Distributed Resource Autoscaling in Kubernetes Edge Clusters

Dimitrios Spatharakis*, Ioannis Dimolitsas*, Eleftherios Vlahakis†, Dimitrios Dechouniotis*
Nikolaos Athanasopoulos†, Symeon Papavassiliou*

*School of Electrical and Computer Engineering, National Technical University of Athens, Greece
†School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Northern Ireland, UK
{dspatharakis, jdimol, ddechou}@netmode.ntua.gr, {e.vlahakis, n.athanasopoulos}@qub.ac.uk, papavass@mail.ntua.gr

***Abstract*—Maximizing the performance of modern applications requires timely resource management of the virtualized resources. However, proactively deploying resources for meeting specific application requirements subject to a dynamic workload profile of incoming requests is extremely challenging. To this end, the fundamental problems of task scheduling and resource autoscaling must be jointly addressed. This paper presents a scalable architecture compatible with the decentralized nature of Kubernetes [1], to solve both. Exploiting the stability guarantees of a novel AIMD-like task scheduling solution, we dynamically redirect the incoming requests towards the containerized application. To cope with dynamic workloads, a prediction mechanism allows us to estimate the number of incoming requests. Additionally, a Machine Learning-based (ML) Application Profiling Modeling is introduced to address the scaling, by co-designing the theoretically-computed service rates obtained from the AIMD algorithm with the current performance metrics. The proposed solution is compared with the state-of-the-art autoscaling techniques under a realistic dataset in a small edge infrastructure and the trade-off between resource utilization and QoS violations are analyzed. Our solution provides better resource utilization by reducing CPU cores by 8% with only an acceptable increase in QoS violations.**

***Index Terms*—Edge Computing, Resource Management, Resource Autoscaling, Machine Learning, Kubernetes**

I. Introduction

The proliferation of Internet of Things (IoT) and the immense growth of modern applications have increased the need for low latency to a remarkable extent. However, the resource management of IoT applications to meet the emerging stringent service requirements remains a major challenge. Edge Computing [2] plays a key role in providing a solution to the ever-expanding need for additional resources for resource-constrained IoT devices. Hosting these applications at the network edge alleviates the communication overhead between end devices and computing resources. On the other hand, optimizing resource utilization is essential, as the edge infrastructure usually has limited resources. Moreover, various open-source orchestration tools aim at supporting every stage of the life-cycle of the deployed applications, e.g., instantiation, monitoring, and scaling. Kubernetes [1] is a state-of-the-art resource orchestration platform for containerized applications. The Horizontal Pod Autoscaler (HPA) [3] is widely used to implement the scaling of resources. The HPA's functionality mainly relies on regulating the utilization of the deployed resources towards meeting a target value of a single or a set of performance metrics, e.g., CPU utilization. Therefore, the HPA scales horizontally the number of deployed containers to meet these targets. However, 5G applications require more sophisticated scaling techniques to incorporate the dynamic nature of the incoming workload and the complex operation of modern applications [4].

The aforementioned challenges are summarised in the following two fundamental problems; (i) the scheduling of the incoming requests and (ii) the appropriate instantiation of resources for hosting the application. In this paper an Additive Increase Multiplicative Decrease (AIMD) based scheme is proposed for the task scheduling problem. Our approach is inspired by the AIMD algorithm, a celebrated method for congestion avoidance in the context of network management [5]. The proposed AIMD-based algorithm has recently been introduced in [6] and provides a simple, stabilizing and decentralised solution for parallel task scheduling and resource allocation of distributed computing systems.

Meeting critical QoS requirements via dynamic resource scaling is a major challenge primarily due to a set of competing Key Performance Indicators (KPIs), such as the utilization of deployed resources and the response time of the underlying application. A key enabler of scaling resources according to the workload traffic is the identification of the maximum number of requests that can be processed by the provisioned resources [7]. Many recent Machine Learning (ML) based studies investigate extensively the challenging problem of resource management [8]. Nevertheless, the proposed ML-based solutions typically have a very challenging training phase and their scalability is limited [9]. Contrary to the one-sided philosophy of the HPA that takes into account only performance metrics, in this study, we propose a versatile architecture for resource management in Kubernetes Edge Clusters (KEC). We consider a scenario where users offload their requests to a KEC for further processing. The key contributions of our work are summarized as follows:

- A holistic scalable architecture to tackle the joint problem of task scheduling and proactive resource management in KEC. We exploit an AIMD-like solution for the scheduling problem of the incoming requests. The proposed event-triggered AIMD scheme facilitates proactive scaling of

