex and time-consuming than in cloud systems [2], [3]. In addition, the risk of failed requests can increase due to wrong workload predictions. Thus, the development of predictive auto-scaling policies under dynamic workloads involves important challenges when they are put into practice in edge infrastructures.

A. Cañete, M. Amor, and L. Fuentes are with the Grupo CAOSD, Universidad de Málaga, Campus de Teatinos, 29071 Málaga, Spain; ITIS Software, Universidad de Málaga, Spain. K. Djemame and A. Aljulayfi are with the Group DSS, University of Leeds, LS2 9JT Leeds, UK. E-mail: {angelcv, pinilla, lff}@lcc.uma.es, {K.Djemame, ml16afa}@leeds.ac.uk

This paper presents a proactive horizontal auto-scaling framework specially well suited for edge infrastructures with different configurations that address these challenges, being capable of reducing the EC of dynamic workloads in edge environments. This solution takes into account both the base (idle) and dynamic (due to application execution) EC of the nodes, as well as the EC of node scaling. The approach has been connected to the open source EdgeCloudSim edge environment simulator [4], which serves real dynamic edge workload provided by Shanghai Telecom [5], and tested in three different workload scenarios. Experimentation has been used to analyze, following the goal-question-metric methodology, the feasibility of using our auto-scaling approach to reduce the EC in edge-based infrastructures. The results show a decrease of up to 92.5% in EC, a failed request rate of up to 0%, and reasonable execution times of the auto-scaling process for different problem sizes.

The rest of the paper is structured as follows: Section II reviews the related work. The proposed orchestration and auto-scaling approach is presented in Section III. The pro-active auto-scaling module is detailed in Section IV, alongside the EC model and workload predictor, the latter being responsible for future workload forecast to be submitted to the edge infrastructure. Section V presents the research questions, outlines the experimental design that will address the research questions, demonstrates the feasibility of the proposed approach, and presents the experiment results with a discussion on their significance. It also reflects on the research outcomes and any limitations encountered. Finally, Section VI concludes with a summary of the research findings and suggestions for future work.

II. Related work

Although in the literature, a plethora of approaches addresses auto-scaling in cloud-based infrastructures to minimize EC (or its associated cost) [3], the feasibility of applying cloud-native auto-scaling techniques in Edge Computing environments has not been evaluated nor compared taking into account the heterogeneous nature presented in edge infrastructures [2], [3], [6]. The reason is that, unlike cloud servers, edge servers have limited computing resources due to their size and power constraints, making efficient resource management crucial and resource allocation a complex task. This phenomenon is aggravated when the tasks delegated to the edge require peripherals that only some of the devices have (e.g., a specific network card, or a GPU). Furthermore, users demand low latency, and then transferring workloads across edge servers inevitably incurs additional latency costs that conflict with edge computing's low-latency goal.

Few works address proactive auto-scaling in Edge Computing environments as this work, and most of them focus on the horizontal scaling of the number of instances (VMs or containers) according to a prediction of user requests and the surplus or deficit of resources used by deployed services (service scaling) [3]. In addition, most of them do not consider EC when deciding between different scaling options. Regarding resource scaling works in edge environments (among which is our proposal), in [6] authors provide a model to energy-efficiently attend to the edge resources demand. It is a proactive auto-scaling solution that uses a very simple EC model based on the consumption per unit of time and considers some node characteristics (such as switching EC) as homogeneous for the entire infrastructure. Authors report up to 57% decrease in EC when compared with other allocation techniques. Chen et al. [7] present EdgeDR, an online market mechanism to achieve cost efficiency in edge demand response programs. Using a primal-dual-based, polynomial-time approximation algorithm, it obtains a near-optimal cloudlets switching solution. The model used by the authors simplifies resource management by considering that services use a fixed percentage of the nodes' total resources, regardless of the node's capacities, and without distinguishing between CPU (Central Processing Unit), memory or disk. EdgeDR gets around 30% of energy saving compared with other similar mechanisms. Song et al. [8] present an online task scheduling algorithm for distributing workloads among clusters in a way to ensure the energy reduction goal of EDR. Authors declare up to 49.8% of energy saving with an acceptance rate of up to 95%. So, an adequate energy reduction goal in edge environments can be set in the range between 30% and 50%.

However, ***in contrast to previous similar approaches [6]–[8], our proposal*** *considers device capabilities and runtime impact independently, making it applicable to high heterogeneous infrastructures. Our approach also characterizes separately the workload requirements in terms of CPU, memory, disk, peripherals, etc., which can be considered separately for task allocation. Therefore, although these approaches attempt to optimize EC in edge environments, they fail to provide a flexible solution to dynamic workload scenarios (one of our contributions) as they simplify the problem by considering resources or workload requirements as homogeneous, or do not take into account the characteristics of the devices in the impact on the QoS.*

III. Our approach

This section presents our auto-scaling and placement approach to minimize EC.

