XAI-Enabled Fine Granular Vertical Resources Autoscaler

Mohamed Mekki
Student, IEEE,
Department of Communication Systems
EURECOM
Sophia Antipolis, France
mohamed.mekki@eurecom.fr

Bouziane Brik
Senior Member, IEEE,
University of Burgundy
Dijon, France
bouziane.brik@u-bourgogne.fr

Adlen Ksentini
Senior Member, IEEE,
Department of Communication Systems
EURECOM
Sophia Antipolis, France
adlen.ksentini@eurecom.fr

Christos Verikoukis
Senior Member, IEEE,
University of Patras and Iquadrat Informatica, ISI/ATHENA
Patras, Greece
cveri@upatras.gr

***Abstract*—Fine-granular management of cloud-native computing resources is one of the key features sought by cloud and edge operators. It consists in giving the exact amount of computing resources needed by a microservice to avoid resource over-provisioning, which is, by default, the adopted solution to prevent service degradation. Fine-granular resource management guarantees better computing resource usage, which is critical to reducing energy consumption and resource wastage (vital in edge computing). In this paper, we propose a novel Zero-touch management (ZSM) framework featuring a fine-granular computing resource scaler in a cloud-native environment. The proposed scaler algorithm uses Artificial Intelligence (AI)/Machine Learning (ML) models to predict microservice performances; if a service degradation is detected, then a root-cause analysis is conducted using eXplainable AI (XAI). Based on the XAI output, the proposed framework scales only the needed (exact amount) resources (i.e., CPU or memory) to overcome the service degradation. The proposed framework and resource scheduler have been implemented on top of a cloud-native platform based on the well-known Kubernetes tool. The obtained results clearly indicate that the proposed scheduler with lesser resources achieves the same service quality as the default scheduler of Kubernetes.**

***Index Terms*—Zero-touch Service Management (ZSM), Cloud-native, Containerized Microservices; Machine Learning; Explainable Artificial Intelligence.**

I. Introduction

The cloud-native concept was born with the advent and the success of cloud resource virtualization (computing, storage, and networking) relying on container technology [1]. In a cloud-native architecture, cloud applications and services are no longer deployed as monolithic blocks but rather as loosely coupled microservices. Each microservice is deployed as a container and managed using container orchestration engines and platforms like Kubernetes. In contrast to the traditional monolithic model, running applications as containerized microservices permits more agility and flexibility by facilitating development, upgrades, maintenance, and hence DevOps operations. However, running microservices in a cloud-native architecture opens the door for new challenges when managing both the application life-cycle and cloud resources [2] [3]. One of the critical challenges is scaling the microservices resources under dynamic workload variations and resource demands. Indeed, before deploying users' applications or services, cloud operators identify the number of virtual instances needed during the execution of the workload as well as the required resources for each application instance, such as the type and number of virtual machines or the number of pod replicas and resources needed for each pod in the Kubernetes cluster. It is well-accepted that estimating the needed resources of an application, i.e., the adequate combination of memory, CPU, and the number of concurrent instances, to avoid service degradation is a challenging task. The number of replicas and the needed computing resources to optimally handle a given application may vary over time due, for instance, to the time period or the application's popularity (i.e., an increase in popularity). Accordingly, it is important to design an intelligent and efficient management system that, throughout the life-cycle of a microservice, computes the needed resources of microservices to run optimally and avoid service degradation.

One of the key functions of the cloud-native management system is the scaler algorithm. The latter's role is to compute: (1) the needed application instances or virtual instances, known as "horizontal scaling"; or (2) the amount of computing resources per instance, known as "vertical scaling". Several solutions have been proposed to devise intelligent and autonomous scaler solutions, which largely leverage machine learning (ML) algorithms and, in particular, Reinforcement Learning (RL) [4] [5]. These solutions mainly learn microservices' load patterns as well as traffic changes and generate the learning models able to predict each microservice needs and scale the vertical resources. However, the scaling approach triggered after the ML algorithm prediction consists of simply doubling or multiplying by a factor the dedicated CPU and memory assigned to the microservice [6]. Scaling the CPU and memory is not optimal, as not all applications are simultaneously sensitive to CPU and memory. Some microservices are sensitive to CPU or memory, while only a few are sensitive to both [7]. Therefore, scaling both computing resources may negatively impact resource usage (waste of resources) and hence on other critical metrics, particularly energy consumption.

In this paper, we propose a Zero-touch Service Management

(ZSM) framework featuring a fine-granular resource scaler algorithm to run microservices in a cloud-native environment optimally. The proposed scaler algorithm relies on ML models to predict the performances of the run microservice. When a service degradation is detected, eXplainable Artificial Intelligence (XAI) algorithms are used to interpret the ML prediction and deduce which features led to that bad performance. More specifically, our framework relies first on an ML algorithm based on eXtreme Gradient Boosting (XGBoost) [8] to predict any violations related to the performance of running applications. Here, we use the application response time metric to characterize the application performance. The trained ML model considers many features related to CPU and memory, namely CPU usage, CPU limit, memory usage, and memory limit. Parallelly, an XAI algorithm is run, namely SHapley Additive exPlanations (SHAP) [9], to deduce the most important features that yield such violation using ML outputs. By knowing the root cause of the performance violation, the autoscaler algorithm scales the CPU, memory, or both. Regarding the scale-down process, we consider a threshold-based approach for CPU and memory, in which a scale-down is possible. But, in order to avoid a ping-pong effect (repetitive scale down and scale up), we also consider a stabilization period after a scale-up where a scale-down process is not allowed. The proposed vertical autoscaling framework can be combined with existing horizontal scaling mechanisms in order to achieve both vertical and horizontal resources autoscaling.

