Reactive Autoscaling of Kubernetes Nodes

Tarek Menouer, Christophe Cérin, Patrice Darmon

To cite this version:

Tarek Menouer, Christophe Cérin, Patrice Darmon. Reactive Autoscaling of Kubernetes Nodes. FRAME - 4th workshop on Flexible Resource and Application Management on the Edge, Massimo Coppola, Hanna Kavalionak, Ioannis Kontopoulos, Luca Ferrucci, Jun 2024, Pisa (IT), Italy. hal-04857321

HAL Id: hal-04857321
https://hal.science/hal-04857321v1

Submitted on 28 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

Reactive Autoscaling of Kubernetes Nodes

Tarek MENOUER*
CGI - Le Carré Michelet
10-12 cours Michelet
Puteaux, 92800
France
tarek.menouer@cgi.com

Christophe Cérin†
INRIA DATAMOVE & Université
Sorbonne Paris Nord
LIPN - UMR CNRS 7030
Villetaneuse, 93430
France
christophe.cerin@univ-paris13.fr

Patrice Darmon*
CGI - Le Carré Michelet
10-12 cours Michelet
Puteaux, 92800
France
patrice.darmon@cgi.com

ABSTRACT

Kubernetes is undoubtedly the most effective container orchestration system that automates container management with high scalability. It allows for running containerized applications on a Kubernetes cluster composed of a set of computing nodes. According to the native Kubernetes operating mode, all nodes in the cluster are used. This massive use of computing resources can lead to resource waste. To address this limitation, we present in this paper a new reactive Kubernetes autoscaler mechanism that allows the number of active computing nodes in a Kubernetes cluster to be controlled dynamically based on several factors. The goal is to reduce resource waste, energy consumption, and the cost of renting a Kubernetes cluster. The idea is to have a pilot that dynamically checks the state of the Kubernetes cluster and scales up or down computing nodes. To select the most pertinent node to add or remove from the Kubernetes cluster, a multi-criteria decision-making (MCDM) algorithm is used. The proposed autoscaler mechanism is offered on top of the Kubernetes framework with minimal changes to make it easy to use with future versions of Kubernetes. Experiments have demonstrated the effectiveness of our solution in different scenarios. The package we provided for the experiments is generic and ready for current Kubernetes flavors.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Efficient orchestration and Resource management for the Cloud, Autoscaling, Container technology, Energy consumption, Kubernetes ecosystem

1 INTRODUCTION

Containerization is a lightweight virtualization technique that allows operating system (OS) processes to be run and managed independently of each other [1]. Kubernetes is the most effective container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a highly flexible and efficient way to manage and orchestrate containers, making it easier to build, deploy, and scale applications in a distributed environment.

Kubernetes allows for automatic allocation of containers to physical nodes using a master-slave architecture, where communication between the master and slave is made using a Kubelet device. It supports container-based deployment within platform-as-a-service (PaaS) clouds, focusing especially on cluster-based systems [2].

Autoscaling is a mechanism that scales resources up or down based on defined situations such as traffic or usage levels. In the literature, there are different types of autoscaling strategies [3], and, to be short, here we quote the terminology of the Middleware company[1] for positioning our work close to the reactive notion:

1. **Reactive autoscaling:** *It is based on predefined 'triggers' or thresholds specified by the administrator, which activates additional servers when crossed. You can set thresholds for server performance metrics such as the percentage of occupied capacity. Our work considers a form of reactive autoscaling triggered by the loads of Kubernetes nodes and incoming container traffic.*
2. **Proactive or predictive autoscaling:** *This type is suitable for applications with more or less predictable server loads. Predictive or proactive autoscaling schedules additional servers to run automatically during peak traffic times based on the time of day. This type of autoscaling uses artificial intelligence (AI) to "predict" when traffic would be high and schedules server augmentations in advance.*
3. **Scheduled autoscaling:** *It is similar to predictive autoscaling; the only difference is scheduling additional Kubernetes nodes for peak time. Although predictive autoscaling does this autonomously, scheduled autoscaling is more based on human input to schedule the nodes.*

Kubernetes naturally provides two autoscaler features related to Pods: Vertical Pod Autoscaling (VPA) and Horizontal Pod Autoscaling (HPA). VPA is a feature that automatically scales resources as CPU or memory allocated to individual pods based on CPU utilization or other metrics, which is a separate project that can be found on GitHub[2]. This means that VPA can increase or decrease the resources allocated to a Pod to ensure that it has the resources it needs to run efficiently. HPA is a feature that automatically scales the number of replicas of a Deployment, StatefulSet, or ReplicationController based on CPU utilization or other metrics. This means that HPA can add or remove Pods (containers) to the cluster to ensure that the workload is running efficiently and meeting demand[3].

