AHP4HPA: An AHP-based Autoscaling Framework for Kubernetes Clusters at the Network Edge

Ioannis Dimolitsas, Dimitrios Spatharakis, Dimitrios Dechouniotis, Symeon Papavassiliou
School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece
{jdimol, dspatharakis, ddechou}@netmode.ntua.gr, papavass@mail.ntua.gr

***Abstract*—Autoscaling resources in a power-efficient way is essential to enable Green Computing resource management solutions. The development of dynamic resource provisioning techniques could lead to the minimization of power consumption and simultaneously guarantee high quality of service (QoS) inline with the workload demand. In this work, we introduce AHP4HPA, an autoscaling framework for Kubernetes Clusters, which is aligned with the Kubernetes architecture and state-of-the-art practices. We define resource profiles, namely a mapping between the QoS and the computing resources, to maximize the performance. Furthermore, Analytic Hierarchy Process (AHP) is exploited to dictate the scaling decision of the resources under various Key Performance Indicators (KPIs) toward power optimization of the allocated resources. To guarantee maximum performance of the deployed image classification application, an ARIMA model is dedicated to providing predictions regarding the incoming workload traffic. The framework is evaluated against a realistic dataset in a small-scale testbed. Numerical results indicate at least a 9% reduction of the average energy consumption when compared to other state of the art techniques.**

***Index Terms*—Edge Computing, Autoscaling, Multi-Criteria Decision Making, Power Efficiency, Kubernetes.**

I. Introduction

With the advent of 5G technology, many Internet of Things (IoT) based applications have been developed for various human and business activities. Therefore, to meet the emerging stringent service requirements, the application delivery model has shifted from cloud computing to edge computing. Although cloud computing provides abundant computing resources, it cannot guarantee the low-latency requirements of modern applications. On the other hand, edge computing provides finite computing resources in the end user's proximity to augment the processing capabilities of resource/energy-constrained IoT devices [1]. The resource management of IoT-based applications is a challenging task and many orchestration platforms, which rely on virtualization technology, provide operations for the entire application life-cycle. For instance, OpenStack [2] is widely used as Virtualized Infrastructure Manager (VIM) for managing virtual machines (VMs) in cloud infrastructure. Additionally, Kubernetes [3] is used for orchestrating and managing containerized applications. Both platforms provide for automating deployment, scaling, management, and maintenance of the applications. Focusing on autoscaling, both Openstack and Kubernetes provide horizontal and vertical scaling. On both platforms, horizontal scaling is widely used. On the contrary, vertical scaling requires the restart of the VM (container), thus, it is not appropriate for real-time resource management. These scaling mechanisms are coarse and the scaling decision is usually based on simple monitoring metrics such as CPU and memory utilization without taking into account other important performance parameters and metrics, such as dynamic workload or energy consumption. Meeting the emerging 5G applications' rigorous requirements in terms of delay, throughput, and location calls for the improvement of the existing scaling mechanisms to tackle the underlying challenges.

Towards dynamic scaling, many recent studies focused on extending the scaling capabilities of the existing platforms by including various performance criteria and application parameters. Regarding VM technology, a scaling mechanism for location-based applications was proposed in [4]. Based on various VM flavors, a scaling and load balancing mechanism determines the number of replicas of each VM flavor to serve the total incoming workload. The authors in [5] proposed two heuristics for VM horizontal scaling. The first heuristic focuses only on cloud computing resources aiming at satisfying the resources requirements, while the second one also includes network resources and relies on multi-attribute decision making (MAMD) algorithms to place new replicas to selected nodes. However, in this approach, the scaling decision is only based on resource availability without considering other system parameters or performance metrics. Regarding Kubernetes, Horizontal Pod Autoscaler (HPA) component is responsible for scaling the containerized applications. Phan et al. [6] proposed a traffic-aware HPA that adjusts the replicas of pods in edge nodes based on the proportion of incoming requests accessing nodes in real-time. The authors in [7] proposed an HPA-based scaling mechanism for edge clusters. A loss-less MMPP/M/c queuing model and an ML-based pro-active scaling method were proposed and compared with default HPA. Four forecasting methods were used to estimate the varying incoming request rate. However, the above studies do not take into account other performance metrics (e.g., power consumption) in the scaling decision.