The rest of this paper is structured as follows. Section II provides a discussion of the related work. Section III details the design and components of the XAI-based framework, focusing on the proposed autoscaler algorithm. Section IV presents the performance evaluation of our framework. Finally, section V concludes the paper.

II. Related Work

In this section, we briefly overview some of the most significant works on ML-based resource autoscaling in cloud-native systems.

In [10], the authors designed Autopilot as horizontal and vertical autoscaling of resources at Google. It mainly aims to minimize slack, i.e., the difference between capacity and real-time resource usage, while ensuring the stable performance of running tasks. Autopilot leveraged machine learning algorithms on top of historical data related to tasks/jobs executions. In particular, Autopilot relies on two main algorithms. The first one enables an exponentially-smoothed sliding window, while the second one is based on reinforcement learning (RL) to select the suitable sliding window algorithm, which gives better performance for each task/job. Practical results show that Autopilot succeeds in reducing, on one hand, the slack to 23%, against 46% for manually-scaled tasks; on the other hand, the number of tasks impacted by out-of-memory by a factor of 10. The authors in [2] proposed a model-based RL scheme to enable both horizontal and vertical auto-scaling mechanisms of container-based applications. The designed model considers several criteria in its auto-scaling, including application performance, resource cost, and adaptation cost. Furthermore, the proposed scheme is integrated into Docker Swarm to realize an Elastic Docker Swarm (EDS). Obtained results show the efficiency and flexibility of EDS in leveraging RL with respect to other existing elasticity policies. The challenge of ensuring end-to-end Service Level Agreement (SLA) while improving resource allocation to microservices is addressed in [3]. The authors proposed a novel SLA-Aware scheme based on Bayesian Optimization to assign necessary resources to meet applications' performance. This scheme is evaluated on top of a real microservice workload, where the results clearly demonstrate the ability to meet the requirements of each microservice and find Pareto-optimal solutions. In the same context, a smart autoscaling system for cloud microservices-based applications considering applications' constraints has been introduced in [11]. The proposed autoscaler comprises two main modules: (i) the first one monitors the microservices requirements regarding resources through a generic autoscaling scheme integrated into the Google Kubernetes Engine. This module auto-scales Kubernetes with respect to the running application needs; (ii) based on the resource requirements and applications QoS, the second one leverages RL and deploys a set of agents to learn and determine the autoscaling thresholds concerning resource utilization and the maximum number of pods.

The above works mainly leverage deep learning (DL) algorithms, in particular reinforcement learning, to optimize the horizontal/vertical autoscaling of cloud applications. However, the main drawback of such works is the black-box deployment of DL/RL-based models. Specifically, DL/RL models are becoming more and more complex, i.e., it is hard to understand their inner workings, especially by non-expert users. Moreover, the DL/RL models give predictions/decisions about scaling up/down without any interpretations or explanations on how and why such outputs are made. Hence, the corresponding users (or container orchestration tools) can neither trust and understand DL/RL models' outputs nor optimize their decisions with respect to DL/RL model outputs. To overcome these limits, we leverage the emerging explainable AI paradigm to design a novel vertical autoscaling framework. XAI enables not only interpreting and explaining predictions made by ML models but also helps in making suitable decisions, e.g., scaling up or down CPU, memory, or both, based on the provided explanations. To the best of our knowledge, this is the first work that combines ML and XAI models to design a vertical and explainable autoscaling framework.

III. Design and specification of the proposed autoscaler framework

A. The envisioned ZSM architecture

To achieve the concept of a fine-granular vertical autoscaler in a cloud-native environment, we propose a novel ZSM framework that combines both ML and XAI. The proposed ZSM framework encloses a closed-control loop that monitors, analyses, and derives appropriate life-cycle decisions regarding