VPA and HPA can be used together to achieve more granular control over autoscaling. For example, VPA can be used to scale the resources allocated to each Pod, while HPA can be used to scale the number of Pods running a workload. In addition, HPA and VPA assume that the number of nodes in a Kubernetes cluster is always

[1] https://middleware.io/blog/what-is-autoscaling/
[2] https://github.com/kubernetes/autoscaler/tree/bb32e8acbd71295585a/vertical-pod-autoscaler
[3] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

fixed. This is because they are designed to scale the number of Pods or the resources allocated to Pods, but not the number of nodes.

In Kubernetes, many nodes may remain active despite a minimal number of running containers, leading to significant resource waste. In this case, human intervention is required to control the utilization rate of the compute nodes, which can reduce the total cost of using a Kubernetes cluster.

To improve this limitation, we propose in this paper a new reactive autoscaler mechanism that allows the number of active nodes to be controlled dynamically while satisfying the Quality of Service (QoS) requirements. To the best of our knowledge, there is a lack of studies that propose autoscaling by considering the Kubernetes compute nodes. The main objective of this paper is to present a mechanism that dynamically adapts the number of active nodes according to the state of the computing infrastructure and uses a multi-criteria decision-making (MCDM) algorithm to select the most pertinent Kubernetes node that must be considered as active nodes.

The main contributions of this paper include:

- Introducing our reactive Kubernetes autoscaler mechanism.
- Implementing and packaging our Kubernetes autoscaler.
- Providing experiments to demonstrate the performance of the proposed autoscaler on top of Kubernetes.

The remainder of this paper is organized as follows. The related literature is reviewed in Section 2. In Section 3, the algorithm and the principle of the proposed autoscaler mechanism are illustrated and explained. Section 4 shows an example of how our autoscaler mechanism works. Section 5 presents the experimental designs and results of the proposed autoscaler. Section 6 concludes the work and highlights further possible improvements.

2 RELATED WORK AND POSITIONING

In this section, we will start by presenting in Subsection 2.1 some container management tools. Then, we present in sub-section 2.2 an overview of some works proposed in the context of autoscaling. Finally, we will conclude this section by positioning in sub-section 2.3.

2.1 Container management tools

In the literature, many container management tools come from different companies and open-source communities. As an example, we can consider Docker SwarmKit [4], Apache Mesos [5], Red Hat OpenShift [6] and Google Kubernetes [7].

Docker Swarmkit [4] is an important container scheduling framework developed by Docker. The Swarm manager schedules containers and chooses a node for each container. The Docker Swarmkit goes through two steps to select one node for each newly submitted container. First, it uses a filter mechanism to select a set of candidate nodes capable of running the container. Then, it selects the most suitable one, from the set of candidate nodes, according to a specific container scheduling strategy.

Mesos [8] is an Apache project. It is a thin resource-sharing layer that enables fine-grained sharing across various cluster computing frameworks by giving frameworks a common interface for accessing cluster resources. Marathon [4], runs on top of Mesos and orchestrates container-based applications on Mesos nodes.

Red Hat OpenShift [6] is a container orchestration platform optimized for web applications. It enables building, testing, and deploying web applications without provisioning and maintaining dedicated servers for each application. Applications hosted in OpenShift are managed through administration servers. All applications management interactions can be performed through a CLI (Command Line Interface) or a web interface [9].

Google Kubernetes [7] is a well-known open-source container orchestration tool. It performs automatic deployment, scaling, and management of container-based applications. Kubernetes is based on the Pod concept, each Pod is an abstraction that aggregates one or several containers. Pods can be run on-premises or on public cloud infrastructures. It sets up a cluster consisting of a Kubernetes master and a set of Kubernetes workers.

The Kubernetes scheduler assigns a Pod to a single node going through two steps. The first step consists of filtering all nodes to remove the ones that do not meet certain requirements of the Pod; if there are not enough resources in the infrastructure, the Pod goes into pending mode. The second step is ranking the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each node that survived filtering.

2.2 Autoscaling studies

In the literature, several studies have been proposed that introduce autoscaling mechanisms, and we review some of them in the following [10–16].

In [10], the authors propose a comparison between a new model-based reinforcement learning policy and the default threshold-based scaling policy of Kubernetes. The proposed solution consists of learning a suitable scaling policy from experience to meet the quality of service requirements expressed in terms of average response time. Using prototype-based experiments, the authors show the benefits and flexibility of the reinforcement learning policy compared to the default Kubernetes scaling solution.

