Journal of Grid Computing (2022) 20: 6
https://doi.org/10.1007/s10723-021-09597-5

EPMA: Elastic Platform for Microservices-based Applications: Towards Optimal Resource Elasticity

Mohamed Hedi Fourati · Soumaya Marzouk · Mohamed Jmaiel

Received: 14 April 2021 / Accepted: 5 December 2021 / Published online: 22 February 2022

Abstract This research paper sets forward an autoscaler, called EPMA, (Elastic Platform for Microservice-based Applications). Its main objective corresponds to resource optimization. The basic contributions of EPMA are twofold. First, an analysis module that correctly detects and identifies the root cause of performance degradation while considering a cross-layer approach. The detected problems may be related to request issues such as a workload increase or a specific request consuming a lot of resources. Other problems related to container and VM (Virtual Machine) layers are also considered. Second, a planning module that considers the above detected problems and offers an optimized elasticity plan avoiding useless resource provisioning and thus permitting resource optimization. Thus, our planning module selects the appropriate microservices to which resources should be added as well as the optimized amount of resources that should be incorporated to correct the situation. Experimental results conducted on two concrete use cases revealed that EPMA autoscaler detects and identifies different problems generating the overload state and launching an optimized resource provisioning that reduces computing resources.

Keywords Microservices · Elasticity · Autoscaling · Anomaly detection · Docker · Kubernetes

M. H. Fourati (✉) · S. Marzouk · M. Jmaiel
ReDCAD Laboratory, ENIS, Sfax, Tunisia
e-mail: mohamed-hedi.fourati@redcad.org

S. Marzouk
FSEGS, Sfax, Tunisia
e-mail: soumaya.marzouk@redcad.org

M. Jmaiel
Digital Research Center of Sfax, Sfax, Tunisia
e-mail: mohamed.jmaiel@redcad.org

1 Introduction

Microservices [34] stand for an architectural style of development consisting of a collection of small independent and loosely coupled components. Nowadays, multiple companies like Amazon, LinkedIn and Netflix have migrated towards microservice architectures in order to increase the efficiency and the scalability of computing resources. Microservices-based applications are deployed in form of many containers hosted in a pool of VMs.

From this perspective, microservices-based applications are often deployed on Cloud, which enables microservices to benefit from Cloud elasticity. In fact, when there is a rise of workload, microservice resources will be overloaded in a way that additional resources should be allocated. In such a case, autoscaling is launched in order to minimize computing resources. The owner of the application does not need to decide when to add resources. The autoscaler automatically provisions essential resources during peak periods using scaling up actions and launches

scaling down actions when there are unused resources. One of the most prominent solutions for microservices elasticity is the use of Docker with Kubernetes.

Docker [1] is a tool designed for creating, deploying and running applications using containers, which allows 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. These containers are orchestrated by Kubernetes [3].

Kubernetes [3] is an open-source container-orchestration system created by Google used for automation of application deployment, scaling and management. Kubernetes manages elasticity based on CPU utilization. Therefore, CPU threshold is specified by the user. In case of exceeding threshold, Kubernetes adds replicas of containers. In our experiments, we use Kubernetes terminology, where a deployment is a microservice, a pod is considered as a container (and it is possible to deploy many containers on the same pod), and a deployment is composed of a set of similar pods.

In literature, there are only a few autoscalers which are elaborated for containers and microservices-based applications [11, 14, 15, 23, 26, 30, 31, 36, 37, 40]. Most of these solutions are threshold based. This implies that, if the threshold of resource usage (e.g. CPU or memory) of a microservice is exceeded, one or many duplicates of the affected containers are directly added. If the threshold is still exceeded, which means that the normal state is not recovered, the autoscaler continues to add duplicates of containers until reaching the maximum number of duplicates.

Such a behavior gives rise to many issues reducing the efficiency of the solution. The first one corresponds to the fact that when thresholds are exceeded, autoscaling is automatically launched; which may be not appropriate in such a situation. Actually, CPU and memory may be highly used but the application is still running correctly and providing acceptable response time. In fact, an investigation performed in [43] corroborates that high CPU usage implies that the container instance is fully utilized, but it may still provide acceptable response time without adding more resources. In such a case, launching autoscaling may lead to allocate unnecessary resources.

The second issue lies in the fact that existing autoscalers suppose that overload is always generated by requests flow rise, which is often correct. However, the overload may be caused by other problems such as specific requests, VM or container issues. Indeed, if the VM hosting the microservice displays execution problems, the microservice may be affected and the overload state will be detected. In such a case, duplicating the microservice is not the appropriate solution and the VM problem needs to be settled. Additionally, a specific request using a large amount of resources, may cause the overload of a microservice. In this situation, the threshold of resources will be exceeded, and scaling-up actions will be launched to integrate other container instances. However, integrating new instances is not the right solution as the specific request does not need other instances but rather needs further resources in the same container during the execution of the specific request. These examples show that launching scaling when detecting overload may lead to provisioning unnecessary resources, which may affect the resource optimization process. Moreover, the problem will persist, and the autoscaler may still add resources.

The third problem, with existing solutions, relates to selecting the microservices which need to be scaled as well as the amount of resources which needs to be added. Basically, classic autoscaling solutions rely on monolithic applications [9, 12, 13, 16, 17, 20, 21, 29, 38] or, in the best case, three tier applications [24, 33]. For these types of applications, selecting the component which should be scaled is not complicated. However, when considering microservice-based applications, the selection of both the microservice which should be scaled and the correct number of instances which should be added becomes complex. This complexity refers to the high number of microservices building up the application as well as the difficulty of locating the microservice causing the problem. From this perspective, a problem caused by a microservice may be propagated to other ones.

In order to handle these issues, this paper presents EPMA autoscaler (Elastic Platform for Microservice-based Applications) which stands for an efficient elastic solution enabling the correct detection and identification of the overload cause as well as running optimized scaling actions while considering the application architecture, application performance and the scaling cost. EPMA autoscaler rests on the autonomic MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop of IBM [27]. The main purpose of EPMA platform lies in resource

optimization. EPMA relies on four modules: Monitor, Analyze, Plan and Execute.

Experimentations, conducted on two real use cases based on microservices, demonstrated that EPMA autoscaler detects and identifies different problems causing the overload state of the application. Moreover, EPMA greatly optimizes computing resources. EPMA can satisfy high workload with minimum computing resources, even through detecting, identifying and solving existing issues or through allocating a small number of containers reflecting the just needed amount of resources. The remainder of this paper is organized as follows: Section 2 introduces the state of the art. It also highlights the background related to this work. Section 3 exhibits EPMA, our proposed autoscaler for microservices-based applications. It details the monitoring module, the analysis module DOCKERANALYZER and the planning module DOCKER-C2A. DOCKERANALYZER detects and identifies abnormal behaviors while DOCKER-C2A builds the adequate action plan to solve identified problems. Section 4 displays the performance of the analysis algorithm DOCKERANALYZER using precision and recall concept. Section 5 depicts the performance of the planning module DOCKER-C2A. Section 6 wraps up the conclusion, crowns the whole work, outlines the major contributions of the research and provides different perspectives for future works.

2 State of the Art

2.1 Background

Elasticity corresponds to one of the major strengths of Cloud computing. It allows users to pay only consumed resources. Elasticity is defined as the ability of a system to adapt automatically to the workload changes through provisioning and deprovisioning resources [25]. There are two types of elasticity: horizontal consisting in adding and removing instances of resources and vertical consisting in adding or removing amounts of resources from existing instances.

In traditional infrastructures, from the beginning, a maximum amount of resources is allocated to the application. In this case, a lot of resources are allocated but not used, and only a little amount of resources is consumed. However, in the context of Cloud computing, resources are allocated as required according to the application workload.