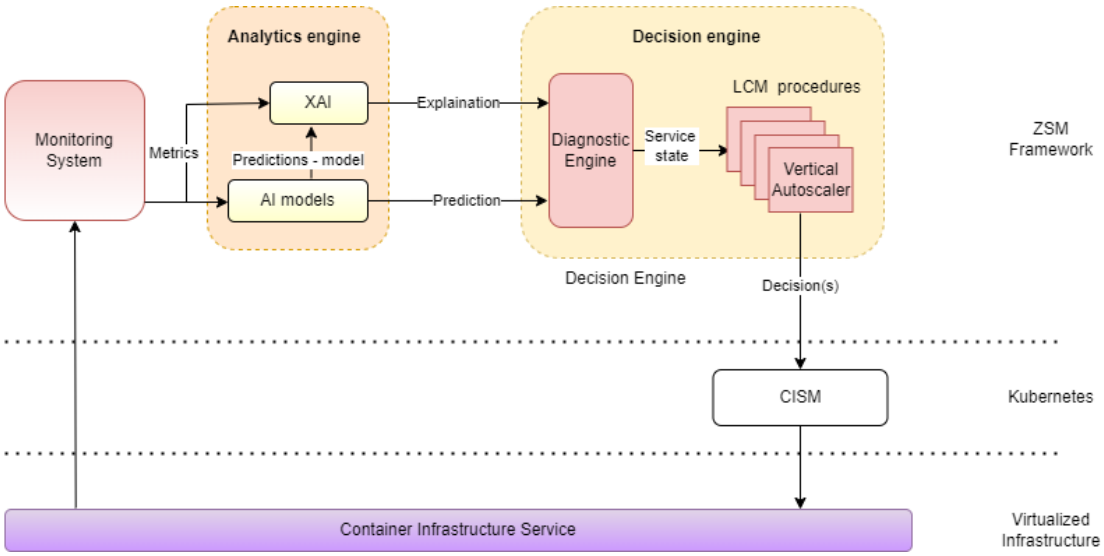


Fig. 1. Zero touch network framework architecture

a microservice, mainly the vertical scaling of computing resources. The component in charge of the vertical scaling (noted vertical scaler) is part of the decision-making function of the closed-control loop. The latter runs autonomously without external intervention and follows the same design of the closed-control loop introduced in [12]; i.e., composed of three key components: (1) Monitoring System (MS) that collects Key Performance Indicator (KPI) regarding the performance of running microservices, such as CPU and memory consumption. (2) Analytical Engine (AE) that uses the collected KPI by MS to analyze the microservice performance behavior and detect Quality of Service (QoS) degradation. To run the analysis, AE may rely on an ML algorithm, which is, in our case, based on XGBoost, to predict the latency performance of a microservice. In contrast to [12], our AE runs, in parallel, the XAI algorithm to interpret the ML output. (3) Decision Engine (DE) runs the life-cycle decision-making process to overcome service degradation. It relies on AE analysis and takes the ML prediction (QoS performance) and its explanation as input. In our solution, DE contains the vertical autoscaler algorithm that scales up resources when a service degradation is detected. The closed-control loop system enables the automatic scaling of vertical microservice resources.

Fig. 1 illustrates a generic architecture of the proposed ZSM framework. We assume that all microservices run in a cloud-native environment. The figure separates between the closed-control loop components described in the preceding paragraph and the virtualized infrastructure and its manager. The latter is known as the assisted system based on ETSI Experiential Networked Intelligence (ENI) group's notation [13]. According to ETSI cloud-native report [14], the cloud-native equivalent of a hypervisor is Container Infrastructure Service (CIS), which provides all the runtime infrastructural dependencies for one or more container virtualization technologies. In contrast, Container Infrastructure Service Management (CISM) is a cloud-native equivalent of Virtualized Infrastructure Manager (VIM). Technologically speaking, CSIM may correspond to Kubernetes. Regarding the closed-control loop, MS monitors the KPI from CIS regarding the container's resource usage, such as CPU and memory consumption. In our case, we extracted information regarding computing resource consumption (CPU, memory) that AE will use to predict the performance of the microservice at the service level. Here, we are interested in predicting QoS as perceived by the end-users. In the context of a web server, the metric reflecting the QoS can be the response time, i.e., the time a web server takes to answer a client request. Usually, high service time means the server is overloaded and cannot handle the requests in a bounded time, hence degrading the user's quality of experience. AE runs the trained ML model along with the XAI algorithm to predict whether the response time corresponds to service degradation. The XAI module uses both the collected KPI as well as the ML prediction to provide an explanation. Both the explanation and the prediction are transmitted to DE and, more precisely, to the Diagnostic Engine module (Fig. 1). The latter uses the output of the AI model responsible for detecting whether the application response time is appropriate or not. It also receives explanations from the XAI module about inference. The explanation gives the contribution of the features to the model output, which means that if the model detects a high response time occurrence (i.e., QoS degradation), the XAI output indicates the contribution of the application's resource features in this result. These characteristics are related to either CPU usage or memory usage. The diagnostic engine then detects the element that caused the high response time. This information is then passed to the vertical autoscaler algorithm, which makes a decision on which resource to scale, hence performing a fine-granular scaling rather than blindly scaling both CPU and memory. It is worth noting that the autoscaler decision is enforced using the northbound API

exposed by CISM that allows updating the resources dedicated to the container running the application by modifying the application's controller object of Kubernetes. Afterward, the Kubernetes controller will rollout a new instance with the new resources definition and delete the old instance.

Regarding the scaling down process, we use a stabilization period after a scale-up in which scaling down will not be performed in order to avoid resource scaling oscillations, i.e., the autoscaler performs one action and, after a short period, performs the opposite action. If no performance drop has been detected during the last few seconds of the stabilization period, a scaling down is possible. To perform this operation, we rely on historical data on resource usage. For memory, if during the last stabilization period, the maximum memory usage was under a chosen percentage, then a scaling down of memory resources is possible. For CPU, if the mean CPU usage during the last stabilization period was less than a certain percentage, then a scale down of CPU resources can be performed. It is worth noting that the scaling down process has no impact on the granularity of the resource allocation. Indeed, when scaling down an application, the resources are released and can be used by another running application instance.