Contrary to the default HPA and the above studies, this work

This work was supported by the CHIST-ERA grant CHIST-ERA-18-SDCDN-003 (DRUID-NET), and is co-financed by Greece and European Union under the Operational Programme "Competitiveness, Entrepreneurship and Innovation" (EPAnEK) through the Greek General Secretariat for Research and Innovation (GSRI), grant number T11EPA4-00022.

978-1-6654-3540-6/22 © 2022 IEEE

aspires to include more criteria in the scaling decision focusing on power consumption and allocated resources. Towards this direction, we propose an AHP4HPA framework that is based on a custom autoscaling controller for Kubernetes and a multi-criteria decision making (MCDM) approach, which considers various key performance indicators (KPIs) and application parameters. Specifically, the contributions of our work are summarized as follows:

- The scaling decision relies on resource profiles that provide a mapping between Quality of Service (QoS) metrics and computing resources of an application pod to minimize any performance violations.
- The scaling decision of AHP4HPA is computed by Analytic Hierarchy Process (AHP) [8], which is an MCDM method and considers various KPIs and parameters such as incoming workload, power consumption, number of active servers, and monetary costs. This light-computing approach can assess numerous scaling solutions in real-time and demonstrate the trade-off between the application performance and cost minimization.
- AHP4HPA is evaluated in a small-scale edge cluster using a compute-intensive application and a dataset acquired from a touristic application. The results show that AHP4HPA significantly outperforms HPA in terms of power consumption and allocated resources, a the cost of a slight only increase in QoS violations.

The remainder of the paper is organized as follows. Section II provides an overview of Kubernetes scaling features and highlights the modifications introduced in our proposed framework. Section III presents the modeling of various performance, power and cost parameters included in AHP4HPA, while Section IV describes the AHP algorithm, which provides the scaling decision. Section V presents the experimental setup and the performance evaluation of the proposed AHP4HPA approach and finally, Section VI concludes the paper.

II. System Architecture

In the context of Kubernetes, (i) pod is the smallest deployment unit of computing resources for a containerized application, (ii) service is an abstract way to declare pods that have the same set of functions (applications), and (iii) deployment is a declarative way for creating, modifying and scaling the pods. Pods can run multiple containers of the same application, which are managed as a single entity. In this work, we consider that each pod runs only one container.

Definition 1: Resource Profile φ_i is a mapping of the pod's allocated resources to the maximum request rate without violating the delay constraints.

Let m denote the number of the distinct resource profiles for an application. For all resource profiles $\varphi_i,\ i = 1, \ldots, m$, we consider distinct services, and deployments and implement different resource limits for the pods. As stated in the Kubernetes documentation, the resource limits are exploited to enforce that a running pod utilizes at most the predefined resources in terms of CPU and memory, which are defined as tuple $\langle c_i, r_i \rangle$ for CPU and memory respectively. Moreover, for each

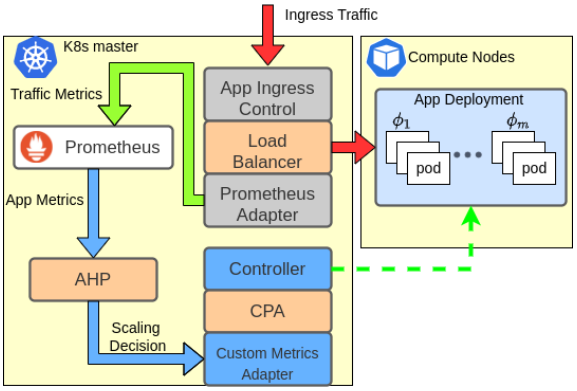

Fig. 1. AHP4HPA integrated with Kubernetes.

deployment, the number of identical pod replicas is defined as k_i and the upper replica limit is k_{max}. Thus, for each resource profile φ_i, $k_i \in [0, k_{max}]$, $k_i \in \mathbb{N}$. The resource profiles are created by experimenting with three different application settings, i.e., (i) the request rate, (ii) the resource limits, and (iii) the number of replicas. Through experiments, we identify the maximum request rate before observing deterioration in the overall application performance. To this extent, for each resource profile, a linear function $g_i(k_i) = \alpha_i k_i + \beta_i$ is computed by linear regression and maps the maximum request rate to the number of replicas. The App Deployment, which will be used by the AHP ranking below, is an abstraction that includes all deployments of all resource profiles, which serve the requests of a specific application.