Figure 1 shows a full overview of the proposed framework modules and processes. The user requests of applications/VNFs are served by an edge infrastructure. Applications and VNF are implemented as containerized workloads and services, and are contained in a repository (e.g., DockerHub). The edge infrastructure is composed of a set of nodes, which can be master or worker nodes. *Master nodes* (there may be more than one to avoid a single point of failure [9]) are responsible for assigning the workloads to the *Worker nodes*–master nodes can also be worker nodes at the same time. The *Master node* includes an orchestrator (e.g., Kubernetes [9]), a platform that facilitates the declarative configuration and automation of the deployment, management, coordination and availability of services, and the scaling of nodes.

Each user request requires instantiating an application/VNF, which is assigned on demand by the Energy-aware orchestrator to an edge node, where it is executed in a packaged form inside a container (e.g., Docker). To minimize the EC of the task execution, the default scheduler of the orchestrator (i.e., in the case of Kubernetes, *kube-scheduler*) has been modified to assign tasks/applications to the most energy-efficient nodes capable of running the application/VNF. This placement decision is performed with the help of the Proactive auto-scaling module, which can be running on a master node in the infrastructure or even in an external node (even in the cloud). Periodically, the *Master node* (or one of them) requests the Essential Node Identifier (ENI) component to perform the proactive horizontal auto-scaling process. This component receives the expected workload (number of requests) from the Workload predictor and the current state of the infrastructure and determines which nodes should keep active to serve user requests the next time interval (defined by the infrastructure administrator). It also accesses the application requirements (detailed in the Application repository (e.g., memory, disk, computational cost, peripherals, etc)–see Figure 1. The ENI component prioritizes the nodes with the lowest EC and the sufficient resources available to serve the expected workload while minimizing the number of active nodes (and therefore, EC). Once the *Master node* receives the information on the nodes to be kept active, it deactivates (puts into sleep mode) those that are not required.

IV. Proactive auto-scaling module

A. Energy consumption model

The energy consumption model is used by the auto-scaling module to estimate two types of EC: dynamic EC, which depends on the workload of nodes and idle EC, which is the EC of a node for remaining active [10]. Our solution decreases the

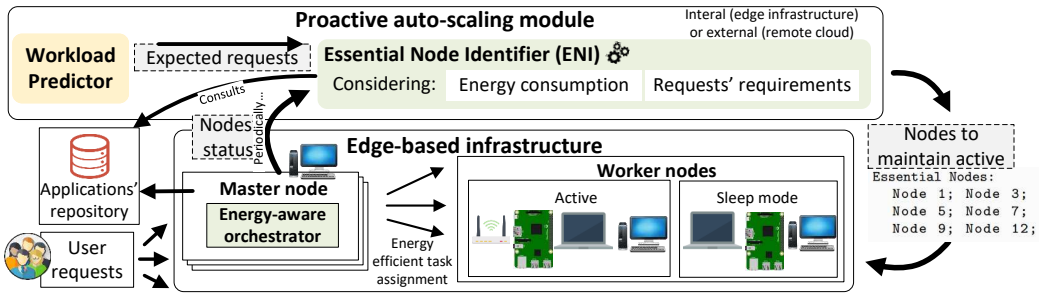

Fig. 1: Full overview of our approach.

global EC at two different points: at the scheduler, prioritizing the allocation of computational load to energy-efficient nodes (reducing the dynamic EC); and at the auto-scaling process, putting on sleep mode those nodes that are considered non-essential (decreasing the idle EC).

In dynamic EC, the model associates an EC to computation and communication separately. The computational EC is influenced by several factors, such as the usage of CPU, storage, and RAM (Random Access Memory), being CPU usage the most influential factor and the one in which most Edge Computing approaches base their EC models [10].

Equation bellow contains the expressions used to calculate the EC (J) associated to computation ($eComp$), as well as the EC for remaining nodes active ($eIdle$), for being inactive ($eSleep$), for switching on and off nodes ($eSwitch$) [7], and for data sending and receiving ($eSend$ and $eRcpt$) [10], [11]:

$$eComp_{n,i} = (1-\alpha_n)eMax_n v_i \frac{w_i}{CPU_n} ew_n + eDeploy_n$$

$$eIdle_n = \alpha_n eMax_n ew_n t$$

$$eSleep_n = \beta_n eMax_n ew_n t$$

$$eSwitch_n = s_n eSOn_n$$

$$eSend_{dataUp_i,n} = P_n^{Tx} \frac{(1+pR)dataUp_i}{R_n^{Tx}} ew_n$$

$$eRcpt_{dataDown_i,n} = P_n^{Rx} \frac{(1+pR)dataDown_i}{R_n^{Rx}} ew_n$$

where α_n, whose value is between 0 and 1, represents the fraction of the idle EC (e.g., 40%) of node n [10]; $eMax_n$ is the EC of node n when it is fully-utilized (in terms of CPU); v_n its CPU utilization ratio (0-1); w_i the CPU instructions required to compute task i (obtained on practice using tools like *perf* [12]); CPU_n the CPU frequency of node n. This model is based on the observation that the EC is linear to the CPU utilization ratio which depends on the computation load [11]. $eDeploy_n$ is the (fixed) amount of energy (J) required by node n to create the container of the task, and t is the time (s). β_n, which is part of the expression that denotes the EC of a node when sleeping, is the fraction of the sleep EC of node n. Regarding switching EC, s_n is 1 if the node was sleeping before the switching, 0 on the contrary, being $eSOn_n$ the EC for node switching on [7]. $eSend$ and $eRcpt$ denote the EC for data sending and receiving (the amounts of data (bits) $dataUp$ and $dataDown$) respectively–being pR the probability of retransmission due to packet loss. P^{Tx} and P^{Rx} represent the transmission powers, while R^{Tx} and R^{Rx} represent the data sending and receiving transmission rates [10]. The variable ew (energy weight), presented in all the expressions, allows the selection of the priority of saving energy in each node separately. Thus, if the goal is to focus the EC reduction on some specific devices, this variable is set to 1 for those devices and 0 for the others. This weight-based EC model makes more flexible the deployment and eases the definition of policies according to the infrastructure necessities. An example of practical use would be to prioritize the execution in devices powered by solar panels, thus reducing the carbon footprint.

B. Workload Predictor

The workload predictor component is responsible for forecasting the future workload based on historical data. This component is implemented within the *Context-aware Prediction Framework* presented in [13], [14] and integrated as part of our solution. This framework applies different machine learning algorithms according to the context: Linear Regression, Support Vector Regression, and Neural Networks. The context is identified by the Context Analyzer according to the workload pattern (i.e. decreasing, increasing, and fluctuating workload); the Selector Algorithm decides to select one of the machine learning algorithms that provides the best prediction of the future workload according to the current workload pattern. The main benefits of using this prediction framework are: (i) it has been shown to fit at least 10% better than other approaches; (ii) it adapts its mode of operation to the changing context of the type of system; and, (iii) has been tested with real workload [14]. Nevertheless, our approach is flexible enough to allow other predictive models, simply replacing the implementation of the workload predictor module component with another one.

C. Essential Node Identifier (ENI)

The Essential Node Identifier (ENI) module is responsible for deciding the nodes that must be kept active to satisfy the expected workload and minimize the EC. For this purpose, it uses the information on the predicted workload received from the workload predictor, and the infrastructure's current status (see Figure 1). Using the energy model presented in Section IV-A, it calculates the allocation of resources needed to meet the expected workload. This module has two different operation modes (OM), more and less resource-preserving. In the first one (OM1), the ENI module can determine the deactivation of nodes that are executing a task at the time when the auto-scaling process is being performed if deemed appropriate. In this case, nodes will be put into sleep mode after the ongoing execution. While this is, a priori, the most energy-efficient

Module 1 Essential Node Identifier (ENI)

Data: Nodes; currentStatus; expectedWorkload (expWl); autoscalingInterval; operationMode (OM1/OM2)
Result: actN

1 Nodes ← currentStatus
2 neededResources = getAssociatedResources (expWl)
`// Parameters to optimize:`
3 activeNodes (actN); computationEnergyConsumption (compEC); transmissionEnergyConsumption (sendEC); receptionEnergyConsumption (rcptEC); idleEnergyConsumption (idleEC); switchingEnergyConsumption (switchEC); expectedWorkload (expWl)
`// Acronyms: freeResources (freeR); dataToUpload (dataUp); dataToDownload (dataDown)`
`// Constrains:`
4 sol = Optimize()
5 sol.addConst (aN ⊂ Nodes)
6 sol.addConst(activeResources == $\sum_{n \in aN}\ n_{freeR}$)
7 sol.addConst(neededResources ≤ activeResources)
8 **if** *operationMode == OM2* **then**
9 | sol.addConst($\forall n \in actN$: **if** (n.isworking) **then**: n ∈ actN)
10 **end**
11 sol.addConst(compEC == $\sum_{n \in aN}$ eComp(n,expWl)
12 sol.addConst(sendEC == $\sum_{n \in actN}$ eSend(n_{RTx}, n_{pTx},avg($expWl_{dataUp}$)
13 sol.addConst(rcptEC == $\sum_{n \in actN}$ eRcpt(n_{RRx}, n_{pRx}, n_{freeR},avg($expWl_{dataDown}$)
14 sol.addConst(idleEC == $\sum_{n \in actN}$ eIdle(n_{eIdle}, autoscalingInterval)
15 sol.addConst(sleepEC == $\sum_{n \in (N \not\subset actN)}$ eSleep(n_{eSleep}, autoscalingInterval)
16 sol.addConst(switchEC == $\sum_{n \in N}$ eSwitch(n_{eSOn}, n_{eSOff}, n.isworking, $n \in N$))
17 sol.minimize(compEC + sendEC + rcptEC + idleEC + sleepEC + switchEC);
18 satisfiable = sol.checkSatisfability() `// Checking the satisfacibility`
19 **if** *satisfiable* **then**
20 | result = sol.solve()
21 **end**
22 **return(result.actN)**

solution as just the needed resources are active, nodes take some time to become active after being deactivated. Therefore, it is possible to receive requests that demand resources that are being activated, resulting in failed requests. To maximize the number of successful requests, we define a second operation mode (OM2) that is more resource-preserving than the previous one. OM2 keeps active the nodes that are running an application at the time the auto-scaling is performed.