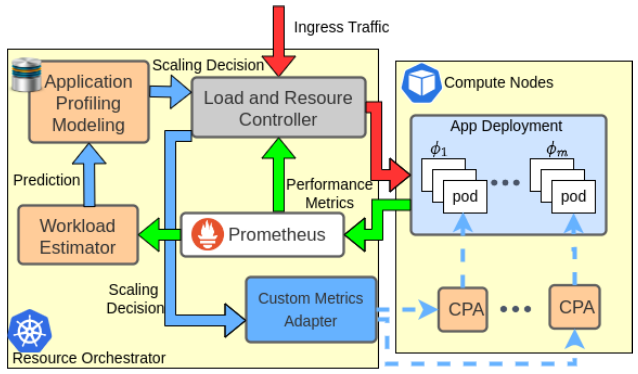

Fig. 1: The proposed System Architecture for KEC.

resources subject to a dynamic workload profile via a novel, intuitively conceived locally identifiable triggering condition. This mechanism enables decentralized resource orchestration at the network edge.

- An Application Profiling Modeling component is introduced that leverages information from (a) the scheduling and resource allocation solution, (b) the monitored KPIs, and (c) the workload prediction algorithm to estimate the essential number of replicas for meeting the workload demand. By collecting a set of distinct resource profiles, all associated with a resource allocation solution obtained from the AIMD mechanism, we can optimize resource utilization without violating the QoS requirements. Then, the scaling decision is calculated using a simple ML technique that allows us to also incorporate the performance metrics. The proposed design is an easily re-configurable solution to the autoscaling problem in KECs.
- The proposed architecture is evaluated in a small-scale KEC using a computing-intensive application and a real dataset of a touristic application. Our numerical results illustrate how to efficiently trade off between allocated resources and application performance, highlighting that the proposed framework significantly outperforms various configurations of well-known autoscaling solutions in terms of the utilization of resources.

The remainder of the paper is organized as follows. Section II, the proposed system architecture is presented. Section III presents the AIMD-like solution for the task scheduling problems. In Section IV, we introduce an ML-based Application Profiling Modeling that performs the resource scaling in the KEC. Section V presents the experimental evaluation of the proposed work compared with other solutions using a realistic dataset. Finally, Section VI concludes the paper and discuss our future plans.

II. System Architecture

In this Section, we present the core components of the proposed architecture deployed in the KEC. We assume that IoT devices generate workload for a specific application that arrives at the edge layer. We select Kubernetes as the Resource Orchestrator of the virtualized resources. Therefore, the architecture relies on widely used open-source state-of-the-art software tools for resource management and monitoring. An example application could be but is not limited to a computationally intensive algorithm, such as image classification. We should note that in most cases, Kubernetes is mainly used for web applications that need to perform rapid scaling decisions according to the ingress traffic. The proposed architecture operates in the same layer as the Resource Orchestrator and has three main components, namely i) the Load and resource Controller, ii) the Application Profiling Modeling, and iii) the Workload Estimator, as Figure 1 depicts. These components are responsible for incorporating the decisions of the resource allocation mechanisms with the ones of the time-varying workload. In particular, we attempt to propose a holistic solution that simultaneously optimizes well-defined metrics related to the infrastructure (computing resources), and stabilizes the performance of the offloaded application, leading to safe, predictable, and optimal behaviors. The functionalities of each component are summarized here:

Load and Resource Controller: the objectives of this component are (i) task scheduling and (ii) resource autoscaling. Regarding the first objective, this component is responsible for dynamic task scheduling of the offloaded requests based on the output of the AIMD scheme. Secondly, using information from the Application Profiling Modeling component, this component dictates the resource allocation strategy, aiming at guaranteeing the application's performance requirement in terms of QoS metrics and concurrently guaranteeing various system properties.

Workload Modeling: this component is responsible for estimating the number of incoming requests for the containerized application leveraging a time-series forecasting method. The output of this component is used to take proactively scaling decisions.

Application Profiling Modeling: this component aims at constructing representative dynamic models that can be subsequently utilized for dynamic resource allocation of KECs. Based on ML, various models are extracted to sufficiently express the dynamic relationship between output (QoS) and control variables, i.e., allocated CPU resources, admitted workload, and active requests.