Moreover, Kubernetes runs the containerized applications in Compute Nodes, which are hosted in Openstack VMs in our case. To instantiate additional compute nodes and expand the Cluster we employ the Kubernetes Cluster Autoscaler (CA)[1]. It resizes the cluster, i.e., (i) adds a node to the cluster when the available resources are limited, (ii) removes a node, if the resources are underutilized. In this work, we select a compute-intensive application to showcase the performance of AHP4HPA, i.e., an image classification application developed with OpenCV. Figure 1 presents the system architecture implemented in Kubernetes along with the workflow. The ingress traffic is redirected via a Load Balancer to the application pods, which are accessible externally from App Ingress Control. We deployed a Python Flask REST API to act as the Load Balancer. Then, Flask is integrated with Celery[2], which creates asynchronous tasks assigned to workers. For each service, a Celery worker is responsible for redirecting the HTTP requests to the corresponding pod. Subsequently, relying on deployed replicas for every profile, we dynamically load balance the incoming workload on the instantiated pods. We rely on the Prometheus monitoring system and Prometheus Adapter respectively for collecting traffic metrics and app metrics (per deployment, per pod). The core intelligence of the proposed framework is the AHP component. We consider that period-

[1]https://github.com/kubernetes/autoscaler
[2]https://docs.celeryq.dev/

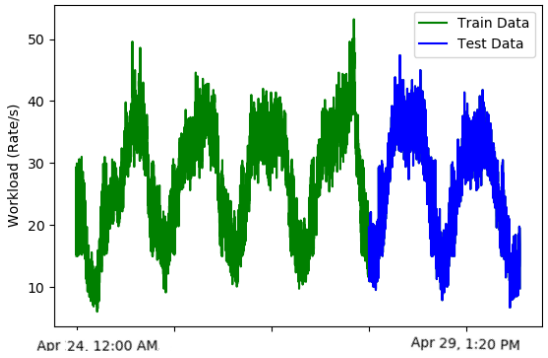

Fig. 2. Workload Trace used for training and experimentation.

ically the AHP checks the metrics and dictates the resource scaling decision for each deployment aiming at minimizing the total power consumption while meeting various QoS criteria. More information regarding the AHP-based scaling is provided in the following sections. The main scaling component in the Kubernetes ecosystem is the Horizontal Pod Autoscaler [9]. However, the HPA's algorithm is quite simplistic and mainly aims to stabilize the overall performance towards meeting a target value for a specific metric (e.g., CPU utilization). HPA scales horizontally (increases/decreases) the number of deployed replicas for a deployment. Therefore, the Custom Pod Autoscaler (CPA) [10] is employed to operate as the autoscaling controller, which enables custom algorithms for scaling to be realized. Moreover, CPA enables scaling to zero pods, which in the case of HPA is impossible and leads to higher power consumption. At each time slot, for each deployment, a CPA instance performs scaling according to the decisions of the AHP algorithm. All essential metrics for AHP assessment are exposed via the Prometheus Adapter and the scaling decision is consumed by the CPA via the Custom Metrics Adapter. Finally, we implement an autoregressive integrated moving average (ARIMA) model [11], which is widely used for time-series forecasting, to provide a prediction for the request rate for the next time slot. To validate the presented framework, we selected a workload trace from Ferryhopper website[3], which provides ferry booking services around Europe. The workload trace consists of HTTP requests produced by clients. As Kubernetes is mainly used for HTTP-based applications, this dataset is suitable for realistic evaluation. Figure 2 presents the distribution of request rate per minute over six days. The data of the first four days were used to train the ARIMA model while the latter for evaluation. At each time slot, the estimation of the ARIMA model is used by the AHP algorithms as described in III-F.