B. Analytical Engine (AE)

The analytical engine is responsible for analyzing the microservices' performance and detecting QoS degradation. It is composed of two main components: (1) the AI model, which is based on XGBoost, to predict the latency performance of a microservice. (2) The XAI model interprets the output of the AI model using SHAP. Before describing the functioning of these components, we describe the data generation process where we perform a benchmarking of different types of applications.

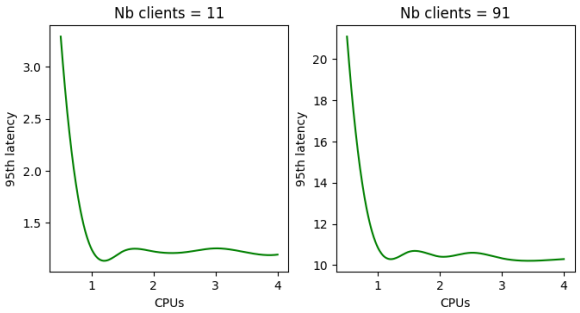


Fig. 2. Web Server's latency in relation to the allocated CPU

1) Data generation: In order to understand the behaviour of microservices in cloud-native environments, we built a dataset containing information about different applications' resource usage and performance, including web servers, data brokers, and 5G Core Network functions. The performance of the applications is measured using the response time to the client requests. Each tuple of the dataset contains the following information: the memory and CPU allocated to the workload, the memory, and CPU used by the application, the application

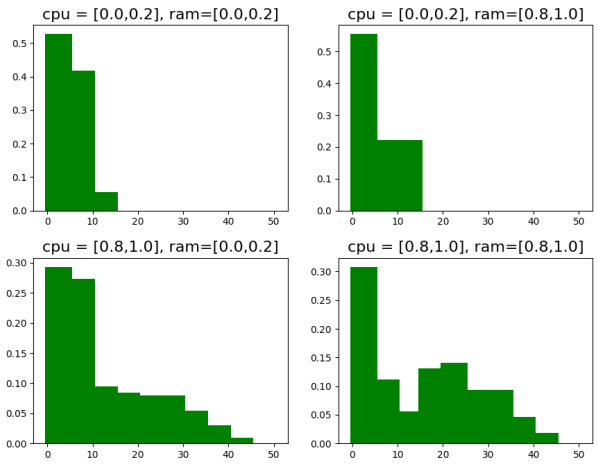


Fig. 3. Web Server's latency statistical distribution

response time, and the load on the application. The latter is measured by the number of concurrent requests received by the application during an interval of time. Fig. 2 shows the response time of the webserver in relation to the CPU allocated, we can notice that under different loads, represented by the number of concurrent clients sending requests to the server, the more CPU the application has, the lower the response latency is. Moreover, in Fig. 3 we show the distribution of the response time of the webserver. We consider the relative CPU and memory, which represent the percentage of resources used from the provided limit. We notice that the higher the relative CPU is, by comparing the distribution while the relative CPU is between 0 and 0.2 and between 0.8 and 1, the greater the percentage of high response times is. In contrast, the memory percentage does not change the distribution of latency values. More information about the dataset collection and analysis is available in [7], while the complete dataset is available in [15].

2) ML training: Considering the collected dataset, we can observe that the resources allocated to the application and the relative usage of resources are related to the performance of the application. First, the allocated resources show the limit of performance; the application with fewer available resources will perform worse. Second, the relative resource utilization indicates the possibility of the occurrence of high response times, which means that the degradation of the application performance is more likely to occur when the resource utilization approaches the limit allocated to the application. Therefore, we implement an ML model using the XGBoost classifier to detect performance deteriorations of the application. The model uses resource usage and limits information which can be collected on the running applications via the MS.

XGBoost is a scalable ML system for tree boosting. It implements the gradient-boosted trees algorithm, a supervised learning algorithm that can be used for regression or classification tasks. We train the XGBoost classifier to detect the application's performance drop based on resource usage patterns.

To train the XGBoost classifier on the web server's dataset, we label the dataset's lines as QoS respected or QoS not respected when the response time is lower or higher than a threshold, respectively. Finally, the model gets the following information as input: memory limit, memory usage, CPU limit, CPU usage, relative CPU, and relative memory. Those metrics can be collected for all the running workloads via MS during runtime. Based on the label and the resource usage, the model classifies the performance of the application, using the resource consumption of the workload, into respecting QoS or not based on the resource usage and limit.

During training, we compared several classification algorithms: K-Nearest Neighbors classifier, Artificial neural network classifier, logistic regression, Random forests, and XGBoost classifier. The XGBoost model was selected based on the classification report by comparing the precision and recall for class 0, which represents the performance degradation of the service. The model's accuracy was 0.95, and the precision and recall for both classes (0 for Qos not respected and 1 for QoS respected) were respectively 0.86, 0.74 for class 0, and 0.97, 0.99 for class 1.

3) XAI: The second element of the analytical engine is the XAI module, which is responsible for interpreting the output of the AI model. Several XAI techniques exist and can be classified into global or local explanation techniques. Global explanation techniques, such as SHAP, are applied to obtain the general behavior of a model by attempting to explain the whole logic of a model by inspecting its structure. On the other hand, local explanation techniques, such as SHAP and LIME [16], tackle explainability by segmenting the solution space and giving explanations to less complex solution subspaces that are relevant to the whole model. These explanations can be formed through techniques with the differentiating property that only explain part of the whole system's functioning.