Kubernetes is used in this work as a resource orchestration platform. In this ecosystem, (i) a *pod* is the smallest deployment unit of computing resources for a containerized application, (ii) a *service* is an abstract way to declare applications as a network service, and (iii) *deployment* is a declarative way for the instantiation, configuration, and scaling of the pods that serve an application. Typically, multiple containers co-exist in the same application pod, managed as the same entity. Without loss of generality, we assume that each pod hosts exactly one container. Next, we define the application Resource Profiles:

Definition 1. *An application Resource Profile φ_i is the relationship between the allocated resources of a pod and the maximum request rate of incoming requests that the pod serves* *while keeping a certain QoS level.*

For example, we assume that one container allocated with one vCPU and 1 GB of RAM, can serve on average two requests per second without having QoS violations. We let m be the number of the different resource profiles for a specific application. Therefore, a Kubernetes service and deployment are realized for all resource profiles; $\varphi_i,\ i = 1, \ldots, m$. We define the App Deployment as the abstract way to refer to the network and the computing resources of each resource profile. Then, each App Deployment has different available resource limits for the deployed pods. In the context of Kubernetes, the resource limits are used to enforce that the instantiated containers of the pod will operate in predefined regions in terms of CPU and memory. Moreover, for each App Deployment, r_i denotes the number of identical pod replicas that are running. The upper replica limit is considered the same for all deployments and expressed as r_{max}. To identify the resource profiles, extensive offline experimentation has been carried out involving three different configurations regarding (i) the incoming request rate, (ii) the resource limits, and (iii) the number of replicas. The aim was to empirically compute the maximum request rate before QoS violations occur for different sets of resource limits and replicas. Concerning the monitoring system, we rely on Prometheus [10] for collecting (i) the workload-, (ii) performance- and (iii) application-, related metrics. The metrics are updated at each time slot. Moreover, we utilized the Custom Pod Autoscaler (CPA) [11] as the controller that scales the App Deployments. For each resource profile, a CPA instance is realized to scale in/out the number of replicas, which varies from zero to r_{max}. Contrary to CPA, HPA does not support zero replicas. Hence, CPA enforces the scaling decision of the Load and Resource Controller, checking for changes in the desired number of replicas at each time slot.

The core intelligence of the proposed framework is provided by the Load and Resource Controller. The incoming requests are load-balanced between the App Deployments according to the output of the scheduling algorithm, which is deployed on this component. The Workload Estimator takes input from Prometheus to predict the incoming workload. At each time slot, given this prediction, the performance metrics, and the output of the scheduling algorithm, the Application Profiling Modeling component computes the desired number of replicas for each App Deployment using an ML model. Then, the Load and Resource Controller exposes the scaling decision for each App Deployment, via the Custom Metrics Adapter to be fed to the CPA's Controller unit. The complete architecture of the proposed solution is illustrated in Figure 1. In the following Sections, the above three components are analyzed in detail.

III. Load and Resource Controller

This Section describes the functionalities of the Load and Resource Controller. An overview of the AIMD-like task scheduling algorithm is described. Then, the actual implementation of the algorithm for a KEC is presented.

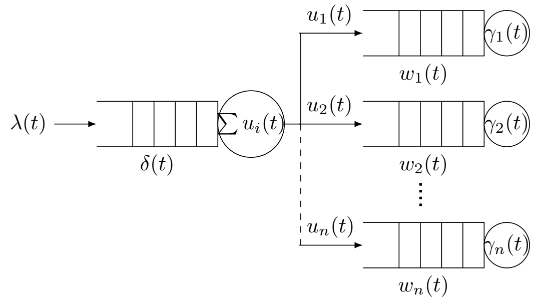

Fig. 2: A multi-queue system with AIMD policy.

A. AIMD-like Task Scheduling Algorithm

As Figure 2 shows, a central task queue $\delta(t)$ is an aggregation point at which requests enter a multi-queue system with an arrival rate $\lambda(t)$ and, then, are dispatched to processing nodes in a First Come First Served policy. An event is generated whenever this queue vanishes, which is guaranteed to happen since the admission rates $u_i(t)$ are piecewise increasing functions of time as shown in Eq. (1). At the time of the empty queue event, the admission rates drop instantaneously. Each request is processed via a multi-queue system with multiple processing nodes. Each processing node $i = 1, \ldots, n$ has a service rate $\gamma_i(t)$ and the queued requests at each node are denoted by $w_i(t)$.