Elastic treatment depends on Cloud managed entities. It can be based on VMs or containers such as Docker technology. Elasticity was basically proposed at the VM level. However, such solution involves slow elasticity response due to important startup time of VMs. Moreover, the efficiency of elasticity depends highly on the way the application is decomposed. Indeed, for monolithic application deployed on a single huge VM, horizontal elasticity corresponds to duplicating the hole VM which may induce unnecessary cost. Therefore, a fine decomposition of the application may enhance the efficiency of elasticity. Within this framework, the use of microservices-based applications deployed with containers became widely used for Cloud applications, especially when Docker technology [1] emerged.

In fact, microservice is an architectural style of development consisting of a collection of small independent and loosely coupled components [34]. Each microservice represents a small component of the application acting as an elementary treatment. Unlike monolithic architecture, where all components are inseparable and considered as a single entity, microservices work in synergy to accomplish the same tasks while being separate and independent.

With a microservice style, we can duplicate only the necessary components without needing to duplicate the whole application as in the case of monolithic architecture. As a result, we add resources with high precision while duplicating only the needed components. Thus, with microservice architecture, we save a lot of resources and avoid charging high cost for application deployment.

2.2 Related Works

Since the birth of Cloud technology, numerous studies have been particularly oriented towards elasticity focusing on VM level. These studies are dedicated to Cloud applications deployed on VMs. However, a few solutions propose autoscalers focusing on container level and are dedicated to microservices-based applications. Among the most prominent container solutions, there are autoscalers that handle non Docker containers such as OSGI [7] containers which have particular usage and cannot be applied to recent

applications that use Docker containers and microservice architecture. Although, autoscalers of Docker containers are fewer than non Docker ones, they are in a progressive growth. In the following, we shall focus on solutions working at the container level and classify them according to the employed provisioning strategy.

2.2.1 Solutions Using Classic Resource Allocation Treatment

Solutions based on classic treatment add a fixed number of instances to microservices where thresholds are exceeded. These solutions do not calculate the needed resources nor do they identify the eligible components for scaling. Additionally, they do not consider the application architecture in scaling treatment.

Kukade et al. [40] elaborated an elastic approach for microservices-based applications using containers. This approach uses reactive strategy based on threshold. It uses the number of requests and memory usage metrics. The autoscaling process is treated as follows: If the requests count or the memory load of the container exceeds the threshold, then a new container is created. If the requests count or the memory load of the container is below the threshold, then the container is scaled down.

Al-Dhuraibi et al. [11] set forward an autonomic vertical elasticity approach for Docker containers based on MAPE-K loop of IBM [27]. It uses CPU and memory usage metrics, and scales up and down the CPU and the memory of each container according to the workload of the application. However, the vertical scaling of containers depends on the available resources in the VM. Indeed, if the VM does not have available resources, the autoscaler cannot launch the vertical scaling. In such a case, a container live migration to another VM with available resources is performed, in order to allow the elastic treatment in the vertical mode.

Ashraf et al. [12] designed CRAMP autoscaler which is a cost aware resource allocation for multiple web applications using proactive scaling. The elastic treatment is applied on web applications (bundles) as well as application servers (VMs). Thanks to OSGi specification [7], each OSGi corresponds to a bundle. This approach uses a hybrid elastic strategy where the predictive strategy is used for VMs and the reactive strategy is used for bundles. Firstly, the system foresees the use of CPU and memory based on prediction and current values, then compares the CPU and memory usage with the used thresholds.

2.2.2 Solutions Calculating Required Resources

In this section, we report works providing solutions enabling the estimation of the quantity of resources that should be added to recover the required QoS (Quality of Service). These solutions suppose that threshold exceeding is always generated by workload increase.

Hoenisch et al. [26] developed a four fold autoscaling approach using Docker containers. The proposed approach, based on resource metrics such as CPU and memory usage, uses horizontal and vertical strategies for both VMs and containers providing four scaling strategies. It uses a multi objective performance model to get the optimal number of VMs and containers as well as the needed resource configuration. This approach proved that a performance model can afford appreciable autoscaling decisions.

Calheiros et al. [14] introduced a QoS driven platform called Aneka for resource provisioning dedicated for elastic applications deployed on a hybrid Cloud. This platform is composed of three levels, namely the infrastructure level composed of servers and Cloud IaaSs deploying containers and applications, the container level which manages containers and allocates the needed resources and, the application level containing interfaces and different APIs allowing developers to manage and control Aneka platform. The elastic treatment in Aneka platform is managed by the middleware which scales up and down containers. The elastic treatment is based on threshold and uses horizontal strategy.

Zhang et al. [44] proposed an autoscaler based on containers and microservices for IoT applications. This autoscaler uses a mathematical model and monitors two primary metrics, which are requests flow and metrics related to resources such as memory and CPU. This autoscaler is based on the execution history of the application which mainly allows to launch suitable actions already recorded. Based on the history, the autoscaler evaluates the adequate number of containers to be added to each microservice. While this solution determines the suitable amount of resources, it does not find out the appropriate microservices for

scaling, which is a difficult problem that can optimize and reduce deployment resources. Guerrero et al. [22] presented a genetic algorithm to manage the elastic treatment and the resource provisioning for containerized applications based on microservices. The main objective of the designed approach is to optimize the assignment of containers in existing VMs as well as the autoscaling of containers. This approach builds up the appropriate model for the entire system such as the needed resources, microservices and the deployed applications. Afterwards, it uses the genetic algorithm to optimize this model and to generate the appropriate requirements for resource allocation and container scaling. The main contribution of this study resides in the use of the application architecture as well as the relationship between components in the resource allocation treatment. Additionally, this solution determines the suitable amount of resources for microservices and containers.

Kan [32] created DoCloud, an elastic Cloud platform dedicated for web applications deployed with Docker containers. In scaling out actions, DoCloud uses proactive and reactive models. In scaling in actions, the platform uses a proactive model. This solution uses resource related metrics such as CPU and memory usage. Experimentations are performed on a single microservice to assess the efficiency of the platform according to the change in the number of containers compared to the variation in the number of requests.

2.2.3 Solutions Defining a Reaction Policy in Case of Overload

We found one container-level study defining a reaction policy in case of overload. This study only detects if there is an issue causing the overload state. It neither identifies the root cause nor resolves detected issues.

Yu et al. [43] proposed Microscaler, an autoscaler for microservices-based applications. This autoscaler selects the appropriate microservices that need to be scaled using service-power, a customized metric based on requests latency. The latter is useful to determine the appropriate microservices for scaling. Microscaler estimates the suitable number of containers for each selected microservice using a heuristic approach and a Bayesian Optimization (BO). Multiple outstanding points were considered in this study. Firstly, the scaling process is based on the response time instead of resource saturation. Secondly, this approach selects eligible components for scaling and calculates the optimal number of containers. However, this solution is dedicated only for applications based on service mesh, which makes this approach not applicable in other contexts.

2.3 Related Works

Since the birth of Cloud technology, numerous studies have been particularly oriented towards elasticity focusing on VM level. These studies are dedicated to Cloud applications deployed on VMs. However, a few solutions propose autoscalers focusing on container level and are dedicated to microservices-based applications. Among the most prominent container solutions, there are autoscalers that handle non Docker containers such as OSGI [7] containers which have particular usage and cannot be applied to recent applications that use Docker containers and microservice architecture. Although, autoscalers of Docker containers are fewer than non Docker ones, they are in a progressive growth. In the following, we shall focus on solutions working at the container level and classify them according to the employed provisioning strategy.

2.3.1 Solutions Using Classic Resource Allocation Treatment

Solutions based on classic treatment add a fixed number of instances to microservices where thresholds are exceeded. These solutions do not calculate the needed resources nor do they identify the eligible components for scaling. Additionally, they do not consider the application architecture in scaling treatment.

Kukade et al. [40] elaborated an elastic approach for microservices-based applications using containers. This approach uses reactive strategy based on threshold. It uses the number of requests and memory usage metrics. The autoscaling process is treated as follows: If the requests count or the memory load of the container exceeds the threshold, then a new container is created. If the requests count or the memory load of the container is below the threshold, then the container is scaled down.