In [11], the authors propose KOSMOS, a novel autoscaling solution for Kubernetes. Containers are individually controlled by control-theoretical planners that manage container resources on the fly (vertical scaling). A dedicated component handles resource contention scenarios between containers deployed on the same node (a physical or virtual machine). Finally, at the cluster level, a heuristic-based controller is in charge of the horizontal scaling of each application.

In [12], the authors designed a Resource Utilization-Based Autoscaling System (RUBAS) that can dynamically adjust the allocation of containers running in a Kubernetes cluster. RUBAS improves upon the Kubernetes Vertical Pod Autoscaler (VPA) system non-disruptively by incorporating container migration. As a result, the authors show that compared to Kubernetes VPA, RUBAS improves the CPU and memory utilization of the cluster by 10% and reduces the runtime by 15%, with an overhead for each application ranging from 5% to 20%.

In [13], the authors investigate the Horizontal Pod Autoscaler (HPA) through various experiments to provide critical knowledge

[4] https://mesosphere.github.io/marathon/

on its operational behaviors. They also discuss the essential difference between Kubernetes Resource Metrics (KRM) and Prometheus Custom Metrics (PCM) and how they affect HPA's performance. Lastly, the authors provide deeper insight and lessons on how to optimize the performance of HPA for researchers, developers, and system administrators who work with Kubernetes in the future.

In [14], the authors propose an auto-scaling scheme that dynamically adjusts the number of application instances to determine a balance between resource usage and application performance. The key components of the proposed solution include a scheme to monitor the load status of physical hosts, an algorithm that determines the appropriate number of application instances, and an interface to Kubernetes to perform the adjustment. Experiments have been conducted to investigate the performance of the proposed scheme, and the results confirm its effectiveness in reducing the response time of the application.

In [15], the authors propose the Traffic-aware Horizontal Pod Autoscaler (THPA), which operates on top of Kubernetes to enable real-time traffic-aware resource autoscaling for IoT applications in an edge computing environment. The proposed THPA performs upscaling and downscaling actions based on network traffic information from nodes to improve the quality of IoT services in the edge computing infrastructure. Experimental results show that Kubernetes with THPA improves the average response time and throughput of IoT applications by approximately 150% compared to Kubernetes with the horizontal pod autoscaler. This indicates that it is important to provide proper resource scaling according to the network traffic distribution to maximize IoT applications' performance in an edge computing environment.

In [16], the authors propose a Proactive Pod Autoscaler (PPA) for edge computing applications on Kubernetes. The proposed PPA can forecast workloads in advance with multiple user-defined/customized metrics and scale edge computing applications up and down accordingly. The PPA is optimized and evaluated on an example CPU-intensive edge computing application. As a result, the proposed PPA outperforms the default pod autoscaler of Kubernetes on both efficiency of resource utilization and application performance.

In [17], the authors propose an Intelligent Horizontal Proactive Autoscaling (IHPA) mechanism that leverages resource usage metrics of processing edge nodes such as CPU, RAM, and Bandwidth in order to provide timely scale-up and scale-down decisions. The proposed IHPA is based on a double tower Deep Learning (DL) architecture. In order to find a close to optimal DL architecture and guarantee the generality of the proposed approach, the authors also propose the innovative hybrid Bayesian Evolution Strategy method.

Karpenter is an open source node provisioning project[5] built for Kubernetes. Karpenter improves the efficiency and cost of running workloads on Kubernetes clusters by provisioning nodes that meet the requirements of the pods. Among the requirements, Karpenter deals with prices, which is not the case in our proposal.

2.3 Positioning

In contrast to these related and above-mentioned works, our autoscaler mechanism is designed to dynamically control Kubernetes computing nodes. It can be seen as a reactive pilot that dynamically controls the state of the Kubernetes cluster based on several factors and only keeps the resources that are used to reduce waste of resources and energy consumption.

Our proposed autoscaler mechanism features the following points:

- One of the few studies that focus on reactive autoscaling for Kubernetes computing nodes;
- Our autoscaler allows us to reduce the waste of resources and energy consumption in the Kubernetes cluster by automatically stopping nodes that are not in use;
- The autoscaler algorithm is based on several factors to decide when to scale up or down nodes;
- The decision of whether to scale up or down Kubernetes nodes is based on a multi-criteria decision-making (MCDM) algorithm to select the most pertinent Kubernetes node for each step (scale up or down).
- Our autoscaling mechanism can be easily deployed on a Kubernetes cluster using a Helmpackage [18].

