ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 5, No. 6, 972-980 (2020)
www.astesj.com
Special Issue on Multidisciplinary Sciences and Engineering

ASTES Journal
ISSN: 2415-6698

Docker-C2A : Cost-Aware Autoscaler of Docker Containers for Microservices-based Applications

Mohamed Hedi Fourati[*,1], Soumaya Marzouk[1], Mohamed Jmaiel[1,2], Tom Guérout[3]

[1] *ReDCAD Laboratory, National Engineering School of Sfax, University of Sfax, Sfax, 3038, Tunisia*

[2] *Digital Research Center of Sfax, University of Sfax, Sfax, 3021, Tunisia*

[3] *Laboratory for Analysis and Architecture of Systems, University of Toulouse, Toulouse, 31031, France*

ARTICLE INFO

Article history:
Received: 14 September, 2020
Accepted: 26 November, 2020
Online: 08 December, 2020

Keywords:
Cloud Computing
Microservices
Docker
Kubernetes
Autoscaling
PSO algorithm

ABSTRACT

*This article proposes a cost-aware autoscaler for microservices-based applications deployed with docker containers. This autoscaler decreases the cost of the application deployment as it reduces computing resources. In elastic treatment, microservice resources are scaled when the used metric as the central processing unit (CPU) exceeds the threshold. In case of threshold exceeding, an autoscaler adds many instances of docker containers in order to satisfy the need of the application. In many studies, the autoscaler adds many containers without selecting the appropriate microservices for scaling and without in advance calculation of the adequate number of containers. This may lead to allocating additional resources to inappropriate microservices and a non optimal number of containers. For this reason, we propose our autoscaler "*Docker-C2A*" which identifies the adequate microservices to add resources. It also calculates the optimal number of needed containers. "*Docker-C2A*" analyses the state of the application, uses the execution history and uses a Particle Swarm Optimization (PSO) algorithm to identify the adequate microservices for scaling resources and to determine the optimal number of containers. As a result, "*Docker-C2A*" helps to reduce computing resources and to save extra costs. Experimental measurements were conducted on a microservices-based application as a concrete use-case demonstrating the effectiveness of our proposed solution.*

1 Introduction

Microservices is an architectural style of development consisting of a collection of small independent and loosely coupled components [2]. Nowadays, several companies like Amazon, Linkedin, Spotify and Netflix have migrated towards microservice architecture in order to increase the efficiency and the scalability of computing resources.

Currently, the most of microservices-based applications are deployed in docker containers [3] and orchestrated with kubernetes tool [4]. Docker is a container technology designed for creating, deploying and running applications using containers. The latter allow developers to package up an application with all needed parts such as libraries and deploy it as one package. Each instance of a microservice is a docker container. They are orchestrated and deployed in a cluster of VMs (Virtual Machines) configured by kubernetes, an open-source container orchestration system created by Google for automation of application deployment, scaling and management.

When there is a rise on the workload, microservices resources are overloaded and thus additional resources should be allocated. One of the most important stakes in Cloud environment is to minimize computing resources so as to reduce the deployment cost of the application. We can resolve this issue and optimize computing resources using autoscaling techniques. Over-provisioning of computing resources leads to allocating unused resources. Whereas, under-provisioning causes performance degradation of the application. Thus, an autoscaling technique should identify the needed resources for the application in order to allocate quasi-exact needed computing resources.

In literature, few autoscalers are proposed for containers and microservices-based applications, and most existing solutions have the same treatment as kubernetes autoscaler. The autoscaler of kubernetes is called Horizontal Pod Autoscaler HPA [5]. HPA scales the number of containers based on CPU usage. CPU threshold is

[*]Corresponding Author: Mohamed Hedi Fourati, ReDCAD Laboratory Sfax, Tunisia. Email : mohamed-hedi.fourati@redcad.org
This paper is an extension of work originally presented in [1].

specified by the user. When the application receives high workload due to the increase of requests number, many microservices are overloaded and the CPU threshold is exceeded. Consequently, kubernetes HPA adds one container instance to each overloaded microservice. If the threshold is still exceeded, the autoscaler continues adding other containers until reaching the normal state of the CPU or the maximum number of containers. This autoscaling process has several issues that allocate many resources and thus charge a high deployment cost.

The first issue is the usage of CPU metric. In fact, using CPU metric in autoscaling is not effective and may lead to allocating unnecessary resources. Indeed, it was proved in [6] that a high CPU usage means that the container instance is fully utilized, but it can still provide acceptable response time without adding more resources. So, it is possible to execute further requests and to have acceptable response time even if we have high CPU and high workload. This is also why many elastic solutions tend to use response time metric instead of CPU usage to benchmark their autoscaler. Moreover, it is highly recommended to use response time metric instead of resource metrics as CPU when launching autoscaling actions.

The second issue is that existing autoscalers do not select the adequate microservices for adding resources. In fact, the microservices of an application are in relationship, and an issue in a given microservice may be propagated to other ones. Similarly, if this issue is resolved for a microservice, it is automatically resolved for related microservices. Thus, an autoscaler should select and find the eligible microservices for scaling. To do that, the autoscaler should analyze the whole application with an overview of all microservices. Also, it should use the execution history of the application to find eligible microservices.