The XAI module of the AE relies on the local explanation method based on SHAP to compute the scores of the features contributing to the model's output. The module's output is the contribution score values of the features to the output. Fig. 4 represents a visualization of an output of the SHAP method for an ML prediction. The negative values indicate that the feature pushes the model's output towards the output 0. while the positive values signify that the feature pushes the output of the model towards the positive output 1.

For this inference, the XAI module reports the following Shapley values or scores of the features, ordered by decreasing contribution to the model output: CPU_percentage -2.56, meaning that the CPU_percentage value pushed the model towards the output 0 (SLA not respected) with a score of 2.56, the second affecting feature is RAM_limit with a score of -1.81, the next contributing feature is RAM_usage with a score of +0.99 meaning that this feature pushed the model towards the output 1 (SLA respected) with a score of 0.99; for the less impacting features, the XAI module indicates CPU_usage +0.71, RAM_percentage -0.38, and CPU_limit -0.13.

Afterward, the selection of the resources to scale up is made at the DE level. This is done by comparing the weighted sum of the contribution of the features related to CPU resources with the weighted sum of the contribution of resources related to memory resources. We can deduce from the previous scores that the combined score of CPU-related features is -1.98 and memory -1.2, meaning that CPU-related features have more influence on the model decision. Therefore, the DE decides that the cause of the performance drop is insufficient CPU allocation. This information allows the vertical autoscaler to decide to allocate more CPU resources to the workload.

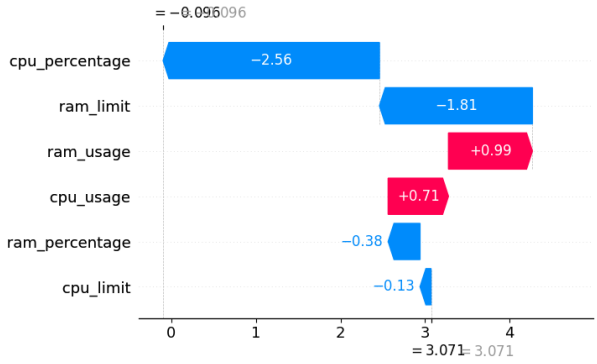


Fig. 4. Shapley values provided by the SHAP method

IV. Performance Evaluation

In this section, we present the results of validating the ZSM components (i.e., MS, AE, and DE) and testing the XAI-based vertical auto scaling framework. For the sake of comparison, we implemented two versions of the vertical scaler. The first one includes the XAI module, whereas the second implementation does not. Next, we provide metrics on the efficiency of autoscaling, i.e., the amount of resources allocated to the workload regarding the performance achieved by the workload. We demonstrate the benefits of introducing XAI into resource management both at the application level by achieving better performance and at the infrastructure level by reducing resource usage, achieving the goals of both the service owner and the network provider.

A. Testing Environment

The test facility includes a Kubernetes cluster, which is deployed on top of an Intel server PowerEdge T440 with 128GB of RAM and 64 Core (Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz) with hyper-threading enabled. The cluster was bootstrapped using Kubeadm v1.21.1, and the host operating system is Ubuntu 18.04.5. All the framework components and the tested application run as containers in the cluster. The cluster has a Prometheus deployment for pods and node metrics collection used by MS to collect KPI regarding the infrastructure (i.e., CPU and memory usage of applications).

During the performance evaluation, we run a web server as the target application to be scaled. The application instances are deployed on the Kubernetes cluster as pods with an initial resource configuration of 64MB of memory and 0.25 CPU core.

B. Performance results

For the tests, we use two versions of the vertical autoscaler, an XAI-based autoscaler to vertically scale the web server instances resources and one without using the XAI output. The running application is exposed to requests load produced by a test component that uses ApacheBench[1] to make a number of concurrent HTTP requests. We refer to a test round as a set of N requests made by C concurrent clients.

It is worth recalling that AE runs the ML model to predict QoS degradation. The latter was trained on the dataset related to the performance of web servers under changing configurations. If the model detects degradation, the XAI module is called. The XAI takes as input both the ML output as well as the data set to return the Shapley (or score) values of the features as a numerical score. Then the diagnostic engine of DE compares the weighted sum of the memory-related features scores with the weighted sum of the CPU-related features scores. This output will allow the autoscaler to decide what type of resources need to be scaled.

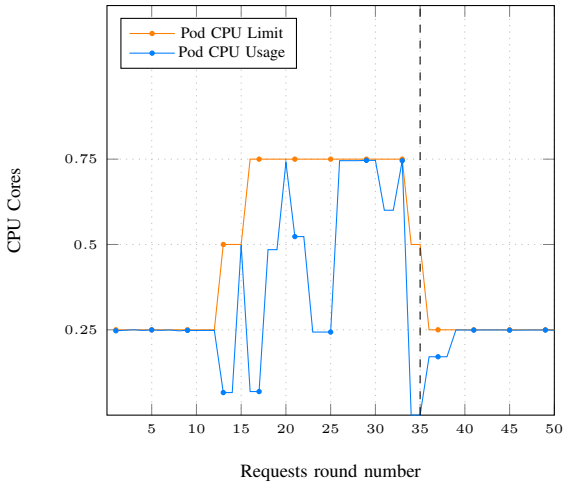
In case the XAI module is not involved, the autoscaler obtains information about the service's state using only the ML module's output (XGBoost); it has no information about the contribution of the features to the model output. If degradation is detected, both CPU and memory resources are scaled.