3 THE AUTOSCALER PRINCIPLE

The native Kubernetes framework continuously utilizes all computing resources when containers are scheduled on-premise. This mechanism leads to increased resource wastage and higher energy consumption, as in some cases, resources are activated even when no Kubernetes object (container) is currently running. To address this issue, we propose a new autoscaling mechanism that adapts the number of active nodes by communicating with the Kubernetes API to assess the overall node load and check the presence of pending pods (nodes that are stopped and waiting to be activated if the Kubernetes cluster load increases).

For performance optimization, our autoscaling mechanism can be deployed using a Helm [18] package on top of Kubernetes. Helm is a Kubernetes package manager that enables the discovery, sharing, and utilization of software designed for Kubernetes.

In our autoscaling mechanism, we use the PROMETHEE multi-criteria decision algorithm in each scale-up/down step to select the most pertinent node, considering several criteria related to the number of CPUs, memory size, storage size, and energy consumption.

In the following, we present the principle of the PROMETHEE algorithm. Then, we outline the algorithm of our autoscaler mechanism and how we scale up/down nodes.

3.1 PROMETHEE algorithm

PROMETHEE is a multi-criteria decision algorithm that enables the establishment of an out-ranking among various alternatives [19]. It operates on a pairwise comparison of potential decisions (computing nodes). In our case, each computing node is represented by five criteria: the number of allocated CPUs, the size of allocated memory, the size of allocated storage, energy consumption, and the number of running Kubernetes objects. Following this, PROMETHEE selects the most pertinent node by considering the maximization/minimization of each criterion depending on the type of autoscaling up or down.

The PROMETHEE algorithm requires a preference function that characterizes the difference for a criterion between the evaluations

[5] https://karpenter.sh/

obtained by two possible nodes in a preference degree ranging from 0 to 1. In summary, PROMETHEE is composed of four steps [20], and is used as follows:

(1) Calculate, for each pair of nodes ($Node_a$ and $Node_b$), and each criterion, the value of the preference degree. Let $g_j(Node_a)$ be the value of criterion j for $Node_a$. We denote $d_j(Node_a, Node_b)$:

$$d_j(Node_a, Node_b) = g_j(Node_a) - g_j(Node_b)$$

$d_j(Node_a, Node_b)$ represents the difference in the value of criterion j between $Node_a$ and $Node_b$. $P_j(Node_a, Node_b)$ represents the value of the preference degree of criterion j for $Node_a$ and $Node_b$. In this work, we use the standard preference functions, defined as follows:

$$P_j(Node_a, Node_b) = \begin{cases} 0 & if\ d_j \leq 0 \\ 1 & if\ d_j > 0 \end{cases}$$

(2) It computes a global preference index for each pair of nodes ($Node_a$ and $Node_b$). Let C be the set of criteria considered and W_j be the weight associated with criterion j. In our case, W_j is equal to 1. The global preference index for a pair of nodes ($Node_a$ and $Node_b$) is calculated as follows:

$$\pi(Node_a, Node_b) = \sum_{j \in C} W_j \times P_j(Node_a, Node_b)$$

(3) It computes, for each node, the positive outranking flow $\phi^+(Node_a)$ and the negative outranking flow $\phi^-(Node_a)$. Let A be the set of nodes with a size of n. The positive and negative outranking flow of nodes are computed using the following formulas:

$$\phi^+(Node_a) = \frac{1}{n-1} \sum_{x \in A} \pi(Node_a, x)$$

and

$$\phi^-(Node_a) = \frac{1}{n-1} \sum_{x \in A} \pi(x, Node_a)$$

(4) It uses the outranking flows to establish a complete ranking between nodes. The ranking is based on the net outranking flows $\phi(Node_a)$, which are calculated as follows: $\phi(Node_a) = \phi^+(Node_a) - \phi^-(Node_a)$. In our work, the first request returned by PROMETHEE is the request that has the highest net outranking value.

One of the advantages of the PROMETHEE method is its reasonable time complexity, which is $O(q.n.\log(n))$, where q represents the number of criteria and n the number of alternatives [21]. This complexity is reasonable for practical use cases with several criteria much below 100.

3.2 Algorithm of our autoscaler mechanism

The algorithm 1 illustrates the principle of our auto-scaling mechanism. Initially, the Kubernetes cluster has one active node. Then, every minute, the autoscaling algorithm checks three conditions:

- The load of all active nodes in terms of CPU, memory, and storage exceeds the load that triggers the scale-up;
- There is at least one pending Pod, which means that the pod is waiting to be executed;
- There is at least one pending node, which means the node is stopped waiting to be activated.