The third issue is that existing autoscalers do not calculate the needed number of containers for each microservice and do not react fast. For example, the autoscaler of kubernetes adds containers one by one until reaching the normal state. In this case, the normal state is recovered after a period of time, while in Cloud applications, the time constraint is critical and fast scaling actions are requisite. Then, the autoscaler should calculate and estimate the needed number of containers for each microservice from the beginning to launch fast actions. To do this, the autoscaler should analyze the application and its microservices to estimate the adequate number of containers for each microservice. Moreover, it should consider the execution history and the architecture of the application as the relationship between microservices to estimate the adequate number of containers. As a result, the autoscaler calculates precisely the needed number of container and launches fast actions. This leads to optimize computing resources and deployment cost.

This paper proposes "Docker-C2A" a **C**ost **A**ware **A**utoscaler for microservices-based applications deployed with docker containers. This autoscaler focuses on existing issues in order to optimize computing resources and reduce deployment costs. Firstly, "Docker-C2A" is based on response time metric instead of CPU metric. Also, it selects eligible microservices for scaling and calculates the optimal number of containers to be added to each microservice. To do this, during autoscaling treatment, "Docker-C2A" considers the architecture of the application and the relationship between microservices. In addition, it uses the execution history of the application and executes a PSO algorithm. As a result, "Docker-C2A" selects appropriate microservices for scaling and calculates the needed number of containers for each microservice. Experimental results conducted on a concrete use case demonstrates that we can satisfy high workload with minimum computing resources by allocating a small number of containers. Thus, "Docker-C2A" optimizes computing resources and reduces the deployment cost. And that is the main purpose of this autoscaler.

This paper is organized as follows: Section 2 presents the related work. Section 3 presents the motivating example of our autoscaler. Section 4 presents in details Docker-C2A autoscaler. Section 5 details the PSO algorithm in Docker-C2A. Section 6 discusses the experimentations. We conclude the paper in Section 7.

2 Related Work

Autoscaling is dedicated for optimizing resource allocation in order to avoid allocating unused resources and to reduce the cost of deployment. Few studies were proposed for the autoscaling of microservices-based applications focusing on containers. As it was explained in section 1, these solutions have many issues. In fact, most of these solutions focus on CPU metric. Also, the resource allocation is not accurate as many resources are allocated without selecting the adequate microservices for scaling and without previous calculating the needed amount of resources. This may lead to allocating unused and unnecessary resources and thus charging the user extra costs. In this section we review these studies and explain the shortcomings of each one.

In [7], the author designed a container-based autoscaling policy with a mathematical model dedicated for Iot applications based on containers and microservices. This autoscaler uses two main metrics, number of requests and resource related metrics as CPU or memory usage. Also, it uses the history of the application to take adequate actions for similar cases recorded in the history. After each scaling action, the autoscaler waits for a cooling period defined by the user to avoid launching other scaling actions until the application becomes stable. The main objective of this autoscaler is to minimize computing resources and the deployment costs. Using the history, this autoscaler calculates and estimates the optimal number of containers used to scale each microservice. Although this solution estimates the needed resources for allocation, it does not select the eligible components for scaling.

In [8], the author presented a message queue as a specific use case in IoT context. They considered metrics related to message queue mi- croservices. Two classes of microservices were considered, compute-intensive and I/O-intensive. In [9], the author aimed to adjust computing resources to the incoming load. In scaling process, a new container is added when the response time exceeds 1000 ms in a 15 s window. A container can be removed if it uses less than 10% of cpu. These two studies neither select containers and components for scaling nor calculate the adequate quantity of resources to be allocated.

In [10], the author proposed a genetic algorithm for resources allocation and elasticity management dedicated for microservices-based applications deployed on a containerized environment. The main objectives of the proposed solution is the container allocation

in VMs and elastic treatment of containers. This study models the whole system such as applications, microservices, the relationship between microservices, the required resources of each microservice, containers, the needed resources, physical machines and the network. Then, it optimizes this model using a meta heuristic genetic algorithm to get optimal results for resource allocation of containers and the number of containers for each microservice. This autoscaler considers the architecture of the application and the relationship between microservices in scaling process. However, it does not calculate the needed amount of resources for scaling.

In [11], the researchers developed an autoscaler called Microscaler for microservices-based applications which identifies the scaling-needed services using a customized metric called Service Power. When detecting a response time violation, Microscaler launches the process of detecting services that should be scaled using the customized metric Service Power. This metric is based on response time and used to identify the microservices that need to be scaled with additional resources. Besides, this autoscaler calculates the adequate number of containers to be added for each selected microservice. This autoscaler uses a Bayesian Optimization (BO) and an heuristic approach to determine the optimal number of instances for selected services. Although this autoscaler selects the appropriate microservices to be scaled and calculates the number of additional containers, it is strongly related to applications based on service mesh that make this approach not applicable in other contexts.

In this paper, we propose "Docker-C2A" autoscaler for microservices-based applications using docker containers. We focus on docker container technology because it is the most used in the context of containerization and microservices-based applications. This autoscaler is based on response time metric. Also, "Docker-C2A" selects eligible microservices for scaling and calculates the optimal amount of resources to be allocated. This autoscaler uses the execution history of the application and a PSO heuristic algorithm. The execution history helps to determine which of application microservices need to be scaled. Then, the PSO algorithm calculates the optimal number of containers to be added to each selected microservice.