III. AHP4HPA in Details

Towards efficient horizontal autoscaling both in terms of QoS and power consumption, we implement the Analytic Hierarchy Process, which is a widely used MCDM method. AHP facilitates the evaluation of alternative scaling decisions for a specific application, by considering several criteria of various types. These criteria are hierarchically structured and separated into *KPIs* and *Attributes*. In detail, KPIs indicate specific technical metrics, such as performance or cost-related metrics, while attributes summarize correlated KPIs. In the following, we define the essential KPIs and attributes related to our scaling problem. The scaling decision concerns the selection of the appropriate Application Deployment (App Deployment), which determines a set of replicas of different resource profiles for the corresponding application.

We consider m distinct resource profiles φ_i for an application. The different combinations of k_i replicas determine the total number of the candidate App Deployments. For $i = 1, \ldots, m$, $k_i \in [0, k_{max}]$, thus, the total number of the candidate App Deployments is $\mathcal{M} = m^{k_{max}+1}$. An App Deployment D_j, where $j = 1, \ldots, \mathcal{M}$, is defined as:

$$D_j = \langle k_{ji} \rangle , \; i = 1, \ldots, m \; .$$

Given the set of candidate App Deployments U_D, the suggested framework strives to nominate the App Deployment $D^* \in U_D$, which minimizes the power consumption and the allocated computing resources from the infrastructure provider's perspective, while guarantees a certain QoS level for the user. The quantification of the above evaluation criteria is done by the definition of the appropriate KPIs as they are presented below.

A. Total Allocated Resources

In the context of micro-services, a resource profile reflects the allocated resources of containers. The most common resources to specify are CPU and memory. As we mentioned above, an App Deployment D_j is a set of replicas of different resource profiles φ_i. Each φ_i has specific resource request c_i of CPU cores and r_i of memory. So, we define two KPIs, to express the total resource demands for a D_j; first, the *CPU Cores* C_{D_j} and secondly the *RAM* R_{D_j}, where:

$$C_{D_j} = \sum_{i=1}^{m} c_i \quad (1) \qquad \text{and} \qquad R_{D_j} = \sum_{i=1}^{m} r_i \quad (2)$$

B. Active Servers

From the provider's perspective, the minimization of the power consumption is directly related to the reduction of active servers, as almost 70% of the maximum power consumption of a server occurs on its idle state [12]. The number of active servers demanded by an App Deployment is derived from the total resources required in relation to those provided in a single server of the infrastructure. Let C_{total} be the total CPU Cores and R_{total} the total GB of RAM, available in a server. For a specific D_j, the required number of active servers S_{D_j} is:

$$S_{D_j} = \max\left(\left\lceil \frac{C_{D_j}}{C_{total}} \right\rceil, \left\lceil \frac{R_{D_j}}{R_{total}} \right\rceil \right). \quad (3)$$

C. App Deployment's Power Consumption

Several energy-aware approaches for cloud computing propose a power model based on the server's maximum power consumption when it is fully loaded (P_{MAX}) and its idle state

[3]https://www.ferryhopper.com/

consumption (P_{min}). The following model is introduced in [12] to predict the server's power consumption:

$$P = \gamma * P_{MAX} + (1 - \gamma) * P_{MAX} * u. \quad (4)$$

The γ parameter refers to the proportion of the consumed power of an idle server with respect to P_{MAX}. The second term of equation (4), denotes the server's power consumption which occurs from its CPU utilization u. In accordance with [12], u depends on the allocated resources of a deployed application, with respect to the available resources of the server. Hence, in our model, we define the power consumption for an App Deployment D_j based on its resource requirements and the CPU utilization of the corresponding active servers:

$$P_{D_j} = (1 - \gamma) * P_{MAX} * \left(\frac{C_{D_j}}{S_{D_j} * C_{total}} \right). \quad (5)$$