Al-Dhuraibi et al. [11] set forward an autonomic vertical elasticity approach for Docker containers based on MAPE-K loop of IBM [27]. It uses CPU and memory usage metrics, and scales up and down

the CPU and the memory of each container according to the workload of the application. However, the vertical scaling of containers depends on the available resources in the VM. Indeed, if the VM does not have available resources, the autoscaler cannot launch the vertical scaling. In such a case, a container live migration to another VM with available resources is performed, in order to allow the elastic treatment in the vertical mode.

Ashraf et al. [12] designed CRAMP autoscaler which is a cost aware resource allocation for multiple web applications using proactive scaling. The elastic treatment is applied on web applications (bundles) as well as application servers (VMs). Thanks to OSGi specification [7], each OSGi corresponds to a bundle. This approach uses a hybrid elastic strategy where the predictive strategy is used for VMs and the reactive strategy is used for bundles. Firstly, the system foresees the use of CPU and memory based on prediction and current values, then compares the CPU and memory usage with the used thresholds.

2.3.2 Solutions Calculating Required Resources

In this section, we report works providing solutions enabling the estimation of the quantity of resources that should be added to recover the required QoS (Quality of Service). These solutions suppose that threshold exceeding is always generated by workload increase.

Hoenisch et al. [26] developed a four fold autoscaling approach using Docker containers. The proposed approach, based on resource metrics such as CPU and memory usage, uses horizontal and vertical strategies for both VMs and containers providing four scaling strategies. It uses a multi objective performance model to get the optimal number of VMs and containers as well as the needed resource configuration. This approach proved that a performance model can afford appreciable autoscaling decisions. Calheiros et al. [14] introduced a QoS driven platform called Aneka for resource provisioning dedicated for elastic applications deployed on a hybrid Cloud. This platform is composed of three levels, namely the infrastructure level composed of servers and Cloud IaaSs deploying containers and applications, the container level which manages containers and allocates the needed resources and, the application level containing interfaces and different APIs allowing developers to manage and control Aneka platform. The elastic treatment in Aneka platform is managed by the middleware which scales up and down containers. The elastic treatment is based on threshold and uses horizontal strategy.

Zhang et al. [44] proposed an autoscaler based on containers and microservices for IoT applications. This autoscaler uses a mathematical model and monitors two primary metrics, which are requests flow and metrics related to resources such as memory and CPU. This autoscaler is based on the execution history of the application which mainly allows to launch suitable actions already recorded. Based on the history, the autoscaler evaluates the adequate number of containers to be added to each microservice. While this solution determines the suitable amount of resources, it does not find out the appropriate microservices for scaling, which is a difficult problem that can optimize and reduce deployment resources. Guerrero et al. [22] presented a genetic algorithm to manage the elastic treatment and the resource provisioning for containerized applications based on microservices. The main objective of the designed approach is to optimize the assignment of containers in existing VMs as well as the autoscaling of containers. This approach builds up the appropriate model for the entire system such as the needed resources, microservices and the deployed applications. Afterwards, it uses the genetic algorithm to optimize this model and to generate the appropriate requirements for resource allocation and container scaling. The main contribution of this study resides in the use of the application architecture as well as the relationship between components in the resource allocation treatment. Additionally, this solution determines the suitable amount of resources for microservices and containers.

Kan [32] created DoCloud, an elastic Cloud platform dedicated for web applications deployed with Docker containers. In scaling out actions, DoCloud uses proactive and reactive models. In scaling in actions, the platform uses a proactive model. This solution uses resource related metrics such as CPU and memory usage. Experimentations are performed on a single microservice to assess the efficiency of the platform according to the change in the number of containers compared to the variation in the number of requests.

2.3.3 Solutions Defining a Reaction Policy in Case of Overload

We found one container-level study defining a reaction policy in case of overload. This study only detects if there is an issue causing the overload state. It neither identifies the root cause nor resolves detected issues.

Yu et al. [43] proposed Microscaler, an autoscaler for microservices-based applications. This autoscaler selects the appropriate microservices that need to be scaled using service-power, a customized metric based on requests latency. The latter is useful to determine the appropriate microservices for scaling. Microscaler estimates the suitable number of containers for each selected microservice using a heuristic approach and a Bayesian Optimization (BO). Multiple outstanding points were considered in this study. Firstly, the scaling process is based on the response time instead of resource saturation. Secondly, this approach selects eligible components for scaling and calculates the optimal number of containers. However, this solution is dedicated only for applications based on service mesh, which makes this approach not applicable in other contexts.

3 EPMA: Elastic Platform for Microservices-based Applications

EPMA autoscaler is an **E**lastic **P**latform for **M**icroservice-based **A**pplications based on the autonomic MAPE-K loop of IBM [27]. It aims to enable resource optimization. This loop, illustrated in Fig. 1, relies upon four modules sharing Knowledge: Monitoring, Analysis, Planning and Execution.

The Monitoring module collects data from different layers: application, microservice, container and VM layers. The collected parameters are related to resources such as CPU usage and memory usage, as well as application performance such as response time. This module correlates and filters these parameters in order to determine symptoms that need to be analyzed. An overload state can be a symptom generated by the monitor module.

When a symptom is generated, DOCKERANALYZER, the analysis module, will be invoked to check if this symptom is caused by a normal state like the increase in requests flow or by an abnormal behavior resulting from a VM or a container problem. Furthermore, DOCKERANALYZER sends a change request to the planner in order to trigger the generation of a plan allowing the resolution of the problem.

The planning module, called DOCKER-C2A, generates the appropriate change plan describing the desired set of changes, and moves that change plan to the execution function. Moreover, DOCKER-C2A selects the adequate microservices that require additional resources and calculates the optimal number of containers to be added to each microservice.

The execution module will apply, in each layer, the change plan proposed by the planner. The Knowledge is shared by these four modules and involves particular types of data such as symptoms, policies, change requests and change plans.

In what follows, we should remind that a microservice is composed of a set of similar containers, and containers are deployed in VMs.

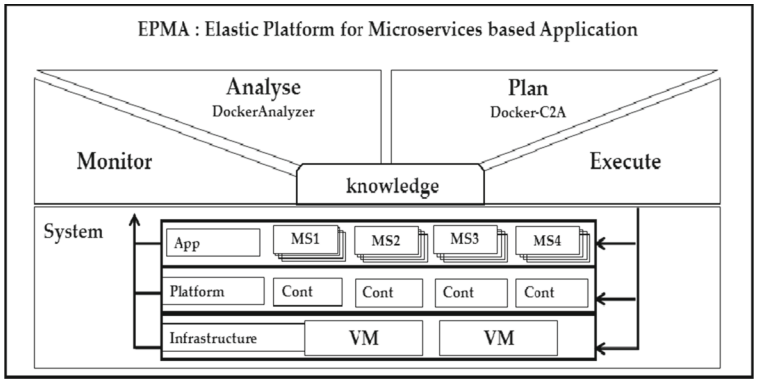


Fig. 1 EPMA : Elastic Platform for Microservices-based Applications

3.1 Monitoring Module

3.1.1 Monitoring Description

Monitoring data should be collected from different layers (Application layer, microservice layer, container layer and VM layer). The application layer is the frontend microservice of an application which is the first receiver of requests. The response time of the application corresponds to the response time of the frontend microservice. In EPMA platform, we consider the set of metrics illustrated in Table 1 related to each layer: application, microservice, container and VM.

Unlike many autoscalers acting on the application and container levels (such as Kubernetes autoscaler), EPMA autoscaler considers VM metrics which helps to a great extent to identify precisely the root cause of anomalies in the analysis phase.