Algorithm 1 Autoscaler algorithm

Require: L_{cpu}, the load of all active nodes in terms of CPUs in all the Kubernetes cluster
Require: L_{memory}, the load of all active nodes in terms of memory in all the Kubernetes cluster
Require: $L_{storage}$, the load of all active nodes in terms of storage in all the Kubernetes cluster
Require: $Pending_{pods}$, the number of pending Pods in all the Kubernetes cluster
Require: $Pending_{nodes}$, the number of pending nodes in all the Kubernetes cluster (nodes that are stopped and waiting to be activated - unused nodes)
Require: $Active_{nodes}$, the number of active nodes in all the Kubernetes cluster (used nodes)
Require: $Master_{node}$, the master node of the Kubernetes cluster which is activated all the time
Require: $Load_{up}$, the load that triggers the scale-up
Require: $Load_{down}$, the load that triggers the scale down
Require: $Pending_{pod}$, The minimum pending time of pod that can trigger the scale-up

```
while 1 minute do
  if Pending_pods > 0 && (L_cpu >= Load_up or L_memory >= Load_up or L_storage >= Load_up) && Pending_nodes > 0 then
    Node_x = Apply the PROMETHEE multi-criteria algorithm to select the most pertinent node that maximizes the allocated number of CPUs, memory size, and storage size and also minimizes energy consumption.
    add Node_x as active node (Scale-up)
  else
    if At least one pending Pod waiting for more than Pending_pod && Pending_nodes > 0 then
      Node_x = Apply the PROMETHEE multi-criteria algorithm to select the most pertinent node that maximizes the allocated number of CPUs, memory size, and storage size and also minimizes energy consumption.
      add Node_x as active node (Scale-up)
    else
      if Pending_pods == 0 && L_cpu <= Load_down && L_memory <= Load_down && L_storage <= Load_down && Active_nodes > 1 then
        Node_x = Apply the PROMETHEE multi-criteria algorithm to select the most pertinent node that minimizes the allocated number of CPUs, memory size, storage size, and number of Kubernetes objects and also maximizes energy consumption.
        Migrate all Kubernetes objects running in Node_x to the other active nodes
        Stop Node_x (Scale down)
      end if
    end if
  end if
end while
```

If these conditions are met, the autoscaling algorithm applies the PROMETHEE multicriteria algorithm to select from all pending nodes the most pertinent node to scale up. If all three conditions are not met, the autoscaling algorithm checks if there is at least one pending node and at least one pending Pod waiting for more than the minimum waiting time of the Pod. If this condition is met, a new node is selected from the pending nodes using the PROMETHEE algorithm to scale up.

To scale down, the algorithm checks if the load of active nodes in terms of CPU, memory, and storage is less than the load that triggers scaling down. If this condition is met, the autoscaler applies the PROMETHEE algorithm to select from the active nodes the most pertinent node to scale down. Then, it migrates all Kubernetes objects running on the selected node to other active nodes of the Kubernetes cluster and stops the selected node.

Let us assume that:

- CPU_{used} (and similarly $Memory_{used}$ and $Storage_{used}$) represents all the CPUs (memory space and storage space, respectively) used in the Kubernetes cluster.
- $CPU_{allocated}$ (and similarly $Memory_{allocated}$ and $Storage_{allocated}$) represents all the CPUs (memory space and storage space, respectively) allocated in the Kubernetes cluster.

To calculate the load of a CPU, memory, and storage, we propose using the following formulas:

- $Load_{cpu} = \frac{CPU_{used} \times 100}{CPU_{allocated}}$.
- $Load_{memory} = \frac{Memory_{used} \times 100}{Memory_{allocated}}$.
- $Load_{storage} = \frac{Storage_{used} \times 100}{Storage_{allocated}}$.

All CPUs used (as well as memory and storage) in the Kubernetes cluster represent the sum of the CPU (memory and storage, respectively) used on each node of the Kubernetes cluster. However, all CPUs allocated (as well as memory and storage) in the Kubernetes cluster represent the sum of the CPU (memory and storage, respectively) allocated on each node of the Kubernetes cluster.

3.3 How to scale up/down nodes

To scale up or down nodes as outlined in Algorithm 1, we use the PROMETHEE multi-criteria algorithm to select the most pertinent nodes. PROMETHEE algorithm is well suited for this task because it allows to consider multiple criteria simultaneously and to weigh them according to specific needs. In our case, we consider all criteria to have the same weight. However, it is possible to favor one criterion over another by modifying the weight of each criterion.

To scale up, we propose to select among the pending nodes the node that maximizes the allocated number of CPUs, memory size, and storage size, and also minimizes energy consumption. The goal is to select the node that:

- Has a large amount of resources in terms of CPUs, memory, and storage to satisfy the maximum number of Pods that will be submitted by users;
- Consumes the least energy.