Therefore, based on (4), (5), the total power consumption under the specified deployment is defined as,

$$P_{D_j}^{total} = S_{D_j} * \gamma * P_{MAX} + P_{D_j}. \quad (6)$$

D. Transformation Cost

The scaling process itself brings about extra resource management overhead, as different workloads require deployment adjustment. To quantify these adjustments required per case, we define the Transformation Cost KPI. It reflects on a penalization of the adjustments of each candidate App Deployment, from the current state. Let the scaling cost of a resource profile φ_i from current App Deployment D_c to the D_j is:

$$sc_j(\varphi_i) = \begin{cases} 1 + (k_{ji} - k_{ci})a, & if\ k_{ci} < k_{ji} \text{ (scale-out)} \\ 1 + (k_{ci} - k_{ji})(1 - a), & if\ k_{ci} \geq k_{ji} \text{ (scale-in)} \end{cases}. \quad (7)$$

where k_{ci} is the number of replicas of φ_i in the current App Deployment D_c. The parameter a is a degree of penalization regarding the scaling-out adjustments with respect to scaling-in changes. If $a = 0.5$ the penalty is equal for both cases. So, the Transformation Cost for the candidate D_j is:

$$TC_{D_j} = \sum_{i=1}^{m} sc_j(\varphi_i). \quad (8)$$

E. App Deployment's Billing

Typically, cloud providers charge VM or container's instance based on its type (e.g., OS type) as well as the required resources, for a specific period of time. Therefore, for each resource profile φ_i, the corresponding billing for one replica of φ_i is defined as $b(\varphi_i)$. Thus, for D_j the total billing is:

$$B_{D_j} = \sum_{i=1}^{m} k_{ji} b(\varphi_i) \ . \quad (9)$$

F. Resource Profiling - QoS

Towards the selection of the appropriate scaling decision, it is important to include a QoS-related metric for the evaluation of the candidate App Deployments. To address this, we rely on (i) a profiling regarding the potential service rate for every available resource profiles φ_i and (ii) on a prediction model of the incoming workload $\tilde{\lambda}$ for the respective application in a specific time-slot. The predicted workload rate is expressed as requests per second. Concerning the resource profiling, which corresponds to the total service rate for the next time slot for each candidate App Deployment, we calculate the QoS profile for each one of them as: $f(D_j) = \sum_{i=1}^{m} g(k_{ji})$. So, in order to evaluate the candidate App Deployments, the QoS Profiling score KPI is determined as,

$$QoS_{D_j} = \begin{cases} 0.2, & \text{if } f(D_j) < \tilde{\lambda} \\ 1 + \frac{1}{2} * [1 + (f(D_j) - \tilde{\lambda})]^{-1}, & \text{if } f(D_j) \geq \tilde{\lambda} \end{cases}. \quad (10)$$

This KPI penalizes candidates that do not guarantee the QoS for the predicted workload. Regarding the case where $f(D_j) \geq \tilde{\lambda}$, the score is lower for over-provisioned resources for the respective $\tilde{\lambda}$. Specifications about the profiling and the workload predictor model can be found in section II.

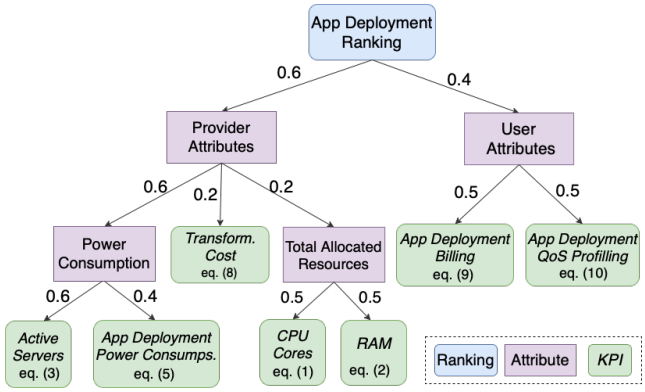

Fig. 3. AHP Hierarchical Structure Model.

IV. AHP-Based Autoscaling

In this section, an overview of the MCDM method of the proposed framework is presented. Specifically, the ranking process of the candidate App Deployments for scaling is based on AHP, which enables the simultaneous processing of various criteria, by structuring them in a hierarchical manner. Thus, provider-related attributes such as Power Consumption, and Resource Allocation can be combined with user-related requirements for high QoS with minimum billing. In the following, we present the AHP adaptations for AHP4HPA. The AHP can be divided into three main phases:

Phase 1 - Hierarchical Structure & Weight Assignment: Based on the above described KPIs and attributes, the hierarchical structure of the proposed framework is depicted in Figure 3. The leaves on the hierarchy tree represent the KPIs of the model, which are condensed in an attribute of a higher level. Furthermore, AHP enables the configuration of the importance of each criterion, as a weight assignment on the edge between two criteria. Regarding the weight assignment, the sum of weights for a set of siblings KPIs and attributes is equal to 1. Figure 3, also, includes the weight assignment for our framework and reflects the main scope of AHP4HPA, which is to minimize the total power consumption, while taking into consideration multiple other parameters.