Module 1 contains the ENI's pseudo-code, which is posed as a constraint satisfaction problem. First, the module updates the nodes' information with the current status and gets the needed resources from the expected workload received as input. The constraints that form the problem do the following: assure that the set aN is a subset of $Nodes$ (line 5); include the resources of the active nodes in the $activeResources$ set (line 6), and check that the resources included in $activeResources$ are enough to meet the expected workload (line 7); lines 8 to 10 include in the solution the nodes that are currently executing any other delegated task (OM2); line 11 calculates the computation EC, while lines 12-13 estimate the communication EC (sending and receiving respectively); line 14 calculates the idle EC (using the auto-scaling interval received as an input); line 15 estimates the sleep EC, while line 16 calculates the switching EC; line 17 minimizes the EC; line 18 checks the problem satisfiability, while lines 19-21 ask for the solution if so. Finally, line 22 returns the solution.

The ENI module is solved using Z3, an SMT (Satisfiability modulo theories) solver [15]. Thus, (1) the algorithm always returns a solution (unlike heuristic-based algorithms), which guarantees that the deployment is feasible or the lack of infrastructure's resources if no solution is found (maintaining all nodes active in this case); and (2) a large number of constraints (required to solve the problem at hand) helps SMT-solvers to reduce the search space and to find the solution faster [15]. Unlike solvers that use Integer Linear Programming or Mixed Integer Linear/Nonlinear Programming [10], SMT solvers neither restrict the variable types involved in the problem formulation nor the degree of the equations [16]. Additionally, it is proved that Z3, and concretely its optimization module (νZ), returns optimal solutions [15]. Nevertheless, the flexibility of our approach may allow the use of any other mathematical model for optimization.

V. Evaluation

This section analyses the feasibility of our approach in a simulated edge-based infrastructure that serves real dynamic edge workload.

A. Research questions

The methodology used in this study is the goal-question-metric approach as follows: *"Analyze the feasibility of using our proactive auto-scaling approach to reduce the EC in edge-based infrastructures"*. To achieve this objective, we set the following research questions (RQs):

RQ1: *To what extent can our proposal reduce the energy consumption in edge infrastructures?* This question will reveal whether our proposal results in a significant decrease in EC.

RQ2: *Which policy performs best in decreasing energy consumption and minimizing the number of failed requests?* This question is an extension of the previous one, but having in mind that deactivating resources may result in failed requests due to a lack of available resources.

RQ3: *Is our approach fast enough to be feasible to be used at runtime?* This question evaluates the applicability of auto-scaling at runtime in terms of the execution time of the modules implemented. Since the critical point in terms of execution time is the ENI module, the evaluation will focus on this module for different problem sizes and operation modes.

RQ4: *How important is it to adjust the mode of operation at runtime? In which cases does it make sense?* This will reveal when the system can be fed back with information from current deployments to modify its mode of operation.

B. Experimental setup

To answer the RQs, we have constructed an IoT scenario using the EdgeCloudSim simulator [4]. EdgeCloudSim is an open-source tool that provides a simulation environment specific to Edge Computing. This simulator has been extended to consider the nodes' EC during the simulation, returning the amount of energy consumed by each edge device at the end of the simulation using the EC model presented in Section IV-A. To ensure its correct operation the core of the simulator remains intact and extensive validation work has been performed.

The Shanghai Telcom dataset [5], reported in [17]–[19] has been used to simulate the edge workload. It contains six months of mobile phone records accessing the Internet via base

stations which are distributed over Shanghai city (i.e. more than 7.2 million records from 9481 mobile devices and 3233 base stations).

A previous data analysis [13] revealed that the workload of June had the lowest percentage of records with missing data and that the second day is representative of the rest days of June (overall workload is periodic). In the same work, it is observed a different trend in three different periods of June, 2: (1) a period in which the number of requests increase over time, which includes late night and early morning; (2) an interval of time in which the number of requests decreases over time (morning); and (3) a period in which the number of requests fluctuates, which includes afternoon to evening. Thus, we will apply our proposal to these three scenarios to evaluate their behaviour in different workload contexts. As in [13], we also select one hour from each period: the 2nd, 12th, and 14th hours simulating 14 minutes of each one that contains 107, 276, and 270 requests respectively. The user requests will ask for the service provided by eight different applications, with different requirements in terms of CPU, memory, disk, peripherals, and data to transmit and receive. Applications are equally likely to be requested by users. Regarding the number of IoT devices (i.e., users) and requests, these values are specified for each workload pattern according to the number of devices and requests in the dataset. Experiments consider an infrastructure of 20 edge devices with randomized characteristics. Their maximal EC is between 20 and 300 Watts; the idle EC (α) between 20 and 50%, P^{Tx} and P^{Rx} between 1-3 Watts; the sleep EC (β) between 0.01 and 0.5; the switching EC is between 10-50 Joules; the deployment EC between 0.5-1 Joules; the instructions per second of the CPU between 100000 and 300000 million; ew is 1 in all cases; the disk's capacity between 200-1000 Gb; and their RAM's capacity between 8-32 Gb [10], [11]. All nodes can be deactivated. This randomization takes into account that the most CPU-powerful nodes may also be the most energy-intensive, as the CPU frequency is directly related to the EC [11], [20]. In the same way, the applications' requirements (CPU, RAM, disk, and data to send and receive) have been also randomized. Experiments have been performed 30 times on one thread of an AMD Ryzen 7 1700X processor, and a subsequent exhaustive statistical study of the results has been carried out[1].