"Docker-C2A" can be distinguished over existing solutions as it allocates the optimal amount of resources to the adequate microservices. "Docker-C2A" is a fast and accurate autoscaler that gives more precise scaling actions. This leads to optimizing application resources and deployment costs.

3 Motivating Example

To illustrate existing issues, we detail the behavior of kubernetes autoscaler on a concrete microservices-based application. We take the example of kubernetes autoscaler as most existing autoscalers have the same treatment. In fact, we consider Bookinfo [12] a microservices-based application. Figure 1 illustrates Bookinfo application composed of four microservices.

This application displays the data related to each book. It is similar to a unique catalog entry of an online book store. It displays book details as ISBN, number of pages, and book reviews. Bookinfo application is composed of four separate microservices: **Productpage** microservice calls details and reviews microservices to get the book data. **Details** microservice gives the book information. **Reviews** microservice contains the book reviews and calls ratings microservice. **Ratings** microservice contains ranking formation deducted from the book review. Reviews microservice has three versions as three types of instance : **Reviews v1** does not call Rating microservice. **Reviews v2** invokes Rating microservice and displays rating stars as 1 to 5 black stars. **Reviews v3** invokes Rating microservice and displays rating stars as 1 to 5 red stars. This application is deployed with docker containers and deployed on a kubernetes cluster.

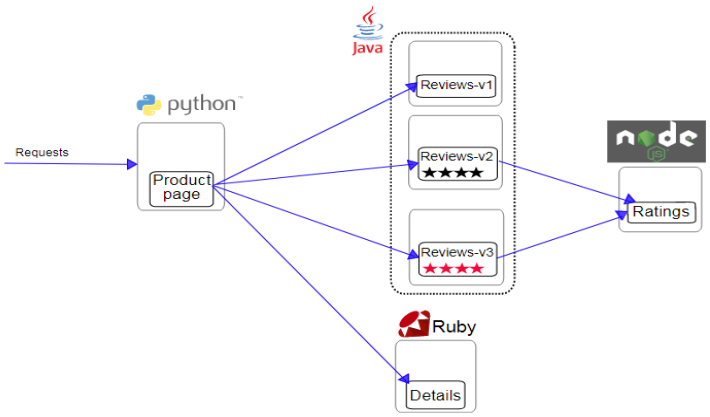


Figure 1: Bookinfo Application

Kubernetes is the most known and efficient orchestrator for docker containers. In the following, we use the pod concept of kubernetes. A pod in kubernetes is considered as a single docker container of a microservice. A microservice is composed by a set of pods (a set of containers). The elasticity controller of kubernetes is called Horizontal Pod Autoscaler HPA [5] and is based on CPU metric. It adds pods when the CPU threshold is exceeded. CPU threshold is specified by the user. If the maximum number of pods (containers) is reached, HPA autoscaler cannot add any more pods.

Many types of workload can be launched in this application. For example we can manage a workload that routes all incoming requests to Reviews v3 of microservice Review. Another example of workload routes incoming requests in 50% to Reviews v2 and 50% to Reviews v3. Technically, it is difficult to route requests flow to a specific container of a microservice. However, it is easy to manage the request flow with Istio [13] service. Istio is based on service mesh which facilitates the creation and management of the network of deployed services. Istio allows to manage network communication between microservices as well as the traffic of requests. In fact, it helps to create configuration rules for traffic routing in order to control the traffic flow between microservices.

In this paper we consider a workload that routes all incoming requests of microservice review to reviews-v3 instances. This workload launches a flow of 70 Req/s. It is not a very high workload but it saturates resources and generates overloaded microservices. This section presents the behavior of kubernetes HPA autoscaler using this workload.

Figure 2 shows the behavior of kubernetes HPA autoscaler using the presented workload with a request flow of 70 Req/s. Many pods were saturated and Kubernetes HPA launches autoscaling ac-

tions. Kuberenetes HPA adds 4 pods for productpage microservice resulting in a total of 5 pods. It adds also 1 pod to reviews-v3 microservice. From t=70, we have a total of 12 pods of reviews-v3. Ratings and details microservices were not saturated and we have no additional pods. The curves of ratings and details are superimposed as they have 1 pod all the time. The total number of pods required for this workload is 19 pods as 5 pods for productpage, 12 pods for reviews-v3, 1 pod for ratings and 1 pod for details. This autoscaler is based on CPU. If the CPU usage of a microservice exceeds threshold, the autoscaler adds many pods until recovering of the normal state.

However, as explained in section 1, using CPU usage in autoscaling treatment is not effective and may lead to allocating unnecessary resources. Therefore, we should also analyze the behavior of kubernetes autoscaler towards the response time. The red curve in Figure 2 illustrates the evolution of the response time in kubernetes autoscaler.