The proposed algorithm has two phases, namely, (a) the Additive Increase (AI) phase and (b) the Multiplicative Decrease (MD) phase. During the first phase, the admission rates towards the processing units increase linearly with time aiming to admit all of the queued requests and drain the central queue. Inevitably, an empty queue event is triggered, i.e., $\delta(t_k) = 0$, where k is the number of events. At this instant, the scheduling algorithm enters the MD phase, which enforces a rapid decrease in the admission rates, and then switches back to the AI phase immediately. The admission rate towards the i^{th} processing node is defined as:

$$u_i(t) = \beta_i u_i(t_k) + a_i(t - t_k),\ i = 1, \ldots, n, \tag{1}$$

where t is the current time, t_k is the time of the k^{th} empty queue event and $\alpha_i > 0, 0 < \beta_i < 1,\ i = 1, \ldots, n$ are tuning AIMD parameters. Specifically, α_i is the growth rate of the admission rate of node i and β_i is the multiplicative decrease factor of the admission rate when an event is generated. Thus, a request arriving at the aggregation point is held at the central task queue until it is redirected to a processing node. The total rate of redirecting requests from the central task queue to the processing nodes at each time is $\sum_i^n u_i(t)$. It is easy to show that the inter-event period between two empty queue events is defined as:

$$T(k) = \max(0,\ \frac{\lambda(k) - \sum_{i=1}^n \beta_i u_i(k)}{\sum_{i=1}^n \frac{1}{2} a_i}), \tag{2}$$

where $u_i(k)$ denotes the admission rate towards the i^{th} node at time $t = t_k$. Under the scheduling solution (1)-(2), results from

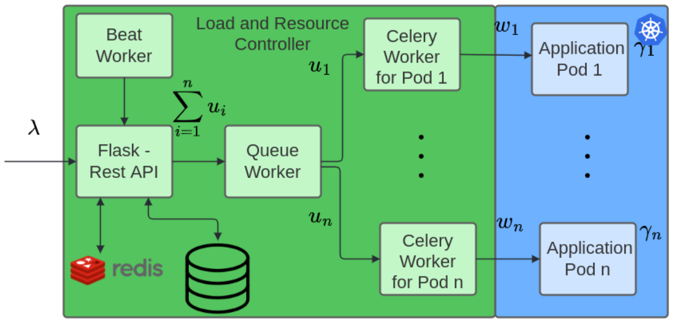

Fig. 3: AIMD integration with Kubernetes.

[6] indicate that if we consider the following decentralized resource allocation policy for the service rate of each node at the k^{th} event:

$$\gamma_i(t_k) = \beta_i u_i(t_k) + \sqrt{2a_i w_i(t_k)}, \tag{3}$$

then, the overall system is stable, in the sense that performance metrics, such as, execution and response times, are bounded as more requests are added to the system. QoS requirements can be attained via an efficient tuning of the AIMD parameters, see e.g., [12]. Moreover, individual service rates are adjusted only when an empty queue event occurs via the decentralised feedback rule (3) which requires only local information, namely, the local admission rate u_i and queue length w_i. To sum up, requests arriving at the first queue are not served instantaneously by a processing node having a slight increase in the total response time, however, they are placed subsequently in local queues in a way guaranteeing the stability of the overall system. We should note that the AIMD parameters can be selected individually for each processing node and remain constant. Thus, Eq. (3) is a decentralized resource allocation feedback controller, scalable and locally configurable. Moreover, in [6] it is shown that $T(k)$ also converges after a few steps of the algorithm. For the interested reader, a thorough overview of AIMD-like algorithms can be found in [13].

B. Integration with Kubernetes

In this section, we provide information about the integration in Kubernetes of the Load and Resource Controller component that exploits the AIMD-like task scheduling policy and resource allocation. To develop the AIMD algorithm several open-source software tools and technologies are used. In what follows we briefly explain the functionalities, as we believe it is important for understanding the remainder of the paper. The main components are presented in Figure 3. The ingress traffic for a specific application hits the exposed service of a Flask Rest API. This API is the aggregation point that also performs the task scheduling policy. To implement the dynamic task scheduling policy of the AIMD algorithm, we selected Python Celery, which is a software that suits well with Flask with its main functionality being the creation of workers to implement asynchronous tasks. The workers communicate via redis, a messaging broker, monitoring several events, e.g., the arrival of a request, completion of a task, etc. The Queue Worker implements the central task queue $\delta(t)$. Celery can implement a rate limit for each worker, thus, the Queue Worker redirects tasks with a rate of $\sum_i^n u_i(t)$ at each time slot. Moreover, the Beat Worker is triggered every time slot to update the admission rates and enforce the new rate limit to the Queue Worker. Hence, the requests may be held in the aggregation point before being redirected to the processing nodes due to the rate limit. We should note that the rate limit is a common practice at the production level to enforce billing policies for the incoming traffic. The dynamic task scheduling policy serves as a stabilizer for the processing nodes giving them time to process the requests and instantiate new resources accordingly. The configuration and monitoring of all of these procedures are stored in a Postgresql relational database.