1) Application's performance evolution: In this test, we run a web application with an initial configuration of 0.25 CPU core and 64 MB of memory. We perform 35 rounds of requests to the application with a load that uses a concurrency level of 50 clients making 250 total requests, followed by 15 rounds of requests with a concurrency level of 10 clients making 200 requests in total. Parallelly, we note the vertical autoscaling decisions (i.e., scale down or up) and measure the application's response time and resource utilization during each round of requests.

Figs. 5 and 6 show the results of the two key metrics regarding resource usage, CPU and RAM, obtained when the autoscaler decision is taken with and without the help of XAI, respectively. Both figures represent the evolution of the PoD's CPU_limit, CPU_usage, RAM_limit, and RAM_usage according to the request round number. The vertical dashed line after round 35 indicates a reduced load on the application. Regarding the case of the autoscaler using XAI (Fig. 5) and based on the response time, we observe both changes in the resources allocated to the application and their effect on its performance shown in Fig. 7. The application initially receives the first twelve rounds of requests, after which, based on the metrics on pod resource usage that are collected by the monitoring system, the ML module detects that the application performances degrade (i.e., an increase in the response time). For this ML prediction (i.e., inference), the XAI module reports the Shapley values or scores of the features shown in Fig. 4. As described in section III-B3, the DE concludes that insufficient CPU allocation is the cause of the performance drop. Consequently, the vertical autoscaler decides to allocate

[1]https://httpd.apache.org/

(a) CPU limit and CPU used by the pod in relation to the round number



(b) Memory limit and memory used by the pod in relation to the round number

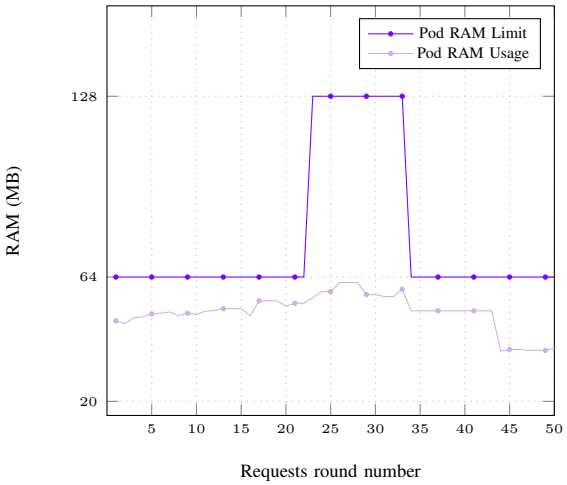
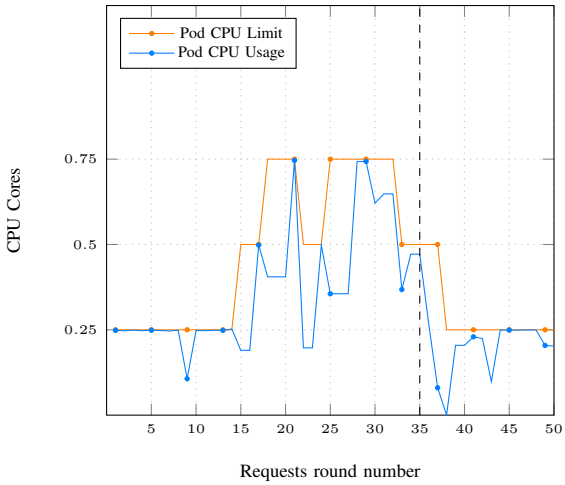


Fig. 5. Evolution of CPU and allocated RAM using the XAI-assisted autoscaler

more CPU resources to the workload. This operation increases the CPU limit for the application; it can be observed in the metrics of round 13 as we notice in Fig. 5.a that the amount of CPU allocated to the application is 0.5 Core instead of the previous 0.25 Core. Similarly, we notice changes in the resources allocated to the pod at round 15, as the CPU has increased from 0.5 Core to 0.75 Core, and at round 22 from Fig. 6.b, the memory allocated to the application has increased from 64MB to 128MB. At these same points, we notice the effect of the autoscaler decisions on the application response time. It drops from 20 seconds to about 4 seconds after round 16.