Besides, like Kubernetes, many autoscalers consider only metrics related to resources such as CPU usage or memory usage in monitoring. However, using exclusively such metrics in autoscaling treatment is not effective and may lead to allocate unnecessary resources. Indeed, as it is proven in [43], a high CPU usage implies that the container instance, the pod in our case, is fully utilized, but it can still provide acceptable response time without adding more resources. Hence, it is possible to handle requests and get acceptable response time even if we have saturated resources with high CPU usage. In addition, according to [35], the correlation between CPU usage and response time might not be strong, and the trigger of autoscaling should be driven by the response time. This refers to the fact that we can have a well satisfied SLA with response time metric while the CPU usage is too high and triggers unnecessary scaling actions. This also corresponds to the reason for which many elastic solutions tend to use response time metric to benchmark their autoscaler. For all these reasons, EPMA autoscaler considers in addition to resource metrics the response time metric.

Table 1 Monitoring metrics

Layer	Metrics
Application	Number of Requests
	Response Time
Microservice	Number of Requests
	CPU usage
	Memory usage
Container	Number of Requests
	CPU usage
	Memory usage
VM	CPU usage
	Memory usage

The monitoring component correlates and filters these parameters in order to determine symptoms that should be analyzed. When a symptom is generated, the analysis component will be triggered. Figure 2. reveals monitoring and analysis components and the relationship between them.

As demonstrated in Fig. 2, the filtering process is performed using four microservices as a microservice for each layer.

Each microservice filters the related metrics:

- Application layer: filtering number of requests (No. Req) and response time (RT).
- Microservice (MS) layer: filtering resource related metrics such as CPU and RAM.
- Container layer: filtering number of requests and resource related metrics useful for outlier detection in analysis phase. This layer contains the history data which represent the evolution of resources according to the number of requests. It is expressed in terms of the corresponding equation: Resources = f (No. Req).
- VM layer: filtering resource related metrics.

When the monitoring generates a symptom due to resource violation and threshold exceeding, the analysis component will be triggered.

3.1.2 Monitoring Implementation

The monitoring module monitors all layers and gets data from each layer at real time. In order to collect these data, we use sysdig monitoring system [10] which is used for microservices-based applications. It is dedicated for Cloud applications to monitor all layers such as VMs, containers, microservices, applications. Almost all metrics can be monitored by sysdig. Several proposed metrics are related to CPU, memory, response time, requests flow, network, storage, etc.

Sysdig supports the Kubernetes tool [3]. It monitors all related components such as pods, deployments,

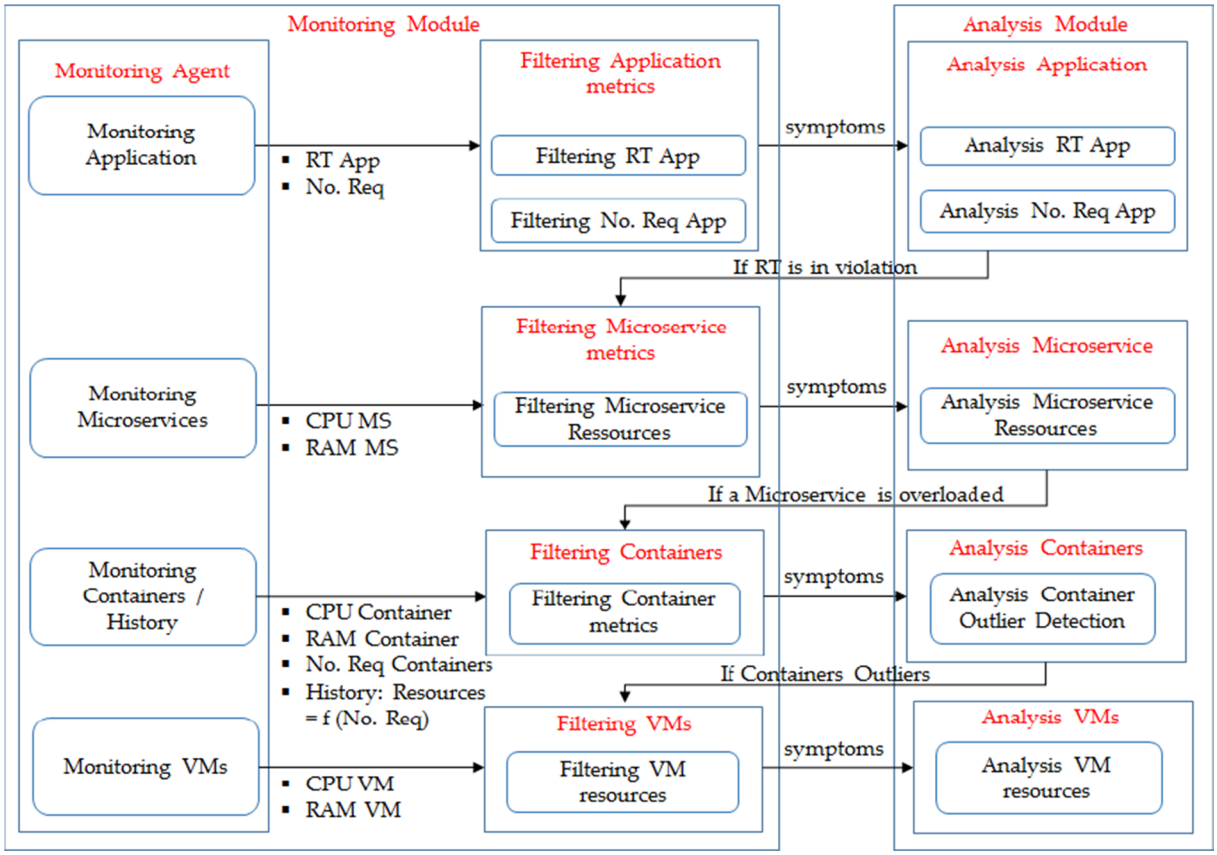


Fig. 2 Monitoring and Analysis Modules

services, replicasets, nodes, etc. Additionally, it is flexible and easy to integrate to many Cloud infrastructures. For instance, it can be used in Google Cloud, IBM Cloud and private Cloud. In our case, a private Cloud based on OpenStack [6] is used; and a sysdig agent is installed in each VM.

In addition, a NoSql database Elasticsearch [2] is used to record needed data for each layer. The monitoring database is composed of four tables as one table for each layer: application, microservice, container and VM layers. For each layer, we record the related data at real time using a time interval of ten seconds. Using the container layer, we define our history database which contains data related to the behavior of each container.

3.2 Analysis Module: DockerAnalyzer

In this section, we present in details DOCKERANALYZER, the analysis module of EPMA autoscaler. An earlier version of our analysis module using a small part of the decision tree and simple experimentation is displayed in [18]. The latter illustrates a basic decision tree for DOCKERANALYZER, while in this work we present a more developed decision tree that encompasses the majority of issues that can arise for a microservices-based application.

The main contributions of our analysis module are (1) the precise detection of the root cause of the violation which is usually considered, in previous works, such as the increase in requests flow (2) the consideration of all stacks including microservices, containers and VMs.

As inferred from Fig. 2, the analysis module is composed of four microservices - one microservice per layer:

- Application layer analyzer: determines the root cause of generated symptoms.
- Microservice layer analyzer: analyses the root cause of resource violation at microservice level.
- Container layer analyzer: analyses and detects containers which have an abnormal behavior in

comparison to the history. These containers are called outliers.
– VM layer analyzer: analyses the VM state in case of detecting containers outliers.

These microservice components are launched when a symptom is generated by the monitoring module, and they cooperate to execute the decision tree plotted in Fig. 3.

The output of the decision tree corresponds to a set of states of the application that can be useful for the planning component. These states are:

– Increase in requests flow: in order to recommend the planner to add additional resources.
– Underloaded microservices: to recommend the planner to launch scaling down actions to reduce computing resources.
– No action recommendation: to recommend the planner not to take any action for the current state of the application.
– A problem at the container level.
– A problem at the VM level.
– A specific request consuming a lot of resources.