C. Reduction in the energy consumption

This section evaluates to what extent our proposal reduces the EC and its impact on the number of failed requests.

1) Dynamic energy consumption: Sometimes the nodes are shared with other applications and users, and they cannot go into sleep mode. In this case, the only way to reduce EC is by assigning applications to the most energy-efficient nodes. Thus, in this section, we focus on the reduction of (dynamic) EC obtained through our new orchestrator policy, *Green fit*. This policy compares the EC of running applications and selects, from among the nodes that meet the application requirements, with sufficient resources available and located within the user's reach, the one that consumes less energy. We compare the EC obtained with *Green fit* with the obtained with the policy *Best fit* [21], which selects the node with the most available resources; *First fit*, which assigns the application to the first node that meets the application requirements; *Random fit* (also considered in other works [6]), that selects a random node that accomplishes the application's requirements; and *Fastest fit*, that searches for the most powerful node in terms of CPU capable of running the application.

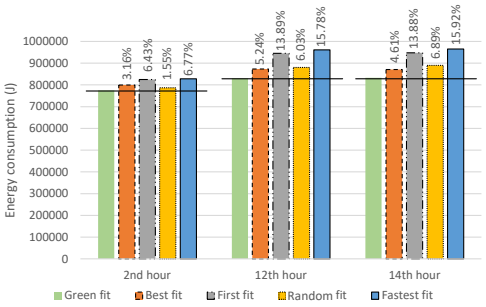

Fig. 2: EC for each orchestration policy.

Figure 2 shows the results in terms of EC for the three periods of time considered and compares them with *Green fit*. In all cases, our policy obtains the least EC (dynamic and idle EC are considered). Specifically, experiments show up to 15.9% of reduction in the EC (14th hour) when compared with *Fastest fit*. As expected, the reduction in the EC is major for the 12th and 14th hours (the ones with more requests), since we focus on the dynamic EC and it is directly related to workload. Regarding the execution time, predictably *Fastest fit* obtains the lowest service time on average, having *Green fit* and *Random fit* similar service times on average. Note that all the assignments accomplish the applications' requirements in terms of QoS. The percentage of failed requests is 0% in all cases, having the infrastructure resources enough to allocate the user requests.

2) Dynamic and idle energy consumption: For the same infrastructure, applications, and periods we apply our auto-scaling approach. The auto-scaling interval has been set in one minute, and the orchestration policy used is *Green fit*.

Some predictive models provide just the number of expected requests, not the applications demanded–as is our case [5], [13]. As the resources needed to meet these requests will depend on the demanded applications, we have elaborated four different resource reservation policies. If we suppose that we expect 10 requests in the next period and we handle 8 applications with a uniform probability of being requested, each application would be requested 1.25 times, which is not feasible. Then we consider that each application is demanded once while two requests are uncertain. On this basis, we define four assignment policies: *Random*, which would assign the two remaining requests randomly; *Oversizing*, which would consider that each application is demanded twice; and *Most/Least resource demanding*, which would assign these two uncertain requests to the most/least resource-demanded task. Note that our algorithm gives a solution for the worst-case scenario, in which the entire workload arrives at once right after performing the auto-scaling.

[1]Results spreadsheet available at: https://doi.org/10.5281/zenodo.7248106

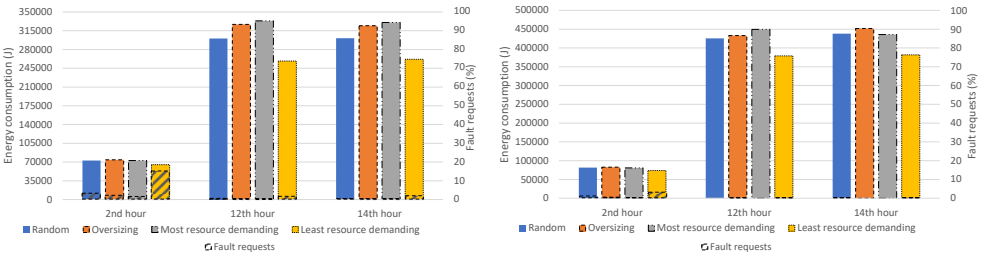

Fig. 3: Energy consumption (left y-axis) and percentage of failed requests (right y-axis) applying our auto-scaling approach for OM1 (left) and OM2 (right).