After round 35, the load on the application is reduced to a concurrency level of 10 clients. We observe that the vertical autoscaler took decisions to scale down both CPU and memory

(a) CPU limit and CPU used by the pod in relation to the round number



(b) Memory limit and memory used by the pod in relation to the round number

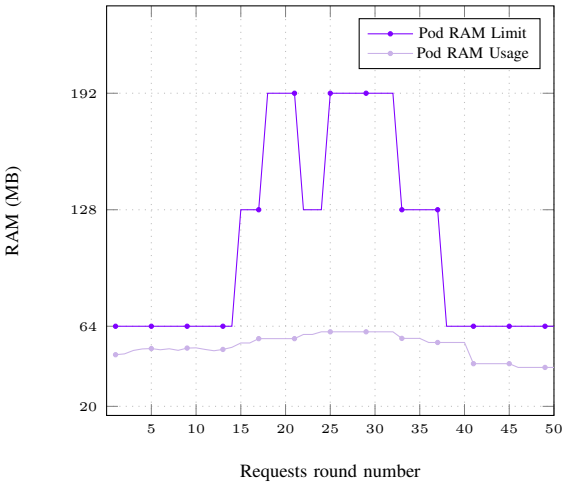


Fig. 6. Evolution of CPU and allocated RAM using the autoscaler without XAI module's assistance

at (1) round 34, the CPU allocated to the container is reduced to 0.5 Core, and the memory is reduced to 64 MB; (2) round 37, the CPU allocated to the container is decreased to 0.25 Core. We can also remark, from Fig. 7, that the application's response time remains constant after round 35.

When the diagnostic engine has no access to the XAI module output (i.e., no information about the contribution of the features to the model output) (Fig. 6), the cause detection at this level is less granular, which leads the autoscaler's decision to be less specific to the type of resource (CPU or memory); hence both resources are increased.

In this scenario, and by comparing the response time of the application in both cases, shown in Fig. 7, we can conclude that the introduction of the XAI module allowed the vertical autoscaler to provide lesser resources to the application (less

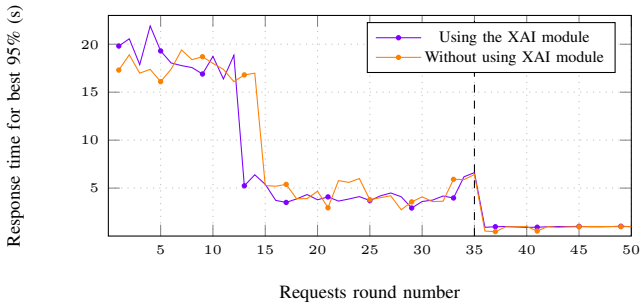


Fig. 7. Evolution of the application performance (response time) using the autoscaler with and without XAI module's assistance

memory) to achieve the same level of performance (a response time of around 4 seconds).

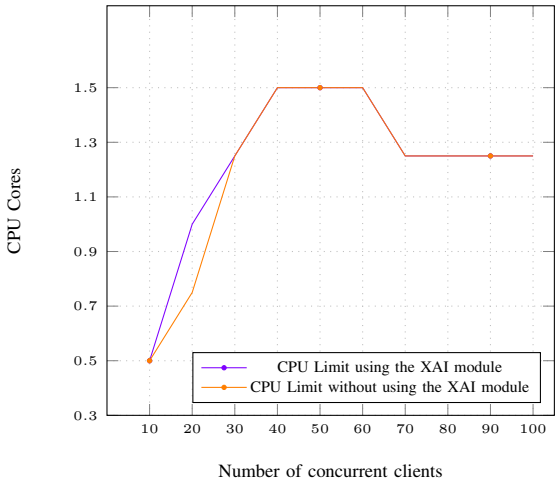
2) Testing multiple instances while varying the load on the auto-scaled applications: This time we perform an extensive test regarding the performances of the two vertical autoscalers. Hence, we deploy 30 applications and we vary the load to which each application is exposed. The application is first deployed with an initial resource configuration of 0.5 Core of CPU and 128MB of memory. The number of concurrent clients sending requests to each application varies from 10 to 100, while the number of requests varies from 90 when using 10 concurrent clients to 450 when a concurrency level of 100 is used. Finally, we perform 100 rounds for each concurrency level. For each application, the pod configuration is reinitialized afterward. Resulting in a total of 300 application instances to be scaled by each autoscaler (30 services exposed to a load varying from 10 to 100).

Fig. 8 shows the highest amount of CPU allocated to the pod running the application for each load (Fig.8.a) and the highest amount of memory allocated to the pod (Fig.8.b) for both vertical autoscalers. Whereas, Fig. 9 illustrates the mean response time of the 30 applications for the 100 rounds of requests that each scaled application receives.

During the experimentation, the 300 instances that have been scaled using the XAI-based autoscaler employed a total of 184.25 cores of CPU and 29.312 GB of memory, while the 300 instances that have been scaled using the non-XAI-based autoscaler used a total of 188.0 core of CPU and 48.128 GB of memory. Thus, the percentage of memory gained is 39%, while the percentage of CPU resources gained is 1%.