In order to generate these states, the decision tree involves two main processes: OUTLIERDETECTOR and ANOMALYIDENTIFIER. The OUTLIERDETECTOR is responsible for detecting abnormal behaviors. An abnormal behavior corresponds to an outlier. A container is considered as an outlier if it consumes an excessive amount of resources compared to its usual resource usage. If the container is not an outlier, it is considered as having a normal behavior. In the Decision Tree illustrated in Fig. 3, there are two nodes "Pod outliers?" and "Request outliers?" that are represented twice in the figure. Both nodes are related to the process OUTLIERDETECTOR. In fact, the same process is launched to detect pod outliers and request outliers.

In case the OUTLIERDETECTOR detects an outlier, the ANOMALYIDENTIFIER is launched to identify the root cause problem of this behavior. The ANOMALYIDENTIFIER represented in Fig. 5, is a basic part of the Decision Tree and is represented twice in Fig. 3.

The first step of the decision tree relates to the analysis of response time and requests flow metrics. If one of these two metrics is violated, the analyzer launches two separate treatments to check if there are overloaded resources: one for microservices in horizontal scaling and the other for microservices in vertical scaling. In fact, each microservice has its own scaling mode: horizontal or vertical. Horizontal mode adds duplicates of containers, whereas vertical mode adds resources for containers from available VM resources.

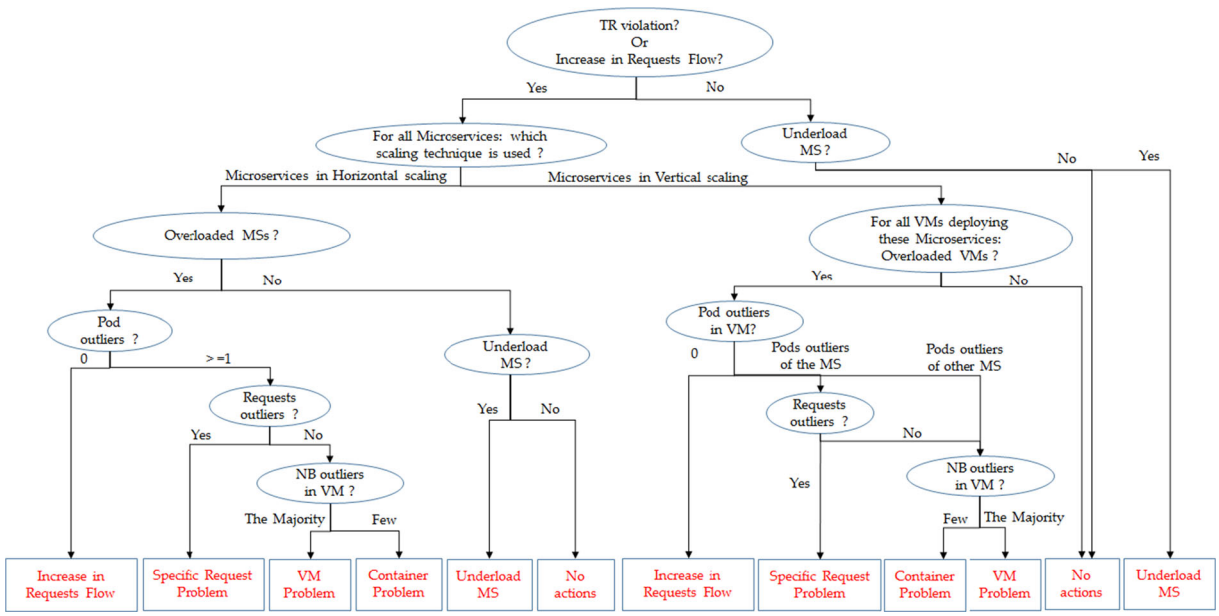


Fig. 3 Decision Tree: DOCKERANALYZER

We start with the first branch treating microservices in horizontal scaling. At this stage, the analyzer focuses mainly on microservices since horizontal scaling deals with adding instances to microservices. Therefore, the analyzer checks if there are overloaded microservices. In case of overload, the analyzer launches outlier detection for each overloaded microservice so as to detect pods outliers. If there is no pod outlier, there is no abnormal behavior and the overload state is caused by the increase in requests flow. In case the analyzer detects pods outliers, it launches the ANOMALYIDENTIFIER process explained in Section 3.2.2 to identify the root cause problem. If there are no overloaded microservices, the analyzer checks if there are underloaded microservices. In such a case, the analyzer generates underload state of these microservices to recommend the planner to reduce computing resources. Otherwise, no action is recommended.

Subsequently, we explain the second branch which handles microservices in vertical scaling. In this case, the whole analysis will focus on VMs as vertical scaling success is highly related to the VM state. At this stage, for each considered microservice, the analyzer checks if there are overloaded VMs deploying the pods of the microservice. In such a case, the analyzer checks if there are pods outliers in each VM. If there is no pod outlier, the overload state is considered as caused by the increase in requests flow. If there are pods outliers belonging to the microservice under treatment, the analyzer launches requests outlier detection. The next steps following requests outlier detection are elucidated in ANOMALYIDENTIFIER process to identify the root cause problem. In case we have pods outliers belonging to other microservices, we consider the number of pods outliers in the VM. If the majority of pods are outliers, the root cause is a VM problem. Otherwise, if there are few pods outliers, the root cause is a container problem.

The second part of the decision tree is launched when there is no violation in response time and requests flow metrics. In this case, the analyzer checks if there are underloaded microservices. In case of underload, the analyzer generates underload state of these microservices to recommend the planner so as to reduce computing resources. Otherwise, no action is recommended.

In the following, we shall provide further details about these two main processes of DOCKERANALYZER: OUTLIERDETECTOR and ANOMALYIDENTIFIER.

3.2.1 OutlierDetector

OUTLIERDETECTOR is responsible for detecting abnormal behaviors of pods or requests.

A request is considered as an outlier when it consumes an excessive amount of resources compared to its usual resource usage. For example, if a given request usually consumes 0.005 cores of CPU. If suddenly this request consumes 1 core CPU, we conclude that it consumes a high amount of resources which highly exceeds the usual amount of CPU. This request is considered as an outlier request. In this section, we will explore the case of pod outlier. The solution adopted for request outlier is similar. We recall that a microservice is composed of a set of similar pods; if one of these pods has an abnormal behavior, then we conclude that this microservice is overloaded due to an abnormal behavior.

To identify abnormal behaviors in pods, we use an outlier detection technique [8]. A pod is considered as an outlier when it has an abnormal behavior according to its history.

For example, if we consider a pod called frontend, according to its history, this frontend consumes 5 millicores CPU when it treats 100 req/s; if suddenly this pod consumes 1 core CPU with the same number of requests, frontend pod will be considered as an outlier while having an abnormal behavior according to the history.

Our history is generated and updated at real time. For each number of requests, we saved CPU usage, memory usage, and response time of the pod. For each record, we check if it's an outlier; if it's the case it won't be saved in the history. Our outlier detection is based on the modified z-score [28]. Modified z-score is a robust method for outlier detection with MAD denoting the median absolute deviation and $\bar{x}$ denoting the median. We calculate the Modified Z-score for each pod according to the following formula:

$$Mi = \frac{0.6745.(xi-\bar{x})}{MAD}$$

If the score Mi is greater than the threshold, then the value xi (CPU usage of the pod in our case) is considered as an outlier. The default threshold of score is fixed by the authors of the method to 3.5 and it can be modified according to experiments. This implies that every point with a score above 3.5 will be considered as an outlier. Algorithm 1 which is plotted below clarifies our outlier detection using modified z-scores method. In this algorithm, we consider CPU

usage metric; other metrics such as memory usage can be treated similarly. This algorithm receives as input the deployment (microservice) to be processed, current_request_number and current_cpu_usage. As a first step, we browse pods constituting the deployment one by one to check if it's an outlier. For each pod, we retrieve values of cpu_usage from the history based on the current number of requests. Afterwards, we calculate the median $\bar{x}$ and the median absolute deviation *MAD* of CPU usage. Next, we deduce the *Mi* score of the pod using current_cpu_usage. We fixed the threshold_outlier initially at 3.5 value. Based on a series of tests, we may update this value. For cpu_usage, we define threshold_outlier_cpu. If the score *Mi* is greater than the threshold_outlier_cpu, the pod will be considered as an outlier and will be added to pods_outliers list.