Phase 2 - Relative Attribute Score Computation: Candidate App Deployments are assigned a value for each KPI of the

TABLE I
SETTING FOR RESOURCE FLAVORS

Resource Profiles	Small	Medium	Large	Kubernetes Node
CPU cores	1	2	4	8
RAM (GB)	2	4	8	16
Billing per φ_i ($)	7	13	21	

structure based on equations (1) - (10). A Relative Comparison Matrix (RCM) is computed for every KPI. In our work, the KPIs are numeric values of two types: (i) *Higher-is-better* and (ii) *Lower-is-better*. In our model, all KPIs are numeric, *Lower-is-better* criteria, except of the *QoS Profiling* in eq. (10), which is of *Higher-is-better* type. Let A_j denotes the value assigned in KPI X for the candidate D_j. Then, for $j = 1, \ldots, \mathcal{M}$, where $\mathcal{M} = m^{k+1}$, we compute the RCM of KPI X -if X is of *higher-is-better* type- as follows:

$$RCM_X = \begin{bmatrix} 1 & A_1/A_2 & \ldots & A_1/A_{\mathcal{M}} \\ A_2/A_1 & 1 & \ldots & A_2/A_{\mathcal{M}} \\ \vdots & \vdots & & \vdots \\ A_{\mathcal{M}}/A_1 & A_{\mathcal{M}}/A_2 & \ldots & 1 \end{bmatrix}. \quad (11)$$

For *Lower is better* KPIs, the RCM occurs by the computation of the transposed matrix of (11). Furthermore, given the $RCM_X = [x_{ij}],\ i, j = 1, \ldots, \mathcal{M}$, we calculate a Relative Ranking Vector (RRV) for each KPI as:

$$RRV_X = [v_{X1} \ldots v_{X2}],\ \text{where}\ v_{Xi} = \frac{\sum_{j=1}^{\mathcal{M}} x_{ij}}{\sum_{i=1}^{\mathcal{M}} \sum_{j=1}^{\mathcal{M}} x_{ij}}. \quad (12)$$

More details regarding the AHP calculations of different type KPIs can be found in [8].

Phase 3 - App Deployments Ranking and Decision: For all Attributes, the RRV is computed in a bottom-up fashion, until the computation of the RRV of the App Deployments Ranking (top level attribute). The computations are based on the lower level RRVs and the weights among siblings KPIs and attributes. For a parent attribute with n sub-attributes and the weight vector, which consists of n values, the RRV is:

$$RRV_{par} = [RRV^T_{sub1} \ldots RRV^T_{subn}] * [w_{sub1} \ldots w_{subn}]^T \quad (13)$$

Finally, an RRV is computed for the ranking of the candidate App Deployments for scaling. This RRV has the form of: $[r_1,\ r_2, \ldots,\ r_{\mathcal{M}}]$. The maximum $r^* = \max[r_j]_{j=1}^{\mathcal{M}}$ value corresponds to the nominated App Deployment D^*.

V. EXPERIMENTAL EVALUATION

In this section, we present the experimental setup and comparative performance evaluation of the AHP4HPA framework. The Kubernetes Cluster is deployed in self-hosted Openstack VMs to enable the Kubernetes Cluster Autoscaler. We consider three types of resource profiles, $m = 3$, i.e., small, medium, large, and the maximum number of replicas for each φ_i is $k_{max} = 4$. The resource limits and billing for each resource profile along with the capacity of the Kubernetes Nodes, are presented in Table I. For the billing of the resources, we take information from Azure[4] pricing calculator, and we consider that solely a new replica instantiation is charged for the whole amount. The maximum power consumption for a fully operational server is $P_{MAX} = 2000W$ in accordance to [12]. We trained the workload prediction model with the Ferryhopper dataset, using an ARIMA model of order=(3,1,1), which has sufficient accuracy. The evaluation of the ARIMA model or an alternative prediction methods is out of scope for this work. Moreover, we assume that when the response time of a request for the image classification application takes three times more than expected, the request is considered lost, and we enforce a connection timeout.