By comparing the decisions of the two vertical autoscalers we observe that the XAI-based one allocates less memory to the application for all amounts of load. In contrast, it allocates the same or more CPU than the non-XAI-based autoscaler. These results clearly show the fine granularity of the resource allocation achieved by the XAI-based autoscaler, thanks to the ability of the latter to determine the factors that led to performance degradation. Moreover, from the response time plot, we observe that the mean response time of the applications while being managed by both vertical autoscalers is approximately equal, meaning that the allocation of lower resources by the XAI-based autoscaler did not affect the

(a) CPU allocated to the workload in relation to the load on the service



(b) Memory allocated to the workload in relation to the load on the service

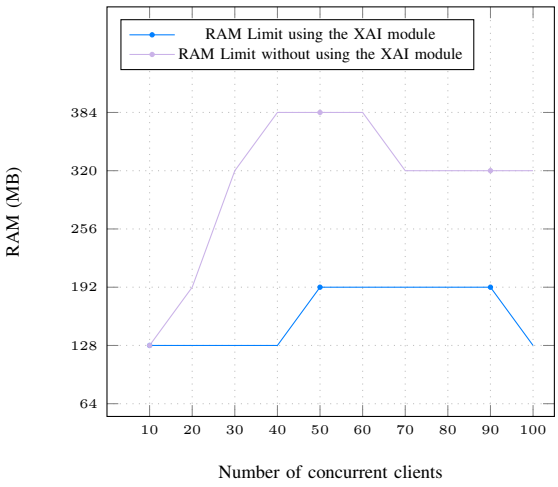


Fig. 8. Highest values of CPU and RAM allocated to an instance of the applications in relation to the number of concurrent clients

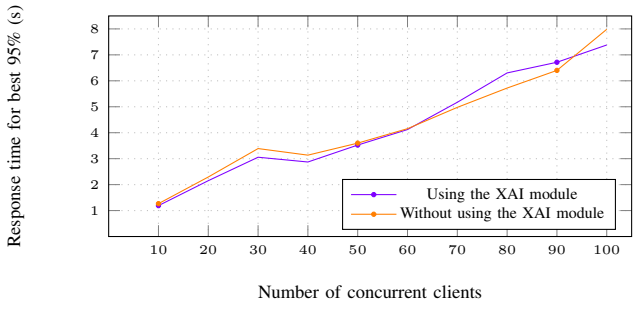


Fig. 9. Mean response time in relation to the number of concurrent clients

applications' performances.

V. Conclusion

In this paper, we introduced a ZSM framework featuring ML and XAI to achieve fine-granular resource management in a cloud-native environment. The proposed framework relies on a novel closed-control loop to ensure vertical scaling of application resources (i.e., computing). Unlike the existing solutions, the proposed closed-control loop combines ML and XAI to detect service degradation and select the appropriate resource to scale up instead of scaling all types of computing resources. All the proposed framework components (including the closed-control loop) have been implemented on top of Kubernetes, where the focus was to evaluate the vertical scaler that relies on XAI using, as an example, a web server. The obtained results showed the benefit of introducing XAI in resource autoscaling, allowing the latter to achieve fine-granular resource allocation. Indeed, for the same performance (same response time), XAI-based autoscaler allows using lesser computing resources (CPU, memory). Therefore, knowing the root cause of application degradation via XAI enable more efficient use of the resources available in the infrastructure.

Acknowledgment

This work was partially supported by the European Union's Horizon 2020 Research and Innovation Program under the AC3 project (Grant No. 101093129).

References

[1] P. A. Frangoudis, L. Yala, A. Ksentini, and T. Taleb, "An architecture for on-demand service deployment over a telco CDN," in *2016 IEEE International Conference on Communications, ICC 2016, Kuala Lumpur, Malaysia, May 22-27, 2016*. IEEE, 2016, pp. 1–6.

[2] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338.

[3] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "Rambo: Resource allocation for microservices using bayesian optimization," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 46–49, 2021.

[4] S. Venkateswaran and S. Sarkar, "Fitness-aware containerization service leveraging machine learning," *IEEE Transactions on Services Computing*, vol. 14, no. 6, pp. 1751–1764, 2021.

[5] M. Hamilton, N. Gonsalves, C. Lee, A. Raman, B. Walsh, S. Prasad, D. Banda, L. Zhang, L. Zhang, and W. T. Freeman, "Large-scale intelligent microservices," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 298–309.

[6] I. Alawe, A. Ksentini, Y. H. Aoul, and P. Bertin, "Improving traffic forecasting for 5g core network scalability: A machine learning approach," *IEEE Netw.*, vol. 32, no. 6, pp. 42–49, 2018.

[7] T. N. Mekki, Mohamed and A. Ksentini, "Microservices configurations and the impact on the performance in cloud native environments," in *Accepted in IEEE Local Networks Conference*, ser. LCN '22, 2022. [Online]. Available: https://www.eurecom.fr/index.php/fr/publication/6971

[8] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[9] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[10] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[11] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35 464–35 476, 2021.

[12] H. Chergui, A. Ksentini, L. Blanco, and C. V. Verikoukis, "Toward zero-touch management and orchestration of massive deployment of network slices in 6g," *IEEE Wirel. Commun.*, vol. 29, no. 1, pp. 86–93, 2022.

[13] *ENI Vision: Improved Network Experience using Experiential Networked Intelligence, ETSI ENI White paper*.

[14] *Network Functions Virtualisation (NFV) Release 3; Architecture; "Report on the Enhancements of the NFV architecture towards Cloud-native and PaaS", ETSI GR NFV-IFA 029 V3.3.1, Nov. 2019*.

[15] *dataset: Benchmarking on Microservices Configurations and the Impact on the Performance in Cloud Native Environments*. [Online]. Available: https://zenodo.org/record/6907619#.YvDXKOxBxaR

[16] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.