Algorithm 1: Outlier Detection.

Data: *deployment, current_request_number, current_cpu_usage*
Result: *pods_outliers*
for *pod in deployment* **do**
  $cpu\_usage = get\_cpu\_usage(pod, current\_request\_number)$
  $\bar{x} = median\ of\ cpu\_usage$
  $MAD = median(|cpu\_usage - \bar{x}|)$
  $Mi = \frac{0.6745.(current\_cpu\_usage - \bar{x})}{MAD}$
  if $Mi > threshold\_outlier\_cpu$ **then**
    pod is outlier
    add pod to pods_outliers

Figure 4. illustrates a concrete example of pod (container) outlier. It displays an extract from the history of the pod where we find normal resource usage for each flow of requests. In this example, the history demonstrates that the normal usage of resources in 120 Req/s is 0.003 cores CPU, whereas the current value of the pod for 120 Req/s is 0.1 cores CPU. The difference is 0.097 cores CPU, which is a considerable amount of resources indicating that this pod is an outlier.

3.2.2 AnomalyIdentifier

The major target of the ANOMALYIDENTIFIER is to identify the root cause of the outlier causing the abnormal behavior of the application.

Three main problems can be identified by the ANOMALYIDENTIFIER:

- Container problem
- VM problem
- Specific Request problem

VM and container problems may saturate resources and lead to an overload state and response time violation.

In addition, specific request problem may saturate the resources and generate outliers. To illustrate this problem, we take the example of an IoT application exhibiting a method in a microservice that controls and checks the state of the whole IoT system. The call to this method is not frequent and can be considered as a specific request as it uses a large amount of resources which depend on the size of the IoT system including the number of sensors and the amount of data. When the specific request is launched in a container (pod), the threshold of microservice resources will be exceeded. This refers to the fact that this microservice involves a container that consumes a very large amount of resources and makes the average of microservice resources exceed the threshold. When the specific request finishes, the resource usage decreases and the normal state will be resumed.

ANOMALYIDENTIFIER is based on a decision tree, portrayed in Fig. 5. The leaves of this tree stand for

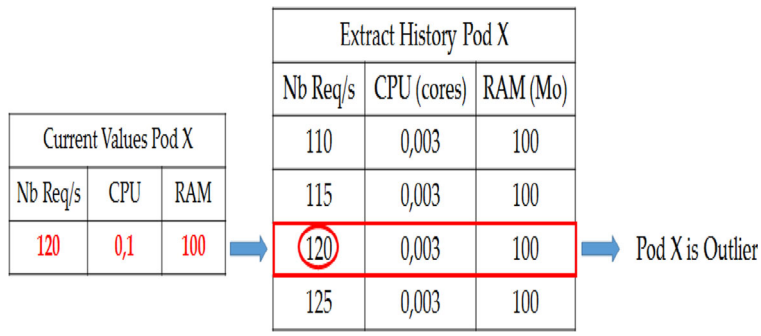


Fig. 4 Pod Outlier

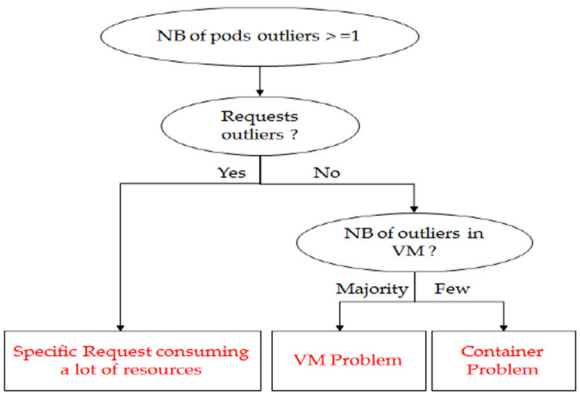


Fig. 5 Decision Tree: ANOMALYIDENTIFIER

the problems representing possible causes of abnormal behaviors.

The first input of the decision tree is the result of OUTLIERDETECTOR corresponding to pods_outliers list. If we have at least one pod outlier, we launch another outlier detection related to requests.

The second outlier detection rests on requests and is applied to each pod outlier. A request is considered as an outlier when it consumes an excessive amount of resources compared to its usual resource usage. For example, if a given request usually consumes 0.005 cores of CPU. If suddenly this request consumes 1 core CPU, we conclude that it consumes a high amount of resources which highly exceeds the usual amount of CPU. This request is considered as an outlier request.

If we detect an outlier request, the cause of the abnormal behavior is a specific request consuming a lot of resources. Otherwise, if we do not detect outlier requests, we consider the number of pods outliers in the VM. If the majority of pods are outliers, the root cause is a VM problem. Otherwise, if there are particular pods outliers in the VM, the root cause is a container problem.

Thus, ANOMALYIDENTIFIER helps identify the root cause of the outliers causing the abnormal behaviors.

3.3 Planning Module

This section addresses the planning module DOCKER-C2A of EPMA autoscaler. DOCKER-C2A is triggered by the analysis module DOCKERANALYZER when a problem is identified. Many problems may be identified by DOCKERANALYZER such as the increase in requests flow, VM problem, container problem and specific request problem. DOCKER-C2A proposes an action plan for each problem in order to resolve it and make the application recover an acceptable execution state. The main contributions of DOCKER-C2A are summarized as follows: (1) The proposal of a cross-layer plan in elastic treatment, thus overcoming existing work shortcomings considering only the resource provisioning issue. (2) The optimization of resource provisioning through the use of PSO (Particle Swarm Optimization) algorithm while considering application performance and computing resources. (3) The proposal of a plan for the specific request problem enabling its execution while providing an acceptable execution time without consuming useless resources.

As a matter of fact, DOCKER-C2A, the planning component in EPMA, proposes an action plan for the four problems identified by the analysis component DOCKERANALYZER, which are:

- Container problem
- VM problem
- Increase in requests flow
- Specific Request problem

Table 2. reveals the action plan for each problem. In fact, if a container problem is identified, it is recommended to restart the container. If the problem

Table 2 Planning Actions: DOCKER-C2A

Planning Actions: DOCKER-C2A		
Problem		Action Plan
Container Level		Restart / Re-creation of the affected containers
VM Level		Restart VM + Migration of hosted containers
Request Level	Increase in Requests flow	Horizontal Scaling using execution history + PSO algorithm
	Specific Request	Vertical Scaling + Isolation Action Plan

persists, the container is deleted and re-created. If a VM problem is identified, containers hosted in this VM will be launched on another VM. The affected VM will be restarted as a first step for resolving problem. If the problem persists, this VM will be ignored and will no longer receive and deploy containers.

In case we have an increase in requests flow, DOCKER-C2A launches resource provisioning process to allocate the required resources for each microservice. From this perspective, we set forward a solution [19] based on the execution history and PSO algorithm, enabling the selection of microservices for horizontal scaling and the optimization of the number of added instances.

In case of specific request problem, DOCKER-C2A launches an isolation action plan for the container executing this request in conjunction with a vertical scaling treatment.

Indeed, specific request is a request which is occasionally enacted and consumes a lot of resources. When the specific request is launched in a container, the threshold of resources are exceeded and the microservice becomes overloaded.

To better and deeper explore this problem, we present in the following the behavior of Kubernetes autoscaler when launching the specific request.

3.3.1 Kubernetes Autoscaler HPA Behavior

In this section, we explain the behavior of Kubernetes autoscaler while launching the specific request problem. We present the behavior of Kubernetes since most of autoscalers display the same behavior towards specific requests. This autoscaler is called HPA: Horizontal Pod Autoscaler [4].