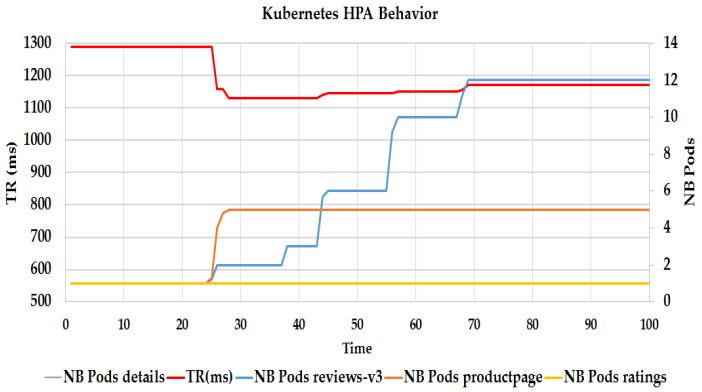


Figure 2: Kubernetes HPA Autoscaler behavior

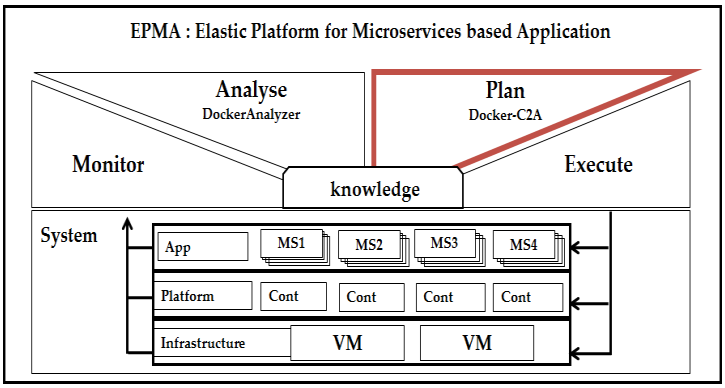


Figure 3: EPMA : Elastic Platform for Microservices based Application : Autoscaling Engine

Approximately, the response time starts with 1300 milliseconds (ms). At time t=25, we have 5 pods for productpage microservice and 2 pods for reviews-v3 microservice. We can notice a considerable evolution of the response time with approximately 1100 milliseconds. From t=37 to t=100, kubernetes autoscaler adds pods for the microservice reviews-v3 until reaching 12 pods. Following this addition, we notice no improvement in response time, but we can see a slight increase starting from t=43. It is concluded that during this interval of time, kubernetes autoscaler added 10 pods for reviews-v3 microservice, while the response time is not improved. So the added pods for reviews-v3 are not useful and kubernetes allocates unnecessary resources without any benefit.

With our autoscaler Docker-C2A, based on response time metric, we aim at optimizing the addition of resources and avoid allocating unnecessary resources. Docker-C2A uses PSO heuristic algorithm and the execution history of the application to optimize resource allocation. Using Docker-C2A, we can satisfy the same workload used in kubernetes example with a very small number of pods and with a better application performance.

4 Docker-C2A Autoscaler

4.1 EPMA platform

Docker-C2A is part of our EPMA platform (Elastic Platform for Microservices based Applications) illustrated in Figure 3. This platform is based on the autonomic MAPE-K loop proposed by IBM [14]. MAPE-K loop is composed of four components that share the Knowledge: Monitor, Analyze, Plan and Execute. EPMA manages the entire system with all levels: VM level, container level and microservice level. The analyze component DockerAnalyzer was previously presented [1] while this paper details the plan component Docker-C2A.

The monitor component collects data from different levels: VM, container and Microservice. This component collects metrics related to resources as CPU usage and memory usage and metrics related to the application performance as the response time. These parameters are filtered and correlated in order to determine symptoms that should be analyzed. As example of a symptom, an overload state caused by the CPU violation. If the monitor detects a symptom, DockerAnalyzer, the analyze component, is invoked to check whether this symptom is caused by a normal behavior as resource saturation or by an abnormal behavior as VM problem. If an abnormal behavior is detected, the analyzer had to check the root cause of the problem. Then, the plan component Docker-C2A generates the appropriate change plan describing the desired set of changes, and deliver the change plan to the execute component. The execute component applies the change plan and executes adequate actions for each layer. These four components share the knowledge containing particular data as policies and symptoms. This paper explains deeply the plan component Docker-C2A of EPMA platform.

Docker-C2A is based on the execution history of the application and a PSO heuristic algorithm. When the application state needs additional resources, Docker-C2A selects the best microservices for scaling and calculates the optimal amount of needed resources for each microservice.

4.2 Execution History

The main purpose of the execution history is to identify the microservices that should be scaled, and those that should not be scaled as there is no need to add resources. To do that, we attribute a score value for each microservice. The score value is a number varying from 0 to 5 depending on the utility of adding resources to the

microservice. For example, if we have a microservice that the addition of pods (containers) improves greatly the response time, the attributed score will be 5. If we have a microservice where adding pods will not improve the response time, the attributed score will be 0. As a result of the execution history, we generate the score value for each microservice. This helps to scale the appropriate microservices.

We account for the score attribution using bookinfo application with the same workload explained in section 3 which routes the incoming traffic of reviews microservice to reviews-v3 instances and uses a flow of 70 Req/s. Bookinfo application is composed of 4 microservices. We present in the following an extract from the history related to two microservices in order to demonstrate the score attribution.

NB Pods Productpage	NB Pods Reviews-v3	NB Pods Ratings	NB Pods Details	Response Time (ms)
1	1	1	1	1289
5	1	1	1	1134

Figure 4: An extract of Productpage history

In Figure 4, we change the number of pods in microservice productpage and we keep the same number of pods for other microservices. In the second row in this table of Figure 4, we have 5 pods of productpage and the response time has been improved considerably compared to the first line with 1 pod. The response time decreased effectively with 155 milliseconds. So requests are much faster with 5 pods than if we had 1 pod of productpage. In this case, we notice that adding pods to productpage microservice improves greatly the response time. We attribute the high score with value 5 for this microservice.