To scale down, we propose to select from the active nodes the node that minimizes the allocated number of CPUs, memory size, and storage size and also maximizes energy consumption. The goal is to select the node that:

- Has a small amount of resources in terms of CPUs, memory, and storage to avoid migrating a large number of Pods and overloading the other nodes excessively.
- Consumes the most energy to reduce the global energy consumption of the Kubernetes cluster.

For scaling up, the idea is to add a node to the Kubernetes cluster with a significant allocation of resources in terms of CPUs, memory, and storage space to accommodate a maximum number of Pods. Simultaneously, we aim to add a node that consumes minimal energy.

For scaling down, the idea is to remove a node from the Kubernetes cluster with a smaller allocation of resources in terms of CPUs, memory, storage space, and number of Kubernetes objects ensuring that the cluster retains ample computing resources to execute new Pods. Additionally, we remove the node that consumes the most energy to reduce the overall energy consumption of the entire Kubernetes cluster.

This technique of scaling up/down helps to ensure that the Kubernetes cluster has the optimal amount of resources to meet demand, while also minimizing energy consumption.

4 EXAMPLE OF USING OUR AUTOSCALING MECHANISME

In this example, we illustrate how our autoscaling algorithm works in practice by considering only the CPU criterion, for the sake of simplicity.

Let us suppose that we have:

- 2 nodes, each equipped with 8 CPUs. The two nodes are identical and have the same characteristics.
- 2 Pods, each with one container that consumes 6 CPUs, and they run for 120 seconds.
  - The first Pod is launched at T0, and the second one is launched at T0+60 seconds.

Let us suppose also that the variables of our autoscaling algorithm are set as follows:

- $Load_{up}$=70%;
- $Load_{down}$=40%;
- $Pending_{pod}$=2 minutes.

In the beginning, our autoscaler keeps at least one node active ($node_1$) which represents the master node of the Kubernetes cluster, and the second node ($node_2$) is considered as a pending node (node stopped and waiting to be activated). At time T0, Pod_1 is submitted. In this case, Pod_1 is assigned to $node_1$ and the load of on $node_1 = \frac{6*100}{8} = 75\%$

At T0+60 seconds, Pod_2 is scheduled for execution. In this situation, the pod cannot run because $node_1$ has only two available CPUs, while Pod_2 requires 6 CPUs. At the same time (T0+60), our autoscaling algorithm detects that there is at least one pending Pod (Pod_2), the CPU load of $node_1$ is 75% (bigger than the $Load_{up}$ set to 70%), and there is at last one pending node ($node_2$). In this case, at T0+60 seconds, our autoscaling mechanism decides to add $node_2$ to the Kubernetes cluster to execute Pod_2 as presented in Algorithm 1. At T0+120, Pod_1 stops. In this case, the load will be equal to 37.5%

Figure 1: Example of how our autoscaler mechanism works to execute two Pods on two nodes

Figure 2: The number of nodes used to execute our workflow of 4 Pods without using the autoscaling mechanism in $infrastructure_1$ and $infrastructure_2$

($\frac{6*100}{16}$), where 6 represents the CPU consumption of Pod_2, and 16 represents the total amount of CPU available in $node_1$ and $node_2$ (8+8). In this situation, the autoscaling mechanism detects that the load of 37.5% is less than $Load_{down}$, which is set to 40%, and there are no pending Pods. Consequently, the autoscaler scales down and removes $node_2$ from the Kubernetes cluster.

Figure 1 shows the evolution of the number of active/pending nodes used to execute the two Pods (Pod_1 and Pod_2). It is worth noting that in this example, for the sake of simplicity, we considered only the CPU criterion for scaling up or down the node.

5 EVALUATIONS

In this section, we present some experiments to evaluate the performance of our autoscaling mechanism. We used the Virtual WALL testbed [22] for our experiments. Virtual WALL is an emulation environment that can be used as bare metal hardware (operating system running directly on the machine) or virtualized through OpenVZ containers or XEN virtualization. In our experimental evaluation, we booked an infrastructure composed of nodes with the following characteristics: Intel(R) Xeon(R) CPU E5645 (2.40GHz) with 24 CPUs, 24GB of memory, and 16GB of storage.

To test our approach, we proposed running a workflow composed of 4 Pods in Kubernetes clusters on two infrastructures ($infrastructure_1$ is composed of 4 nodes and $infrastructure_2$ is composed of 5 nodes), with the following constraints:

- The frequency of Pods submission is 5 min, which means there is a 5-minute wait between each two Pods submission.
- Each Pod consumes 17 CPUs.
- Our autoscaling algorithm has the following configuration:
  - $Load_{up}$=70%;
  - $Load_{down}$=40%;
  - $Pending_{pod}$=2 minutes.