HPA is implemented as a control loop. It computes in every period the ratio between the average usage among available containers and the target CPU utilization, and adds containers if the ratio exceeds the threshold mentioned in the SLA.

In case of exceeding threshold, Kubernetes HPA launches the scaling up of containers, while considering that this exceeding is caused by the increase in the arrival rate of requests. Yet, this exceeding can be generated by multiple other problems such as specific requests that do not need to add duplicates of containers. This refers to the fact that Kubernetes autoscaler HPA does not analyze each situation to check if this exceeding is generated by a problem leading to the abnormal behavior or is caused by an increase in the number of requests. The frailty of Kubernetes autoscaler lies in the fact that there is no analysis phase allowing to know about the cause of resource violation, through detecting and identifying the root cause of anomalies if they exist.

We illustrate in Fig. 6 the behavior of Kubernetes autoscaler when the specific request is injected. Orange curve represents the CPU usage of the deployment which stands for the average of CPU usage of all pods of the deployment. Blue curve represents the current number of pods in the deployment. At time t=10, we inject the specific request in the microservice and we can detect the behavior of the autoscaler.

If a specific request is injected in the deployment, this request will be routed to one pod of the set according to the availability of its resources. This pod will have an outstanding increase in resource usage, which is CPU usage in our case. The average CPU of the deployment is equal to the sum of CPU usage of each pod divided by the number of pods as it is expressed by the following formula:

$$average\_cpu\_usage\_deployment = \frac{\sum cpu\_usage\_of\_each\_pod}{number\_of\_pods} \quad (1)$$

As we notice in Fig. 6, the average CPU usage of the deployment increases considerably and exceeds 80% of CPU demanded when the specific request is injected. In this case, Kubernetes autoscaler considers that this pod needs more resources. For this reason, its autoscaler adds many duplicated pods in order to decrease CPU consumption. From this perspective, the number of pods increases and reaches 10 pods since 10 is the max number of pods we have configured in the autoscaler. As a consequence, the average CPU usage decreases and reaches the normal state while the denominator's value of the average formula above, representing the number of pods, has increased. However, the problem is not really settled owing to

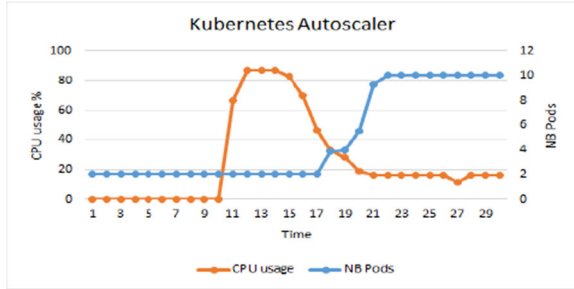


Fig. 6 Kubernetes Autoscaler

the fact that the pod hosting the specific request still needs unplanned resources which may be not available in the current VM. Furthermore, the added instances are under-loaded which may result in unnecessary computing resources.

Unlike the Kubernetes approach, our EPMA autoscaler detects and identifies the origin of the overload state such as the specific request problem using DOCKERANALYZER. It also launches the suitable action plan using DOCKER-C2A planner.

3.3.2 Action Plan for Specific Request Problem

When DOCKERANALYZER identifies the specific request problem, DOCKER-C2A generates the suitable actions dedicated to fix this issue. Indeed, adding more resources in an horizontal scaling mode such as in Kubernetes case will not resolve the problem, since it decreases the average resource usage while the pod executing the specific request is still overloaded.

Therefore, as a first step, the considered pod will switch to a vertical scaling mode, which makes DOCKER-C2A allocate additional resources to this pod with vertical scaling treatment. Hence, DOCKER-C2A launches an isolation action plan.

The isolation action plan rests upon two actions:

- The first action is to ban the pod outlier from executing other requests until the specific request finishes and, thus, the pod will be dedicated to the specific request. This action is launched to avoid disturbing new incoming requests for this pod and to keep a good performance of these requests.
- The second action is to eliminate the pod outlier in the calculation of the average CPU of the deployment which follows the formula (1) in order to keep the average CPU stable without violation while performing the specific request. This seems to have a paramount importance in terms of avoiding redundant triggering of the analysis module during the execution of the specific request.

As the specific request finishes, these actions will be disabled.

4 Performance of Analysis algorithm

This section assesses the performance of DOCKERANALYZER using precision/recall concept [39]. In fact, we try to characterize how the algorithm identifies problems correctly. For this purpose, we conduct a set of experiments, on the IoT application presented in Section 3.2.2, using problem injection and calculate precision/recall values to measure the relevance of our algorithm.

4.1 Precision/Recall Concept

Precision / Recall concept is invested in numerous domains such as pattern recognition and information retrieval. **Precision** (or positive predictive value) is the portion of relevant items among the set of proposed items. **Recall** (or sensitivity) is the portion of relevant items proposed among all relevant items. These two concepts are therefore based on an understanding and measure of relevance.

We have two classes of elements: positive and negative. Precision and recall measures focus on how the algorithm identifies the positive class. In case we need to focus on the negative class, we should use another metric such as specificity.

In a binary classification, specificity is a statistical measure of performance commonly used in medicine. In a diagnosis test, specificity is a measure of a test's ability to identify true negatives. It is the percentage of true negatives of all samples not having the condition, namely: true negatives and false positives. As a matter of fact, specificity (rate of true negatives) calculates the proportion of correctly identified negatives.

4.2 Problem Injection

In order to test the performance of our analyzer, we inject artificial problems. We inject all possible problems which are specific request problem, VM problem, container problem, and high request flow problem.

In our experiments, we use a Kubernetes cluster composed of 4 VMs. All applications are deployed in this cluster. Each application comprises a set of microservices. A microservice involves a set of similar containers (pods).

Using this cluster, we can deploy an average of 5 pods in each VM. Therefore, we can deploy almost 20 pods in total. Figure 7. provides an overview of this cluster.

Our experiments are grounded on several samples for each type of problem. Each problem is injected by launching a specific process.

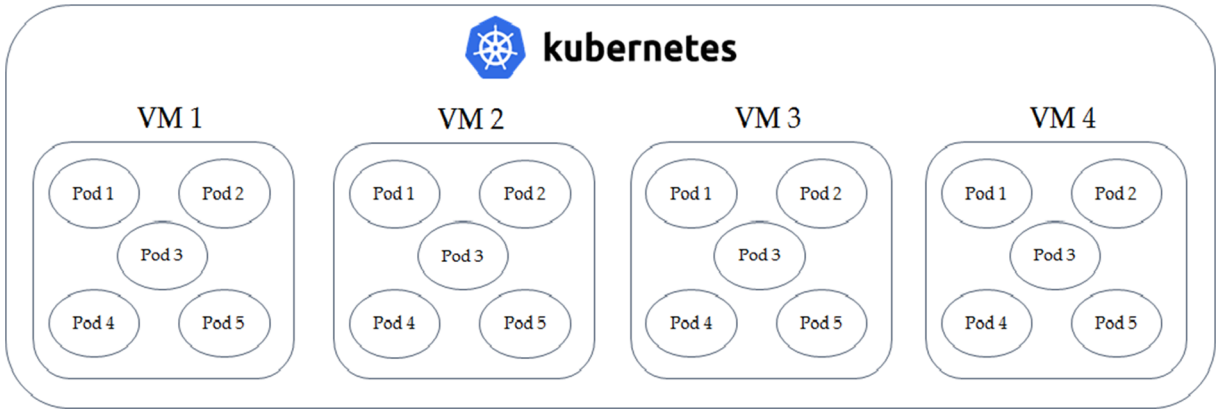


Fig. 7 Overview Kubernetes cluster

Specific request problem is injected by launching the specific request on a given pod. The specific request is a special request launched once for a long period and consumes a large amount of resources.