NB Pods Productpage	NB Pods Reviews-v3	NB Pods Ratings	NB Pods Details	Response Time (ms)
1	1	1	1	1289
1	5	1	1	1292

Figure 5: An extract of Reviews-v3 history

In Figure 5, we change the number of pods in microservice reviews-v3 and we keep the same number of pods for other microservices. There is no remarkable difference in response time between 1 and 5 pods. There is a very small increase in response time with 3 milliseconds, which is not profitable when we add 4 pods. In this case, we notice that adding pods to reviews-v3 microservice has no effect on the response time. Then, there is no change in the result. So we attribute the lowest score to reviews-v3 with value 0.

With the same process, we analyze the evolution of the response time when we add pods to ratings and details microservices. In fact, there is a weak improvement of the response time when we add pods to these two microservices. Then, we attribute the score value of 1 for ratings and details microservices while adding pods improves the results slightly.

As a conclusion for the execution history, we can admit that adding more resources to productpage microservice can effectively improve the response time of the application. Adding pods to reviews-v3 microservice has no effect on the response time. So adding pods to this microservice is unnecessary. Adding pods to ratings and details microservices improves the results slightly. They are not privileged in autoscaling treatment.

We illustrate the score values for each microservice in Figure 6.

	Microservice Index	Score value of scaling
productpage microservice →	0	5
reviews-v3 microservice →	1	0
ratings microservice →	2	1
details microservice →	3	1

Figure 6: Score value for each microservice

The execution history is very helpful when adding resources in scaling actions. In fact, it helps to select the appropriate microservices for scaling which leads to optimize resource allocation. After selecting eligible microservices for scaling, Docker-C2A calculates the optimal number of pods to be added to each microservice. To do this, Docker-C2A uses PSO algorithm to estimate the needed amount of resources for each microservice.

4.3 PSO Algorithm

Particle Swarm Optimization (PSO) is a computational technique based on the behavior of herds of animals that optimizes a given problem. It tries iteratively to improve candidate solutions with regard to a given quality measure. It was developed by Kennedy and Eberhart [15] in 1995 and widely used and researched ever since.

In PSO algorithm we define a particle which is analogous to a bird in a flock of birds. The flock of birds forms the search space. The movement of each particle is guided by the velocity vector. Each particle moves according to its best position pbest and the best position of the swarm gbest.

In this algorithm, we use a fitness function to measure the performance of each particle. In each iteration, the algorithm changes the velocity of each particle towards pbest and gbest positions. The algorithm stops when a stopping criterion is met or the number of iterations is reached. The velocity and positions equations are represented with Eqs. (1) and (2), respectively. The pseudo-code of the algorithm is presented in Algorithm 1.

$$V_i^{k+1} = wV_i^k + c_1 r_1 (pbest_i - X_i^k) + c_2 r_2 (gbest - X_i^k) \quad (1)$$

$$X_{i+1}^k = X_i^k + V_i^{k+1} \quad (2)$$

where

- w : inertia weight

- c_1 and c_2 : acceleration coefficients
- X_i^k : position of particle i at iteration k
- X_i^{k+1} : position of particle i at iteration k+1
- V_i^k : velocity of particle i at iteration k
- V_i^{k+1} : velocity of particle i at iteration k+1
- $pbest_i$: best position of particle i
- $gbest$: best position of the swarm

Algorithm 1 PSO Algorithm

```
Initialize a population of particles with random values positions and velocities from D dimensions in the search space
while Termination condition or Number of iterations not reached do
    for Each particle i do
        Update the velocity of the particle using Equation 1
        Update the position X_i of the particle using Equation 2
        Calculate the fitness value f(X_i)
        if f(X_i) < f(pbest_i) then
            pbest_i ← X_i
        end if
        if f(X_i) < f(gbest) then
            gbest ← X_i
        end if
    end for
end while
```

5 Docker-C2A : PSO Algorithm

Docker-C2A uses PSO algorithm to calculate the optimal number of pods (containers) to be added to each microservice. Two main objects should be defined in PSO algorithm: particle and fitness function.

5.1 Particle definition in Docker-C2A

In PSO algorithm, a particle represents a solution candidate of the problem. In our context, a solution is a particle that contains the adequate number of pods for each microservice. The dimension of the particle is defined by the number of microservices of the application. We consider the same example and workload explained in section 3. In this example, we use bookinfo application composed by 4 microservices: productpage, reviews-v3, ratings and details. In this case we define particles of 4 dimensions. Each microservice is referenced by an index. For example, we consider the productpage microservice with index 0, reviews-v3 with index 1, ratings with index 2 and details with index 3. We illustrate the particle model for bookinfo application in Figure 7.

	Particle_i	
	Microservice Index	Number of Pods
productpage microservice →	0	Nb_pods_ms_0
reviews-v3 microservice →	1	Nb_pods_ms_1
ratings microservice →	2	Nb_pods_ms_2
details microservice →	3	Nb_pods_ms_3

Figure 7: Particle model of bookinfo application

In Figure 8. below we present an example of two particles. Particle_0 contains 3 pods for productpage microservice, 1 pod for reviews-v3 and ratings microservices, 3 pods for details microservice. Particle_1 contains 1 pod for productpage microservice, 5 pod for reviews-v3 microservice, 1 pod for ratings details microservices.