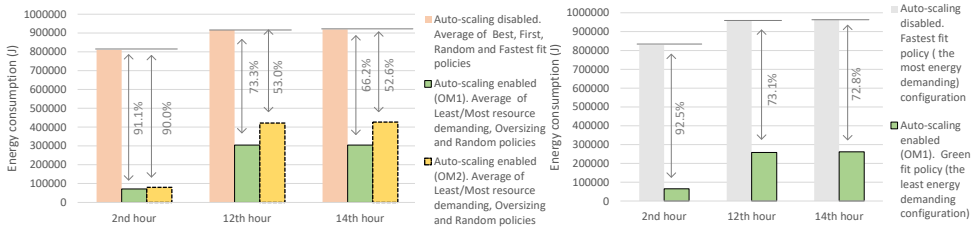

Fig. 4: Comparison of average EC with and without auto-scaling (left), and EC reduction in the best case (right).

Figure 3 shows the average EC (left y-axis) and percentage of failed requests (right y-axis) obtained with the different resource reservation policies and operation modes. As shown in Figure 3 and as expected, the *Least resource demanding* policy is the one that obtains the highest EC reductions. Regarding the number of failed requests, in OM1 (left side of Figure 3), the second hour is the period with the most failed requests, with 1.5% in the best case (*Most and Least resource demanding* policies respectively). Experiments show a 24.8% decrease on average in the failed requests between using *Most resource demanding* policy compared with *Oversizing*. *Least resource demanding* is the policy with the highest number of failed requests–and better EC. Nevertheless, in the 12th and 14th hours the *Least resource demanding* policy obtains an affordable 1.5% (12th hour) and 1.8% (14th hour) of failed requests, with a decrease in the EC of 14% and 13% respectively when compared with *Random* policy (the second best in terms of EC). This means that this policy could be a good choice in some scenarios. Regarding OM2 (right side of Figure 3), the percentage of failed requests is 0% or almost 0% in most cases, being 3% in the worst case (2nd hour and *Least resource demanding* policy). Therefore, the results in terms of requests accepted are better in some cases than in other proposals [3], [8]. Regarding execution time, all resource reservation policies have obtained similar times, since they use the same allocation policy (*Green fit*).

Comparing the EC with and without auto-scaling, experiments report an average reduction in the EC of 91.1%, 73.3%, and 66.2% in the 2nd, 12th, and 14th hours in the case of OM1, and 90%, 53% and 52.6% respectively for OM2 (left side of Figure 4). In the case of OM2, the EC increases by 26% on average in comparison with OM1. The greatest reduction in EC is obtained by OM1 in the *Least resource demanding* policy, which achieves 92.5% energy savings when compared with *Fastest fit* (right side of Figure 4), with better performance than other works (and applied to a more heterogeneous infrastructure) [6]–[8] (see Section II).

D. Execution latency

The time it takes for nodes to process a request depends on the nodes' capabilities. This section evaluates the response times for the different modes of operation of the auto-scaling system. Note that the allocation in our proposal ensures the requirements in terms of QoS (including response time) in all cases. Thus, on average, the time required to complete a request with and without auto-scaling is very similar. Specifically, experiments reveal that OM1 increases on average by 3.2% the response time in the 2nd hour, while reducing it by 2.1% and 2.7% for the 12th and 14th hours. In the case of OM2, the response time increases by 2.8% in the 2nd hour, decreasing by 2.8% and 2.6% in the 12th and 14th hours. Note that using more energy-efficient nodes is not necessarily subject to an increase in execution time, being possible to minimize consumption and latency at the same time [16].

E. Scalability

The time needed by the ENI to provide a solution may vary according to the problem size, which may compromise the applicability of the proposal. Thus, this section evaluates the ENI's execution time for different problem sizes. With this purpose, we develop a *Benchmark* version of our module, which allows setting the number of nodes and expected requests. This Section addresses RQ4.

A preliminary study revealed that, for infrastructures of less than 20 nodes, the ENI returns the solution almost instantly. Thus, the number of nodes has been set at 20 and 30 respectively, while the number of expected requests has been increased to the limit of infrastructure resources: up to 330 requests in the case of 20 nodes and up to 410 for an infrastructure of 30 nodes. The characteristics of the nodes (CPU, RAM, storage, free resources, $eMax$, α, peripherals, etc) and requests expected (requirements in terms of CPU, RAM,

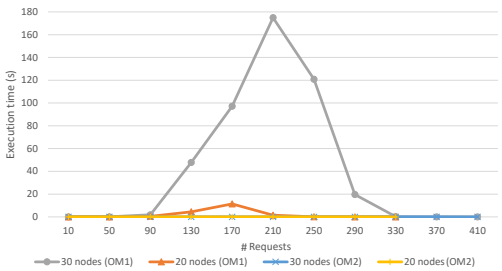

Fig. 5: ENI's execution time according to the problem size.

disk, data to transmit/receive, etc) have been randomized in each experiment.