Figure 2 shows the number of computing nodes used to execute our 4 Pod workflow in $infrastructure_1$ (which is composed of 4 computing nodes) and $infrastructure_2$ (which is composed of 5 computing nodes) without the autoscaler mechanism.

Figure 3: The number of nodes used to execute our workflow of 4 Pods using the autoscaling mechanism in $infrastructure_1$

Figure 4: The number of nodes used to execute our workflow of 4 Pods using the autoscaling mechanism in $infrastructure_2$

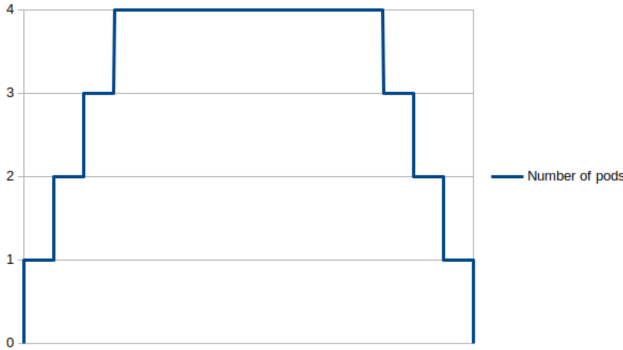

Figure 5: Varying the number of Pods running in $infrastructure_1$ without using the autoscaler mechanism

Figure 3 shows the number of computing nodes used to execute our workflow of 4 Pods in $infrastructure_1$ with the autoscaler mechanism.

Figure 4 shows the number of computing nodes used to execute our 4 pod workflow in $infrastructure_2$ with the auto-scaler mechanism.

In contrast, Figures 3 and 4 show that with the autoscaler mechanism, the number of computing nodes used increases when the load increases (e.g., when a new Pod is submitted) and decreases when the load decreases (e.g., when a pod stops). This mechanism allows computing nodes to be used only when necessary, which can save resources and energy consumption.

In Figure 4, we can see that our 4 Pod workflow is executed only on 4 nodes. The fifth node is turned off because it is not needed.

Typoe of infrastructure	Approaches	Nodes utilization rate	
		Pending nodes	Active nodes
$Infrastructure_1$	Without autoscaler	0	100%
	With autoscaler	17,22%	82,77
$Infrastructure_2$	Without autoscaler	0	100%
	With autoscaler	33,37%	66,62%

Table 1: Comparison between nodes utilization rate in $infrastructure_1$ and $infrastructure_2$

Table 1 shows a comparison between node utilization rates in $infrastructure_1$ and $infrastructure_2$ without and with our autoscaler mechanism. It is observable that with the autoscaler mechanism, the utilization rate of active nodes is significantly lower than the utilization rate of active nodes, which is always 100%, when not using the autoscaler mechanism.

Figure 5 represents the variation in the number of Pods running in $infrastructure_1$ without using the autoscaler mechanism. We observed the same scenario when: (i) Pods are running in $infrastructure_1$ with the autoscaler mechanism, and (ii) Pods are running in $infrastructure_2$ with and without the autoscaler mechanism.

Type of infrastructure	Approaches	Comparison criteria	
		Computing time (s)	Energy consumption (wh)
$Infrastructure_1$	Without autoscaler	4506,58	100,98
	With autoscaler	4737,63	95,3
$Infrastructure_2$	Without autoscaler	4504,52	126,025
	With autoscaler	4682,66	99,1

Table 2: Comparison between computing time and energy consumption to execute our Pods workflow in $infrastructure_1$ and $infrastructure_2$

Type of infrastructure	Speedup/ratio	
	Computing time	Energy consumption
$Infrastructure_1$	0,951%	1,059%
$Infrastructure_2$	0,961%	1,271%

Table 3: Speedup obtained in terms of computing time and energy consumption

Table 2 compares the computing time and energy consumed to execute our workflow of Pods in $infrastructure_1$ and $infrastructure_2$, without and with our autoscaler mechanism.

Table 3 shows the acceleration in terms of computing time and the ratio of energy consumption obtained with the autoscaler versus without the autoscaler mechanism. The speedup of computing time is also the ratio between the computing time obtained without using the autoscaler mechanism and the computing time obtained using the autoscaler mechanism.

Concerning energy consumption, we note that the ratio is greater than 1, which means that the use of the autoscaler mechanism reduces energy consumption compared to native utilization without the autoscaler. The energy ratio represents the ratio between the energy consumed without using the autoscaler mechanism and the energy consumed using the autoscaler mechanism.