As mentioned above, we consider m distinct resource profiles, φ_m, $i = 1, \ldots, m$, with different processing capabilities as the processing nodes. For each resource profile, a separate Celery Worker is created to redirect the HTTP traffic from the Queue Worker to the corresponding Application Pod that serves the virtualized instance of the application. We consider that if a request is redirected by a Celery Worker towards the i^{th} application pod then it belongs to the w_i queue. Subsequently, we dynamically load balance the incoming workload on the instantiated pods. The proposed architecture operates in a distributed manner since, for each computing node, no knowledge of the state of other nodes is needed. We should also mention that it is possible to have multiple aggregation points for the incoming workload of multiple applications. However, a missing capability is to interpret and subsequently enforce the computed theoretical service rates to each processing node, namely, the number of replicas for each resource profile φ_i to guarantee at least a service rate of $\gamma_i(k)$. In the next Section, we provide a solution to this challenging problem.

IV. Workload Estimator and Application Profiling Modeling

In this Section we present the last two components of the proposed system architecture i.e., (a) the Workload Estimator, and (b) the Application Profiling Modeling.

A. Workload Estimator

To timely adjust the service rate of the deployed resources, it is important to predict the admission rates. As the AIMD algorithm dynamically changes the admitted workload $u_i(t)$ towards the pods, the Workload Estimator tries to estimate the number of requests that will be admitted until the next empty queue event under the dynamic workload. This is important as continuously applying different service rates to the pods, i.e., scaling out/in the number of replicas may lead to performance deterioration. Hence, we deploy an autoregressive integrated moving average (ARIMA) model [14]. Such predictive techniques are very common for time-series forecasting. The model is trained using offline information and retrains online as the workload varies. This information is then used from

TABLE I: Dataset's Features Specifications.

Feature	Component	Specification
γ_i	AIMD Algorithm	The service rate of each node.
w_i	AIMD Algorithm	The queued requests at each node.
u_i	AIMD Algorithm	The admission rate to each node.
u_i^{real}	Performance Metric	The actual admission rate to each node during a time window.
u_i^{pred}	Predicted Metric	The admission rate predicted by the Workload Estimator for each node.
art_i	Performance Metric	The average response time obtained by each node.
r_i	Performance Metric	The number of replicas for the underlying resource profile φ_i.

the Application Profiling Modeling component to deploy the desirable number of replicas for each resource profile.

B. Application Profiling Modeling

The Application Profiling Modeling is based on a machine learning (ML) technique, which essentially involves the classification of distinct combinations of computing resources concerning the application's service rate, as in [15]. The classification calculates the number of replicas r_i for each resource profile φ_i, leveraging various metrics retrieved from the core components of the proposed framework. Then, the scaling decision is fed to Load and Resource Controller. Subsequently, the training process of distinct models demands separate data that characterize the respective resource profile φ_i.

Several features can be taken into account to characterize the performance of a pod, the pod's resource utilization, and the service and admission rates occurred by the task scheduling solution. To acquire the data that contain the necessary metrics regarding the scaling decision, we conducted offline experiments, utilizing the components of the proposed architecture for isolated resource profiles. In detail, per experiment, only one resource profile is used for the corresponding application. Different configuration schemes of (i) the incoming request rate, (ii) the resource limits, and (iii) the number of replicas were applied to identify settings that meet the QoS constraints for the underlying application. Subsequently, the collection of data is categorized in the components of the architecture as depicted in Table I. The collected data from the aforementioned experiments are raw and therefore not suitable for training machine learning models. So, we use several ML data pre-processing techniques to prepare the dataset for training our Application Profile Model in order to achieve faster training and improved classification accuracy. Indicatively, feature scaling via normalization is initially performed, followed by dataset balancing, aiming at equal occurrence of data from each class of the dataset, and, finally, feature importance that reflect more information regarding our classification problem while reducing the dataset's dimensions.

The Application Profiling Modeling component is responsible for providing the CPA with the value of r_i at each event, leveraging the aforementioned selected features, i.e., u_i^{real}, γ_i, w_i, and u_i^{pred}. We selected the k-Nearest Neighbors (kNN) [16] algorithm as it is an instance-based learning algorithm with fast training phase, and it involves only storing feature vectors and class labels of the training samples. Also, it suits multi-class problems like ours, with its non-parametric nature being appropriate for small datasets since no assumptions on the data distribution are required. As Table II shows, three different resource profiles are used. For each resource profile, we generate a training and an evaluation dataset. Based on the aforementioned pre-processing techniques, dataset preparation is performed, prior to the training phase. The ratio between training and test sample points is 70% to 30% of the initial dataset. The proposed scaling decision runs in the order of ms, so the overall performance does not decline. More details on kNN hyperparameter configuration are given in Section V.