Particle_0	
Microservice Index	Number of Pods
0	3
1	1
2	1
3	3

Particle_1	
Microservice Index	Number of Pods
0	1
1	5
2	1
3	1

Figure 8: Example of Particles

The PSO algorithm 1 evaluates each particle using the fitness function. The fitness value helps to decide about the best and which one is better than another by searching pbest and gbest positions as described in the algorithm.

5.2 Fitness function in Docker-C2A

The fitness function helps to evaluate particles and update pbest and gbest positions. We aim to minimize the fitness value to get the best solution. In Docker-C2A, the fitness function depends on the score based on the execution history as depicted in section 4.2 and the cost of the particle. We aim to maximize the score of particles in order to improve the response time and to minimize the cost of the particles to save deployments charges.

The fitness value of each particle P_i is calculated following equation 3 which is the cost of the particle divided by the score of the particle.

$$fitness(P_i) = \frac{Cost(P_i)}{Score(P_i)} \tag{3}$$

If we reduce the cost and increase the score then we minimize the fitness value and obtain a better solution. The particle with the lowest fitness value is the best particle, while it has a low cost with high score. The high score reflects a great improvement of the response time.

The score value of the particle is calculated following equation 4 which is the sum of the score of each microservice MS_k multiplied by the number of pods of the microservice.

$$Score(P_i) = \sum Score(MS_k) * NB\_Pods(MS_k) \tag{4}$$

The cost of the particle is calculated according to equation 5 which is the sum of the cost unit of each microservice MS_k multiplied by the number of Pods of the microservice.

$$Cost(P_i) = \sum Cost(MS_k) * NB\_Pods(MS_k) \quad (5)$$

5.3 PSO Example

In this section, we explain a concrete example of executing PSO algorithm. We consider the bookinfo application with the same workload presented in section 3 which routes requests of reviews microservice to reviews-v3 pods. In this example, we use 3 particles and 3 iterations. We follow the algorithm 1.

	Best position of the swarm gbest	Fitness value of gbest position
After Particle Initialization	Particle_1 Microservice Index / Number of Pods 0 / 2 1 / 2 2 / 2 3 / 1	Fitness = 0,538
After Iteration 0	Particle_0 Microservice Index / Number of Pods 0 / 3 1 / 2 2 / 2 3 / 1	Fitness = 0,444
After Iteration 1	Particle_0 Microservice Index / Number of Pods 0 / 4 1 / 2 2 / 2 3 / 1	Fitness = 0,391
After Iteration 2	Particle_0 Microservice Index / Number of Pods 0 / 5 1 / 2 2 / 2 3 / 1	Fitness = 0,357

Figure 9: Summarize of the execution treatment

In the first step, the algorithm initializes the number of pods needed to the 3 particles randomly. Initially, the best position of each particle pbest is the particle itself. The fitness value of each particle is calculated following the equation 3. When we calculate the cost in fitness function, we consider that all pods have the same cost as 1 unit of cost. In the first step, the fitness values of particles 0, 1 and 2 are ”0.75”, ”0.538” and ”0.714” respectively. The particle with the lowest fitness value is the best particle which is Particle_1 with a fitness value of ”0.538”. PSO algorithm selects Particle_1 as the best particle having the best position gbest in the swarm.

In each iteration, the algorithm calculates the fitness value for each particle, selects the gbest position and moves particles to new positions in order to ameliorate particle positions with better fitness values.

Figure 9. summarizes the execution treatment in this example. This figure illustrates the best position gbest after each iteration and its fitness value. We remember that we use 3 iterations in this example. It is clear that the fitness value of the best solution gbest is improved in each iteration, as the purpose of the algorithm is to minimize this value. In fact, if we minimize the fitness value, we guarantee that the particle position has the minimum cost with significant gain in response time.

At the beginning, the fitness value of gbest was ”0.538”, this value was decreased after each iteration, so our algorithm is improving the solution correctly with the decrease of fitness value. The lowest value of fitness ”0.357” is presented in the final iteration and it is the output of the algorithm. This particle has 5 pods for microservice 0 which is productpage, 2 pods for microservices 1 and 2 which are reviews and ratings microservices respectively, and 1 pod for microservice 3 which is details microservice.

In the next section, we discuss the results of our autoscaler Docker-C2A based on PSO algorithm and execution history using this example and we discuss the difference and the contribution compared to Kubernetes HPA autoscaler.

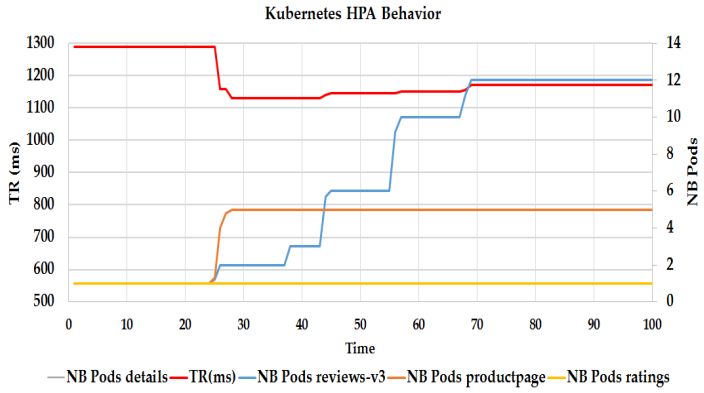
6 Experimentations

In this section, we discuss the impact of our autoscaler Docker-C2A compared to Kubernetes HPA autoscaler.