However, we observe a speedup that varies between 0.951% and 0.961% in terms of computation time. This means that the use of the autoscaler has introduced a small overhead compared to the computing time achieved without the autoscaler mechanism.

We quantified an overhead varying between 4% and 5% using our autoscaler mechanism. This overhead can be explained as follows: If a new Pod is executed and there are no active nodes available to execute the submitted Pod, the autoscaler takes time to add a new node to the Kubernetes cluster to execute the new submitted Pod. This step adds additional cost to the overall execution time of the Pod workflow. On the other hand, without the autoscaling mechanism, each time a Pod is submitted, it can be executed directly if there is a computing node available.

6 CONCLUSION

In this paper, we present a new Kubernetes autoscaler that dynamically adapts the number of active nodes and keeps only the nodes that are used. Our autoscaler is designed to be easily deployed on top of Kubernetes, using the Helm package [18].

In terms of perspective, we propose to use Machine Learning (ML) techniques in our autoscaler mechanism to automatically adapt the number of active nodes. The idea is first to predict the number of active nodes based on the different Pod submission periods, and second to mix two mechanisms: reactive and proactive autoscaling.

In our approach, we consider that each time a Pod is assigned to a node, it will be executed without failures. From another perspective, we suggest working on the problem of fault tolerance in case a node fails in the computing infrastructure. In this situation, we plan to use a smart replica approach of Pods.

ACKNOWLEDGEMENTS

This research was funded by Banque Publique d'Investissement (Bpifrance), Appel à manifestation d'intérêt (AMI) « Stratégie d'accélération Cloud », and specifically « Développement et renforcement de la filière française et européenne du Cloud.

REFERENCES

[1] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[2] Víctor Medel, Rafael Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F Rana. Characterising resource management performance in kubernetes. *Computers & Electrical Engineering*, 68:286–297, 2018.

[3] Marco A.S. Netto, Carlos Cardonha, Renato L.F. Cunha, and Marcos D. Assuncao. Evaluating auto-scaling strategies for cloud computing environments. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 187–196, 2014.

[4] Swarm kit: `https://github.com/docker/swarmkit/`.

[5] The apache software foundation. mesos, apache: `http://mesos.apache.org/`.

[6] Openshift `https://www.openshift.com/`, visited 27-02-2024.

[7] Kubernetes framework `https://kubernetes.io/`.

[8] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, pages 22–22, 2011.

[9] Alexandre Lossent, A Rodriguez Peon, and A Wagner. Paas for web applications with openshift origin. In *Journal of Physics: Conference Series*, volume 898, page 082037. IOP Publishing, 2017.

[10] Fabiana Rossi. Auto-scaling policies to adapt the application deployment in kubernetes. In *ZEUS*, pages 30–38, 2020.

[11] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. Kosmos: Vertical and horizontal resource autoscaling for kubernetes. In *International Conference on Service-Oriented Computing*, pages 821–829. Springer, 2021.

[12] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40. IEEE, 2019.

[13] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16):4621, 2020.

[14] Wei-Sheng Zheng and Li-Hsing Yen. Auto-scaling in kubernetes-based fog computing platform. In *New Trends in Computer Technologies and Applications: 23rd International Computer Symposium, ICS 2018, Yunlin, Taiwan, December 20–22, 2018, Revised Selected Papers 23*, pages 338–345. Springer, 2019.

[15] Linh-An Phan, Taehong Kim, et al. Traffic-aware horizontal pod autoscaler in kubernetes-based edge computing infrastructure. *IEEE Access*, 10:18966–18977, 2022.

[16] Li Ju, Prashant Singh, and Salman Toor. Proactive autoscaling for edge computing systems with kubernetes. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 1–8, 2021.

[17] John Violos, Stylianos Tsanakas, Theodoros Theodoropoulos, Aris Leivadeas, Konstantinos Tserpes, and Theodora Varvarigou. Intelligent horizontal autoscaling in edge computing using a double tower neural network. *Computer Networks*, 217:109339, 2022.

[18] Helm package: https://helm.sh/- last access: 13/10/2023.

[19] S.C.Deshmukh. Preference ranking organization method of enrichment evaluation (promethee). *International Journal of Engineering Science Invention*, 2:28–34, 2013.

[20] P. Taillandier and S. Stinckwich. Using the promethee multi-criteria decision making method to define new exploration strategies for rescue robots. In *International Symposium on Safety, Security, and Rescue Robotics*, 2011.

[21] Toon Calders and Dimitri Van Assche. PROMETHEE is not quadratic: An o(qn log(n)) algorithm. *CoRR*, 2016.

[22] Virual wall tesbed: `https://doc.ilabt.imec.be/`.