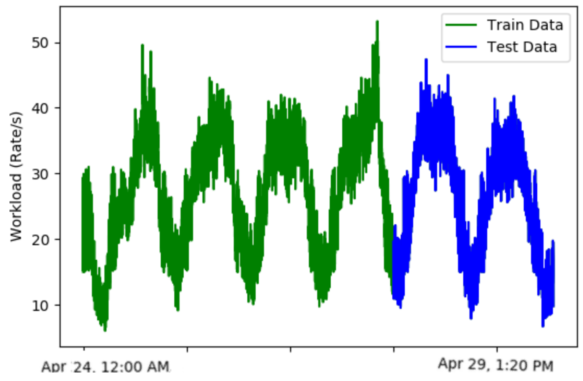

Fig. 4: Workload Trace used for training and experimentation.

V. Experimental Evaluation

This Section presents the experiment setup and the experimental evaluation of the proposed architecture with other autoscaling schemes for KECs. We consider three different resource profiles, $m = 3$, namely small, medium, and large, following the production standards, see, e.g., Azure[1]. The maximum number of replicas is $r_{max} = 4$. The resource limits and AIMD parameters for each resource profile are presented in Table II. Also, the maximum request rate that a pod can serve before noticing QoS violations is presented in the same Table. The application is a simple image classification application that uses OpenCV. We assume that if a request takes three times more than expected to be processed then we have a QoS violation, and the request is rejected by enforcing a connection timeout. We scrape the Prometheus metrics at each time slot of 0.1 sec with the Beat Worker operating at the same rate. For the training of the ML-based Application Profiles, we used a workload trace acquired from Ferryhopper website[2], which provides ferry booking services around Europe. Figure 4 presents the per-minute distribution of HTTP request rate spanning six days. The data of the first four days were used for the ARIMA model and the ML-based

[1]https://azure.microsoft.com/en-us/pricing/calculator/
[2]https://www.ferryhopper.com/

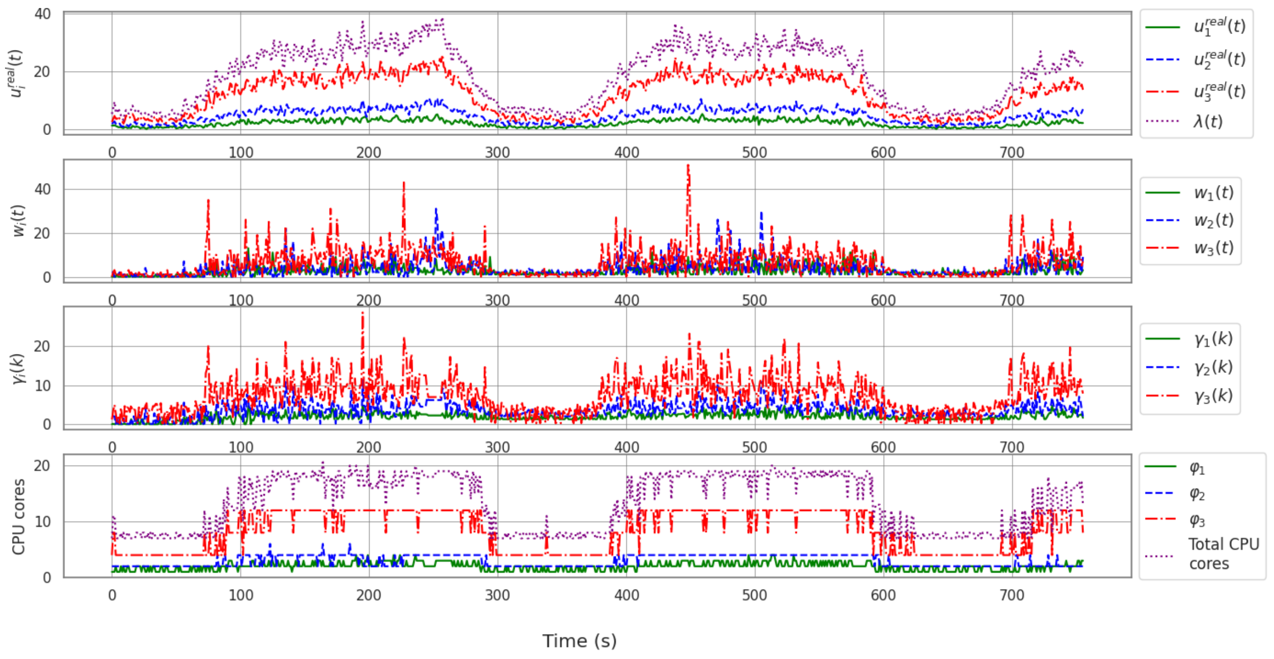

Fig. 5: The performance of the proposed DRA method.