Figure 5 shows the results of the experiments, in which we consider both operation modes of our solution.Regarding OM1, for an infrastructure formed by 20 nodes, the ENI has required about 11 seconds in the worst case (170 requests), being just 0.4 seconds in the case of 90 expected requests. This time decreases to 1.5 seconds when 210 requests are expected, and to 0.02 when 290. The infrastructure's capability is not enough when the expected number of requests increases up to 330, returning that the problem is not satisfiable in about 0.007 seconds. In the case of an infrastructure of 30 edge nodes, the execution time is 0.1, 0.3, and 1.8 seconds when 10, 50, and 90 requests are expected. This time grows up to 175 seconds in the worst case (210 requests, about the 50% of the infrastructure's capabilities). Again, this time decreases when the request requirements are greater than half the infrastructure capacity, being 0.24, 0.009, and 0.01 seconds when 330, 370, and 410 requests are expected.

In OM2, the number of nodes working (i.e., nodes that are executing other tasks when auto-scaling is performed) and their free resources have been randomly generated. To provide a realistic scenario, we consider that the number of nodes working is the number necessary to allocate 25% of the previous workload (assumed like the current one). This time, the ENI returns a solution almost instantaneously in most cases, requiring around 0.02 seconds in the worst case (30 nodes and 170 requests expected). This is because, once a node is considered part of the solution, the ENI module will assign it as many tasks as possible to reduce the idle EC. Thus, OM2 helps the ENI to reduce the search space.

Experiments reveal that the execution time is higher when the amount of resources requested is about half of the nodes' capabilities (see Figure 5). This is because as the number of tasks demanded increases, so does the number of devices involved in the solution and, therefore, the search space. The peak of this phenomenon is found when half of the infrastructure resources are demanded, decreasing gradually until all the resources are needed. Experiments also reveal that the execution time highly depends on the requests' requirements and nodes' capabilities, finding a big difference between executions of the same problem size. While the case of an infrastructure of 20 nodes and 170 requests expected requires about 11 seconds on average, in some executions the time needed dropped to 0.4 seconds. That means that the applicability of our solution in OM1 must be studied for the target infrastructure and expected requests. Regarding the difference between operation modes, experiments reveal that OM2 helps the module to return the solution faster.

F. Answers to research questions

This section uses the results in Section V to answer the RQs presented in Section V-A.

RQ1: *To what extent can our proposal reduce the energy consumption in edge infrastructures?* When the nodes of the infrastructure are shared and can not be put in sleep mode, we have obtained up to 15.9% of reduction in the EC (see Figure 2). In this scenario, the more requests are received, the more energy is saved (compared with other policies). The application of the auto-scaling approach reduces the EC by about 44% in the worst case, and up to 92.5% when using the OM1 and the *Least resource demanding* policy. The EC reduction is due to the time the nodes remain in sleep mode; the lower the workload, the longer the downtime and the EC reduction. The results show that our proposal has obtained a greater reduction in EC than other works [6]–[8] (see Section II). This energy savings is even greater when taking into account non-IT appliances (e.g., cooling and lighting), which can consume around 33%-52% of the total energy of an infrastructure of up to 500 nodes [1].

RQ2: *Which policy performs best in decreasing energy consumption and minimizing the number of failed requests?* The greatest EC reduction has been obtained using OM1 and the *Least resource demanding* policy (up to 92.5%). Although the percentage of failed requests for one of the periods evaluated (2nd hour) is high, the failed requests in the other periods (1.5% and 1.8%) are affordable, being the one that works best for such periods. Regarding the first period (2nd hour), the policy that has worked best is the *Most resource demanding*, with just 1.5% of failed requests and 92.5% of reduction in the EC. When the service must have high availability, the ENI's resource-preservative operation mode (OM2) obtains 0% or almost 0% failed requests in most cases. Compared to other proposals [3], [8], our auto-scaling module can obtain better results in request acceptance rate.

RQ3: *Is our approach fast enough to be feasible to be used at runtime?* Experimentation (see Figure 5) shows that the ENI execution time is longer when the services requested require about half of the infrastructure capacity. The applicability of ENI in OM1 will depend on the auto-scaling interval, as the execution time should always be less than this. For 20-node infrastructures, the average execution time obtained in the worst case is 11 seconds. Since it does not make sense to perform auto-scaling with a higher frequency, we can state that the ENI in OM1 is usable for infrastructures of 20 nodes or less. For larger infrastructures, the execution time may be higher than the auto-scaling frequency, compromising the feasibility (although there are large differences in execution time depending on the application's and infrastructure's characteristics). In this case, it is possible to decrease the auto-scaling interval, thus reducing the expected workload. This can be done by the auto-scaling

system automatically, if detects that the execution time is too long. Another solution would be to consider the infrastructure as two separate infrastructures and divide the workload between them. In any case, experiments reveal that the second operation mode (OM2) reduces the search space and helps the algorithm to find a solution almost instantaneously.