[4]https://azure.microsoft.com/en-us/pricing/calculator/

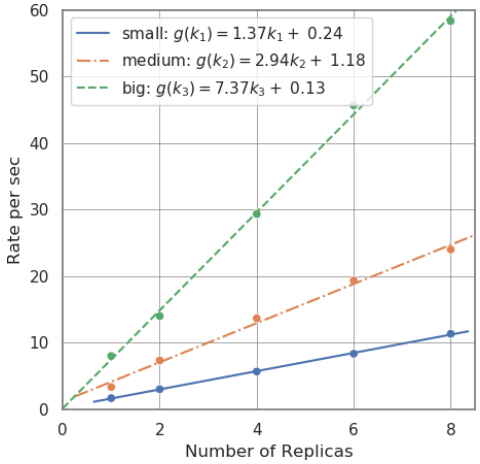

Fig. 4. The maximum rate per sec per pod for each resource profile.

Figure 4 presents the maximum request rate for the respective number of replicas for each resource profile. We use a linear regression technique to extract the respective QoS mapping $g_i(k_i)$ for this application for the three resource profiles. QoS mapping functions are also utilized for dynamic load balancing of the ingress traffic, according to the deployed number of replicas for each App Deployment. We selected to compare three different setups, namely: (i) the proposed AHP4HPA, (ii) modified HPA (M-HPA), and (iii) the default HPA (D-HPA) of Kubernetes. For M-HPA, the resource profiles are also included along with the dynamic load balancing. The HPA is trying to target 70% CPU utilization for the deployed pods. On the other hand, for D-HPA none of the proposed techniques is employed, so we select to target 70% CPU utilization, utilizing only the medium resource profile. To produce unbiased results, all three scaling mechanisms operate every 15s (time slot), while we evaluate them against a three-hour workload traffic dataset from the Ferryhopper test dataset. As the deployed application is mainly compute-intensive, in the following results, we focus on CPU utilization. All three scaling decisions run in the order of ms, as a result the overall performance is not disrupted. In Figure 5, the power consumption of the deployed resources for the three experiments is presented. We calculated the power consumption using (6) according to the deployment instantiated by each autoscaling method at each time slot. The results for each experiment are summarized in Table II. Specifically, the average energy