TABLE II: Resource Profile Setting.

Resource Profiles	Small (i=1)	Medium (i=2)	Large (i=3)
CPU cores	1	2	4
RAM (GB)	2	4	8
$[\alpha_i, \beta_i]$	[0.4,0.5]	[1,0.5]	[2.4,0.5]
Max Req Rate (req/s)	1.5	3.6	7.8

Application Profiles, while the data of the last day is used for the evaluation part. The ARIMA model was trained using the autoarima package, which yielded a model of order (3,1,1). Datasets for each resource profile and information for the kNN hypermaters are available in this public repository[3].

We compare the proposed architecture, hereinafter Distributed Resource Autoscaling (DRA), with four different setups, namely (i) modified HPA (m-HPA), (ii) S-HPA (iii) M-HPA, and (iv) L-HPA. The m-HPA, on the one hand, utilizes the three resource profiles by load balancing the incoming requests with a constant ratio. On the other hand, the resource scaling decision is dictated by an HPA instance for each resource profile by targeting 70% of CPU utilization. The last three setups deploy only one resource profile at each time, i.e., only small (S-HPA), only medium (M-HPA), or solely large (L-HPA) resource profiles, respectively. The scaling is, again, performed by HPA by targeting 70% of CPU utilization for the deployed replicas. These setups are all evaluated against a seventy-minutes workload taken from the test data of the Ferryhopper trace. All HPA instances operate every 1sec. The experiments were conducted ten times for each method, and all the results are averaged.

In Figure 5, we illustrate the performance of the proposed DRA method. The first diagram at the top shows the incoming request rate $\lambda(t)$ and the admission rates $u_i^{\text{real}}(t)$, $i = 1, 2, 3$, towards the small, medium, and large resource profile, respectively. at each time slot. As expected, $\lambda(t)$ is distributed according to the AIMD-based scheduling solution (1), while the workload share of each resource profile depends on the ratio $\frac{\alpha_i}{\sum_{i=1}^{n} \alpha_i}$. In particular, the average admission rates of the small, medium and large resource profile, respectively, for the entire experiment, are calculated as 12%, 31% and 57%, which are consistent with the ratio $\frac{\alpha_i}{\sum_{i=1}^{3} \alpha_i}$. The lengths of individual queues of the application pods denoted by $w_i(t)$ are depicted in the second diagram. Evidently, when the workload is peaked, individual queues grow reasonably indicating an increase in local backlog. The AIMD-based service rates $\gamma_i(k)$ for each resource profile, as obtained from Eq. (3), are shown in the third diagram from the top. Recall that $\gamma_i(k)$ is a strictly increasing function of $w_i(t)$ as shown in Eq. (3). This coupling is clearly demonstrated in the second and third diagrams. The ML-based scaling decision is shown at the bottom diagram. The number of deployed CPU cores for each resource profile, is illustrated at each time slot, and the scaling decision takes into account the behavior of all previously shown metrics. The rest features of the ML-based profiles are omitted as they do not add any value to the discussion of the results.

The utilized CPU cores needed to serve the incoming requests at each time slot are illustrated for all methods in Figure 6. The HPA-based solutions, i.e., S-HPA, M-HPA, and L-HPA utilize on average 14.8, 14.6, and 14.1 CPU cores respectively, throughout the experiment. The M-HPA and L-HPA provide negligible QoS violations, namely, 0.2% and 0.9%, respectively. However, both need at least 28 CPU cores to operate at the peak of the workload. This occurs

[3]https://github.com/Dspatharakis/datasets.git

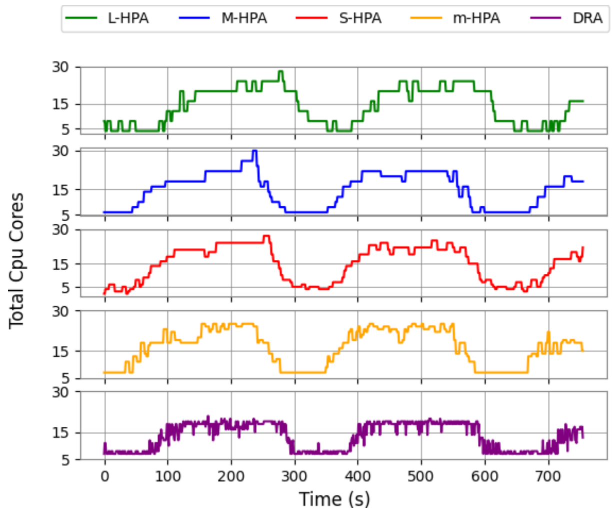

Fig. 6: The Total CPU cores utilized for each method throughout the experiment.