In our experimentations, we use a cluster Kubernetes composed by 4 virtual machines (VMs). Each VM runs on Ubuntu OS and contains 4 cores CPU, 8GB memory and 100 GB for disk storage. We use the same microservices-based application bookinfo detailed in section 3 composed by 4 microservices: productpage, reviews, ratings and details. Each microservice is deployed in Kubernetes cluster as a deployment containing a set of similar pods. We use the same workload used previously which routes the incoming traffic of reviews microservice to reviews-v3 instances with a flow of 70 Req/s.

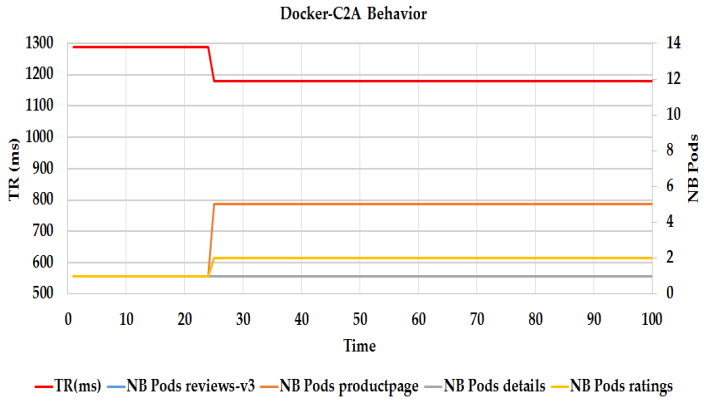
Figure 10 illustrates the difference between Kubernetes HPA and Docker-C2A behaviors. At the time t=24, we launch these two autoscalers. Kubernetes behavior is illustrated in Figure 10a. We detailed the behavior of Kubernetes HPA in section 3 using the workload of bookinfo application. When Kubernetes HPA detects a high load it waits 15 seconds to launch autoscaling actions. In each period of 15 seconds, it calculates and launches the needed number of pods. Every period, it updates the number of needed pods and monitor the situation. Kubernetes HPA stops adding pods when the state of the application becomes stable or it reaches the maximum number of pods. As a result, HPA autoscaler launches 5 pods for productpage microservice, 12 pods for reviews-v3 microservice and 1 pod for both details and ratings microservices. So for this workload Kubernetes HPA launches 19 pods.

Whereas, with Docker-C2A autoscaler, based on the execution history and the PSO algorithm, we can manage the same workload with reduced number of pods. Docker-C2A behavior is illustrated in Figure 10b. When it detects a high workload, it waits 15 seconds and launches the action plan issued by PSO algorithm. In this example, the output of PSO algorithm is presented in Figure 9. The output is the gbest particle of the last iteration. This particle contains 5 pods for productpage microservice, 2 pods for both reviews-v3 and ratings microservices and 1 pod for details microservice. Docker-C2A



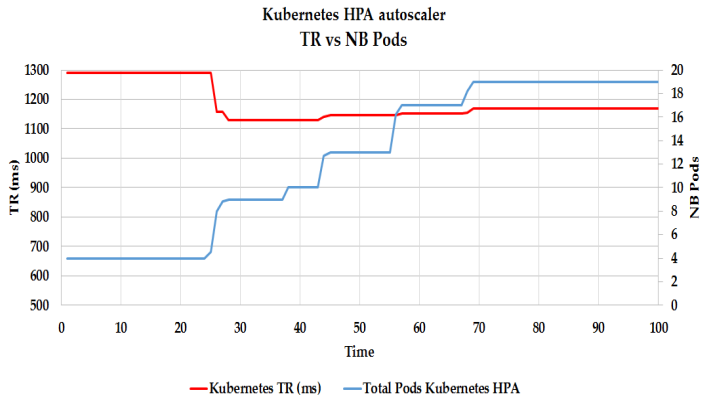
a

(a) Kubernetes HPA Autoscaler behavior



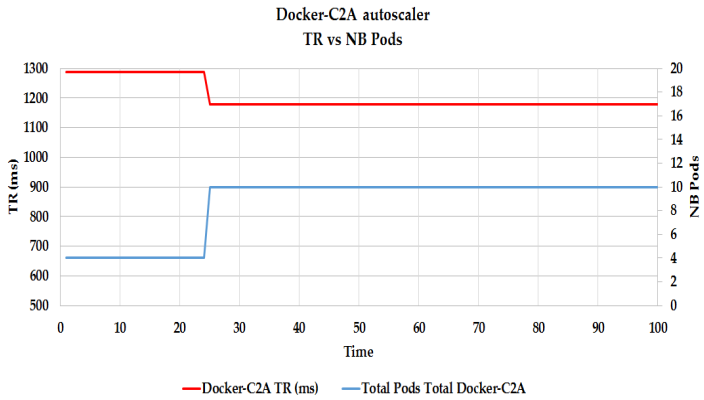
b

(b) Docker-C2A autoscaler behavior



c

(c) Kubernetes HPA Response Time vs NB of Pods



d

(d) Docker-C2A Response Time vs NB of Pods

Figure 10: Kubernetes HPA autoscaler vs Docker-C2A autoscaler

launches in total 10 pods.