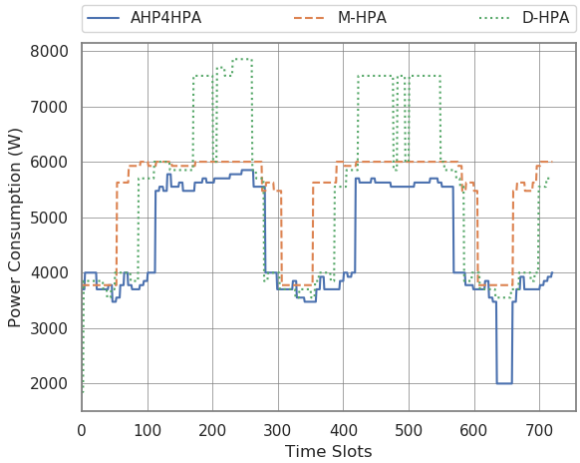

Fig. 5. Power Consumption in W for each experiment at each time slot

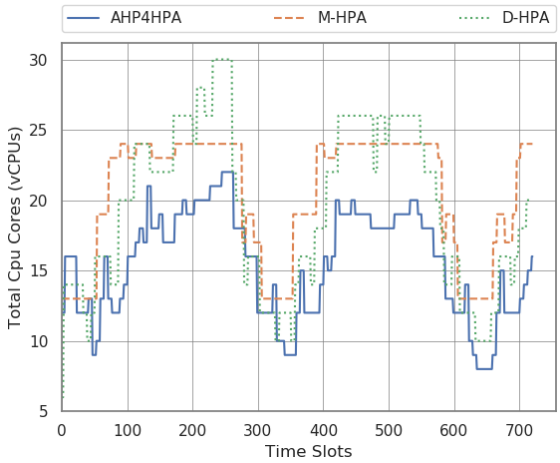

Fig. 6. Number of cores instantiated for each experiment at each time slot

consumption throughout the experiment results in 3.96 kWh for AHP4HPA. Similarly, in Figure 6, the total number of cores instantiated at each time slot for each setup is presented. For AHP4HPA, the CPU cores deployed for all resource flavors are 15.8 vCPUs averaged throughout the experiment. Our solution outperforms both M-HPA and D-HPA, producing 9% and 14% less average energy consumption accordingly. The proposed framework sufficiently serves the incoming workload having only 1.53% percent of total lost requests, while M-HPA has 0.37% and D-HPA has 0.25%. The difference in total request loss is mainly due to the workload prediction miscalculations produced by the ARIMA. We should also mention that the average energy consumption of M-HPA is 4.32 kWh with average CPU cores of 20.5 vCPUs while the D-HPA is 4.57 kWh and 19.4 vCPUs. Hence, these results indicate that the inclusion of resource profiles leads to 6.5% less average energy consumption, regardless of the increased utilization of CPU resources. This is explained since D-HPA autoscaling results in instantiate a 4^{th} node in the Kubernetes Cluster, consuming unnecessary power to serve the incoming traffic.

TABLE II
RESULTS FOR THE THREE EXPERIMENTS

Setup	Total Request Loss	Average Energy Consumption	Average CPU cores
AHP4HPA	1.53%	3.96 kWh	15.8
M-HPA	0.37%	4.32 kWh	20.5
D-HPA	0.25%	4.57 kWh	19.4

VI. Conclusions and Future Work

In this paper, the AHP4HPA framework for autoscaling in Kubernetes Clusters is introduced. Focusing primarily on minimizing power consumption and satisfying several requirements of both infrastructure provider and users, AHP is utilized to enable the ranking of different scaling decisions. Also, we proposed a set of KPIs to be included in the hierarchical structure, as well as the definition of the corresponding weights to adapt the AHP algorithm, focusing on autoscaling with power minimization. To extend the AHP capabilities, we introduced different resource profiles and a prediction of the incoming workload. Evaluation results indicate a significant reduction in the consumed power from scaling decisions derived from AHP4HPA in comparison with other solutions, while reducing the allocated CPU Cores, with a negligible increase in request loss. For future plans, we aim to enhance the resource profiles by leveraging Machine Learning (ML) classification algorithms to achieve minimal request loss.

References

[1] D. Dechouniotis, N. Athanasopoulos, A. Leivadeas, N. Mitton, R. Jungers, and S. Papavassiliou, "Edge computing resource allocation for dynamic networks: The DRUID-NET vision and perspective," *Sensors*, vol. 20, no. 8, p. 2191, 2020.
[2] OpenStack. https://www.openstack.org/, Last Accessed on 2022-05-01.
[3] Kubernetes. https://kubernetes.io/, Last Accessed on 2022-05-01.
[4] D. Spatharakis, I. Dimolitsas, D. Dechouniotis, G. Papathanail, I. Fotoglou, P. Papadimitriou, and S. Papavassiliou, "A scalable edge computing architecture enabling smart offloading for location based services," *Pervasive and Mobile Computing*, vol. 67, p. 101217, 2020.
[5] E. Zeydan, J. Mangues-Bafalluy, J. Baranda, R. Martínez, and L. Vettori, "A Multi-criteria Decision Making Approach for Scaling and Placement of Virtual Network Functions," *Journal of Network and Systems Management*, vol. 30, no. 2, pp. 1–36, 2022.
[6] L.-A. Phan, T. Kim, *et al.*, "Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure," *IEEE Access*, vol. 10, pp. 18966–18977, 2022.
[7] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, 2021.
[8] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1012–1023, 2013.
[9] Horizontal-Pod-Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, Last Accessed on 2022-05-01.
[10] Custom-Pod-Autoscaler. https://custom-pod-autoscaler.readthedocs.io, Last Accessed on 2022-05-01.
[11] R. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using ARIMA model and its impact on cloud applications' QoS," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2014.
[12] L. Ismail and H. Materwala, "Computing Server Power Modeling in a Data Center: Survey, Taxonomy, and Performance Evaluation," *ACM Comput. Surv.*, vol. 53, jun 2020.