TABLE III: Results for the five experiments.

Setup	Total QoS Violations	Average CPU cores
DRA	1.17%	13.1
m-HPA	0.52%	16.2
S-HPA	3.64%	14.8
M-HPA	0.93%	14.7
L-HPA	0.20%	14.2

because of the lack of a task scheduling algorithm that leads to instantiating under-utilized cores. Nevertheless, for the S-HPA solution, which intuitively could be the most fine-grained scaling solution, we notice that it has on average the same performance in terms of CPU cores, however, leading to a significant increase in lost requests, namely, 3.64% of the total. This is reasonable as the small resource profile can serve only 2.5 requests per second as shown in Table II. For the m-HPA, we can assume that under the assistance of the task scheduling policy, the total QoS violations are minimized, however, the average CPU cores are maximized, leading to 16.1 CPU cores. As one can notice, our method outperforms all other setups by utilizing, on average, 12.6 CPU cores to handle the incoming varying workload, having only 1.8% total QoS violations with more than 8% fewer CPU cores utilized throughout the experiment. It is evident, that the task scheduling mechanism is key to handle the time-varying workload and optimize the utilization of the deployed resources. Table III summarizes the results.

VI. Conclusions and Future Work

This paper presents an architecture for resource management in KEC. The proposed method provides a solution to the task scheduling problem. Also, based on the theoretical results from the proposed AIMD like algorithm and various performance metrics, we introduce an ML-based Application Profiling Modeling that decides the number of replicas for the different resource profiles to proactively and in a decentralized manner, scale the deployed resources to serve the incoming workload. Our framework outperforms other commonly used solutions for autoscaling, as the average CPU resources are at least 7% less, having only a slight increase in QoS violations. Regarding our future plans, we will concentrate on power optimization by minimizing the number of active servers in a KEC with a specific capacity guaranteeing the QoS level using the proposed architecture.

VII. Acknowledgment

This research was supported by the CHIST-ERA-18-SDCDN-003 (DRUID-NET), and has been co- funded by the European Union (European Regional Development Fund -ERDF) and Greek national funds through the Operational Program ”Competitiveness, Entrepreneurship and Innovation 2014-2020” of the National Strategic Reference Framework (NSRF) - Research Funding Program: DRUID-NET (eDge computing ResoUrce allocatIon for Dynamic NETworks), MIS 5070466. We would like to also thank the Ferryhopper team for providing us a realistic workload trace.

References

[1] Kubernetes. https://kubernetes.io/, Last Accessed on 2022-07-01.
[2] 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.
[3] Horizontal-Pod-Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, Last Accessed on 2022-07-01.
[4] C.-H. Hong and B. Varghese, “Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–37, 2019.
[5] D.-M. Chiu and R. Jain, “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks,” *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.
[6] E. Vlahakis, N. Athanasopoulos, and S. McLoone, “Aimd scheduling and resource allocation in distributed computing systems,” in *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 4642–4647, 2021.
[7] A. Shahidinejad, M. Ghobaei-Arani, and M. Masdari, “Resource provisioning using workload clustering in cloud computing environment: a hybrid approach,” *Cluster Computing*, vol. 24, no. 1, pp. 319–342, 2021.
[8] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian, “Resource management approaches in fog computing: a comprehensive review,” *Journal of Grid Computing*, vol. 18, no. 1, pp. 1–42, 2020.
[9] S. Verma and A. Bala, “Auto-scaling techniques for iot-based cloud applications: a review,” *Cluster Computing*, vol. 24, no. 3, pp. 2425–2459, 2021.
[10] Prometheus. https://prometheus.io/, Last Accessed on 2022-07-01.
[11] Custom-Pod-Autoscaler. https://custom-pod-autoscaler.readthedocs.io, Last Accessed on 2022-07-01.
[12] W. Ren, E. Vlahakis, N. Athanasopoulos, and R. Jungers, “Optimal resource scheduling and allocation in distributed computing systems,” *arXiv preprint arXiv:2112.00708*, 2021.
[13] M. Corless, C. King, R. Shorten, and F. Wirth, *AIMD dynamics and distributed resource allocation*. SIAM, 2016.
[14] 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.
[15] 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.
[16] K. Taunk, S. De, S. Verma, and A. Swetapadma, “A brief review of nearest neighbor algorithm for learning and classification,” in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, pp. 1255–1260, 2019.