If we consider the response time of requests which is presented with red curves in Figures 10a and 10b, we notice that the response time is almost the same in Kubernetes or Docker-C2A autoscalers. While Docker-C2A launches only 10 pods which is 9 pods less than Kubernetes HPA that launches 19 pods. We summarize the number of pods and the response time of these two autoscalers in Figure 11. The response time of requests with Docker-C2A is 9 milliseconds higher than the response time with Kubernetes. This difference is negligible and is not considerable as the difference is appreciated if it is in the order of tens or even hundreds of milliseconds. So, we have certainly reached our goal with Docker-C2A using the execution history and the PSO algorithm which gives an optimized amount of resources with good performance.

Autoscaler	NB Pods Productpage	NB Pods Reviews-v3	NB Pods Ratings	NB Pods Details	Total NB of Pods	Response Time (ms)
Kubernetes HPA	5	12	1	1	19	1170
Docker-C2A	5	2	2	1	10	1179

Figure 11: Kubernetes HPA autoscaler vs Docker-C2A autoscaler

Also, Docker-C2A takes in average 60 milliseconds to generate the decision of scaling, which is a short period of time that does not disturb the elastic treatment. However, Docker-C2A uses a high amount of resources of about 1.2 cores of CPU to process the PSO algorithm. The high resource usage is the cost of problem optimization and among the weak points of our autoscaler. Moreover, to run our algorithm, it is recommended to execute up to 4 iterations as a maximum. Indeed, in our context, PSO algorithm generates erroneous results from 5 iterations.

7 Conclusion and Future Works

In this paper, we presented our autoscaler Docker-C2A a cost aware autoscaler for microservices-based applications deployed with docker containers. Docker-C2A is based on response time metric. It uses the execution history of the application and PSO algorithm. Docker-C2A selects appropriate microservices for scaling and calculates the optimal number of containers for each microservice. We clearly presented the effectiveness of our autoscaler compared to Kubernetes HPA autoscaler using bookinfo application as a real case study. We compared with Kubernetes autoscaler as most of existing autoscalers have the same issues. They neither select appropriate microservices for scaling nor calculate the optimal number of additional containers. As a result, Docker-C2A optimizes computing resources and reduces deployment cost. And that is the main purpose of our autoscaler. However, there are some

limitations of Docker-C2A. In fact, it consumes a high amount of resources to generate the solution. This fact is caused by the PSO algorithm which requires high resources and this is the case of several optimization algorithms.

In future work, we improved Docker-C2A to consume less resources, either by proposing another particle definition in PSO algorithm, or by integrating another optimization algorithm instead of PSO that consumes less resources and generates good results. Also, in future work, we improved the execution history using other performance metrics and more execution data in order to analyze the application behavior in each workload. This greatly helps to estimate the needed resources with more precision and allows better optimization of computing resources.

References

[1] M. H. Fourati, S. Marzouk, K. Drira, M. Jmaiel, "DOCKERANALYZER : Towards Fine Grained Resource Elasticity for Microservices-Based Applications Deployed with Docker," in 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), IEEE, 2019, doi:10.1109/pdcat46702.2019.00049.

[2] J. Lewis, M. Fowler, "Microservices," https://martinfowler.com/articles/microservices.html, march 2014.

[3] "Docker," https://www.docker.com/.

[4] "Kubernetes," https://kubernetes.io/, june 2019.

[5] "Kubernetes Autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, june 2019.

[6] G. Yu, P. Chen, Z. Zheng, "Microscaler: Automatic Scaling for Microservices with an Online Learning Approach," in 2019 IEEE International Conference on Web Services (ICWS), IEEE, 2019, doi:10.1109/icws.2019.00023.

[7] F. Zhang, X. Tang, X. Li, S. U. Khan, Z. Li, "Quantifying cloud elasticity with container-based autoscaling," Future Generation Computer Systems, **98**, 672–681, 2019, doi:10.1016/j.future.2018.09.009.

[8] M. Gotin, F. Lösch, R. Heinrich, R. Reussner, "Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments," in Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ACM, 2018, doi:10.1145/3184407.3184430.

[9] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," Future Generation Computer Systems, **72**, 165–179, 2017, doi:10.1016/j.future.2016.09.002.

[10] C. Guerrero, I. Lera, C. Juiz, "Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture," Journal of Grid Computing, **16**(1), 113–135, 2017, doi:10.1007/s10723-017-9419-x.

[11] G. Yu, P. Chen, Z. Zheng, "Microscaler: Automatic Scaling for Microservices with an Online Learning Approach," in 2019 IEEE International Conference on Web Services (ICWS), IEEE, 2019, doi:10.1109/icws.2019.00023.

[12] "Kubernetes Autoscaler," https://istio.io/latest/docs/examples/bookinfo/, august 2020.

[13] "Istio," https://istio.io/.

[14] IBM, "An architectural blueprint for autonomic computing," 2005.

[15] J. Kennedy, R. Eberhart, "Particle swarm optimization," in Proceedings of ICNN'95 - International Conference on Neural Networks, IEEE, doi:10.1109/icnn.1995.488968.