RQ4: *How important is it to adjust the mode of operation at runtime? In which cases does it make sense?* The experiments revealed that, when the number of failed requests is too high, the *Least resource demanding* resource-allocation policy achieves the greatest reduction in EC with acceptable failed response rates. Nevertheless, in some cases, the number of failed requests may be excessive. In such cases, modifying the resource reservation policy when such a pattern is detected (e.g., to *Most resource demanding* or *Oversizing*, see Figure 3), would solve the issue. This illustrates the importance of providing an auto-scaling solution with different modes of operation so that its behaviour can be adapted to suit the changing context of dynamic workloads. Concerning the auto-scaling's execution time, the feasibility of using the ENI module in OM1 for infrastructures of 20 nodes or less has been demonstrated in Section V-E. However, for larger infrastructures or when the characteristics of the workload make the auto-scaling process difficult, it is possible to modify the operation mode of the algorithm at runtime. As discussed in RQ3, another solution would be to increase the auto-scaling frequency (thus decreasing the expected workload), which can be easily done at runtime.

VI. Conclusions and future work

Edge Computing is becoming a key technology for B5G, achieving significant reductions in service response times and increased reliability, security and sustainability compared with traditional cloud-based systems. Given the limited resources in edge infrastructures, efficient resource allocation is essential.

Auto-scaling can help to achieve an efficient resource allocation, particularly for edge-based infrastructures [3]. This work provides a horizontal auto-scaling solution for edge infrastructures with different operation modes to reduce energy consumption, applicable to different contexts, and adaptable to the needs of the infrastructure/service. Its effectiveness and applicability are validated by applying it to a simulated edge infrastructure that serves the real edge workload provided by Shanghai Telecom in three different scenarios: increasing, decreasing and fluctuating workload. Using the goal-question-metric approach, the results are used to evaluate the impact on energy consumption, how energy awareness affects auto-scaling effectiveness (in terms of the number of failed requests), and how the energy-efficient proactive scaling module behaves to different problem sizes and operation modes. The results show up to a 92.5% decrease in energy consumption, a failed request rate of up to 0%, and reasonable execution times of the auto-scaling process for different problem sizes. Therefore, our proposal has achieved a higher energy saving than other proposals [6]–[8] (although this value is indicative since they have been applied to different datasets and infrastructures), with a wider accepted request rate in some operation modes [3], [8].

In future work, we plan to provide a self-adaptive solution to the auto-scaling framework, so that it changes its mode of operation according to the context.

Acknowledgments

This work is supported by the European Union's H2020 research and innovation program under grant agreement DAEMON 101017109 and by the projects co-financed by FEDER funds LEIA UMA18-FEDERJA-15, MEDEA RTI2018-099213-B-I00 (MCI/AEI) and RHEA P18-FR-1081.

References

[1] Ganeshalingam et al., "Shining a light on small data centers in the u.s." 2017.
[2] Chen et al., "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Comput. Surv.*, vol. 51, no. 3, jun 2018.
[3] Aslanpour et al., "Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research," *Internet of Things*, vol. 12, p. 100273, 2020.
[4] Sonmez et al., "Edgecloudsim: An environment for performance evaluation of edge computing systems," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, p. e3493.
[5] S. Telecom. (2018) The telecom dataset. [Online]. Available: http://sguangwang.com/TelecomDataset.html
[6] Cuiet et al., "Demand response in noma-based mobile edge computing: A two-phase game-theoretical approach," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.
[7] Chen et al., "Edgedr: An online mechanism design for demand response in edge clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 343–358, 2022.
[8] Song et al., "Edge emergency demand response control via scheduling in cloudlet cluster," in *IEEE INFOCOM WKSHPS*, 2020, pp. 394–399.
[9] "Kubernetes documentation: Production-grade container orchestration," https://kubernetes.io/docs/home/, online; accessed October 25, 2022.
[10] Mao et al., "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.
[11] Dinh et al., "Offloading in mobile edge computing: Task allocation and computational frequency scaling," *IEEE Transactions on Communications*, vol. 65, no. 8, pp. 3571–3584, 2017.
[12] "Perf events," https://perf.wiki.kernel.org/, online; accessed 25 October, 2022.
[13] A. F. Aljulayfi and K. Djemame, "A machine learning based context-aware prediction framework for edge computing environments," in *CLOSER2021*, 2021, pp. 143–150.
[14] Muhammad et al., "Predictive autoscaling of microservices hosted in fog microdata center," *IEEE Systems Journal*, vol. 15, no. 1, pp. 1275–1286, 2021.
[15] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," ser. TACAS/ETAPS, 2008, pp. 337–340.
[16] Cañete et al., "Supporting iot applications deployment on edge-based infrastructures using multi-layer feature models," *Journal of Systems and Software*, vol. 183, p. 111086, 2022.
[17] Li et al., "Profit-aware edge server placement," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 55–67, 2022.
[18] Guo et al., "User allocation-aware edge cloud placement in mobile edge computing," *Software: Practice and Experience*, vol. 50, no. 5, pp. 489–502, 2020.
[19] Wang et al., "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2021.
[20] Cañete et al., "Energy-efficient deployment of iot applications in edge-based infrastructures: A software product line approach," *IEEE Internet of Things Journal*, vol. 8, no. 22, pp. 16 427–16 439, 2021.
[21] Lai et al., "Cost-effective app user allocation in an edge computing environment," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.