To inject VM problem, we stress the VM to generate high CPU usage. In fact, authors in [41, 42] argued that there are many issues leading to high resource usage such as CPU and memory usage. In our case, we consider CPU metric. As a result, VM problem disrupts the smooth running of containers and leads to generate pods outliers involving the abnormal behaviors.

To inject container problem, we stress the container to generate high CPU usage. This fact leads to generate an abnormal behavior and a pod outlier.

High request flow problem is injected while executing high request flow on a given microservice. This problem saturates containers and generates overloaded microservices.

4.3 Precision/Recall of DockerAnalyzer

This section tackles the performance of our analyzer DOCKERANALYZER using Precision/Recall concept. For this purpose, we perform a set of experiments where we inject artificial problems to check if the analyzer identifies correctly the injected problems.

Our experiments rest on three scenarios containing a set of problems.

- In **scenario 1**, we inject seven specific requests in different pods chosen arbitrarily. In addition, we inject a VM problem in two VMs of our cluster.
- In **scenario 2**, we inject a container problem on six pods arbitrarily and a VM problem on two VMs.
- In **scenario 3**, we inject a container problem in five pods and a specific request in four pods.

For each scenario, we calculate precision, recall and specificity percentages to assess the performance of our analyzer. We use the Kubernetes cluster handled in the previous section containing four VMs. We use also the following abbreviations:

- P_{ij}: Pod_j deployed in the VM_i.
- $vm\ \ prob$: VM problem
- $cont\ \ prob$: container problem
- $spec\ \ req$: specific request problem.

Tables 3, 4 and 5 highlight the injected problems for each scenario and the related precision, recall and specificity percentages. For each VM_i, we find the corresponding pods: P_{i1}, P_{i2}, P_{i3}, P_{i4}, P_{i5}.

Table 3. depicts results related to the first scenario where we combine the VM problem with the specific request problem. The precision is about 80% which is an acceptable value. This value corroborates that our analyzer identifies the injected problems correctly but sometimes confuses between VM and specific request problems. In fact, when the virtual machine deploys many pods executing the specific request such as VM_4, the other pods are disturbed and overloaded while the VM cannot satisfy the requested resources. As a consequence, these pods are detected as outliers due to a VM problem, which is not correct and decreases the accuracy of our algorithm. However,

Table 3 Experimentation Results: Scenario 1

Scenario 1: VM problem + specific request problem						
$VM_1(vm\ prob)$	$VM_2(vm\ prob)$	VM_3	VM_4	Precision	Recall	Specificity
$P_{11}(spec\ req)$	$P_{21}(spec\ req)$	$P_{31}(spec\ req)$	$P_{41}(spec\ req)$	80%	100%	77.77%
P_{12}	P_{22}	$P_{32}(spec\ req)$	P_{42}			
P_{13}	P_{23}	P_{33}	P_{43}			
P_{14}	P_{24}	P_{34}	$P_{44}(spec\ req)$			
P_{15}	P_{25}	P_{35}	$P_{45}(spec\ req)$			

such scenario is not frequent since specific requests are launched occasionally. Additionally, we have high recall in this scenario while the analyzer identifies correctly all VM problems. Moreover, we have an appreciable specificity explaining that the specific request problem is almost correctly identified.

Table 4. records scenario 2 related results. This scenario combines container and VM problems. The precision is about 70%, which explains that the algorithm, in some cases, identifies the container problem as a VM problem. In fact, as we can infer from Table 4, VM_3 contains three pods with a container problem, and the analyzer considers that there are several pods with a container problem compared to the total number of pods. As a result, a VM problem is identified and is considered as the root cause of the container problem, which is not correct and decreases the precision of the analyzer. In this scenario, we have a high recall, which means that all identified VM problems are correct. However, the specificity is almost average suggesting that the container problem is not correctly identified.

Table 5. sums up results related to the third scenario. This scenario combines container and specific request problems. We have a high precision amounting to 80%. In fact, the small decrease in the precision implies that the algorithm does not correctly identify the injected problems. Indeed, in VM_1 we have three injected problems, namely two pods executing the specific request and one pod containing a container problem. The analyzer considers that the machine contains several problems due to a VM problem, which is not correct. A high recall is appreciated in this scenario indicating that the container problem is correctly identified. Additionally, there is a high specificity with a low error, which reveals that the specific request problem is almost correctly identified.

Furthermore, our analyzer correctly identifies the high request flow problem which is not confused with other problems. In fact, all previous problems are initially detected as outliers and then identified. In case we inject the high request flow problem, the analyzer finds overloaded microservices with no pod outlier. Hence, there is no abnormal behavior and none of the preceding problems is identified. As a result, a high request flow is detected as having a normal behavior.

5 Performance of Docker-C2A Planner

In order to highlight the merits of EPMA autoscaler, this section displays experimentations using real use

Table 4 Experimentation Results: Scenario 2

Scenario 2: VM problem + container problem						
$VM_1(vm\ prob)$	$VM_2(vm\ prob)$	VM_3	VM_4	Precision	Recall	Specificity
P_{11}	P_{21}	$P_{31}(cont\ prob)$	$P_{41}(cont\ prob)$	76.92%	100%	40%
P_{12}	P_{22}	$P_{32}(cont\ prob)$	$P_{42}(cont\ prob)$			
P_{13}	P_{23}	$P_{33}(cont\ prob)$	P_{43}			
P_{14}	P_{24}	P_{34}	P_{44}			
P_{15}	P_{25}	P_{35}	P_{45}			

Table 5 Experimentation Results: Scenario 3

Scenario 3: container problem + specific request problem						
VM_1	VM_2	VM_3	VM_4	Precision	Recall	Specificity
$P_{11}(spec\ req)$	$P_{21}(cont\ prob)$	$P_{31}(cont\ prob)$	$P_{41}(cont\ prob)$			
$P_{12}(spec\ req)$	$P_{22}(spec\ req)$	P_{32}	$P_{42}(cont\ prob)$			
$P_{13}(cont\ prob)$	P_{23}	$P_{33}(spec\ req)$	P_{43}	80%	100%	80%
P_{14}	P_{24}	P_{34}	P_{44}			
P_{15}	P_{25}	P_{35}	P_{45}			

cases. The experimentation of the resource provisioning treatment is exhibited in [19]. This experimentation uses bookinfo application [5] demonstrating that, using DOCKER-C2A, we can perform the same workload used in Kubernetes example with a smaller number of pods and a better application performance. In this section, we raise the specific request problem using a use case based on an IoT platform. Experimental results demonstrate that using DOCKER-C2A in EPMA autoscaler greatly optimizes computing resources.

Moreover, EPMA autoscaler takes in average 70 milliseconds to generate the decision of scaling, which is a short period of time that does not disturb the elastic treatment. However, EPMA uses a high amount of resources of about 1.2 cores of CPU to process the PSO algorithm. The high resource usage stands for the cost of problem optimization and corresponds to one of the weak points of our autoscaler.

5.1 Experimentation Settings

Part of experimentations of the specific request plan are reported in our paper DOCKERANALYZER [18]. These experiments are conducted on an IoT platform as a real use case to validate our EPMA autoscaler in case of specific request problem.

To evaluate our solution, we compare EPMA behavior to the autoscaler HPA of Kubernetes. Our experiments rest on the specific request problem and are conducted on the IoT platform. In this platform, specific requests are generated by the microservice IoT System Controller with class B requests. Specific request generates the pod outlier. Our autoscaler EPMA should detect the pod outlier by its analyzer DOCKERANALYZER and then should launch adequate action plan using DOCKER-C2A. We launch the same pod outlier in Kubernetes to check the behavior of its autoscaler HPA.

At time t=10, we inject the specific request B to weigh the behavior of each autoscaler. We have clearly explained the behavior of Kubernetes autoscaler in Section 3.3.1. As we notice in Fig. 8, the average CPU usage of the deployment increases considerably and exceeds 80% of CPU demanded. HPA autoscaler adds multiple duplicated pods and reaches the max number of pods so as to decrease CPU consumption.

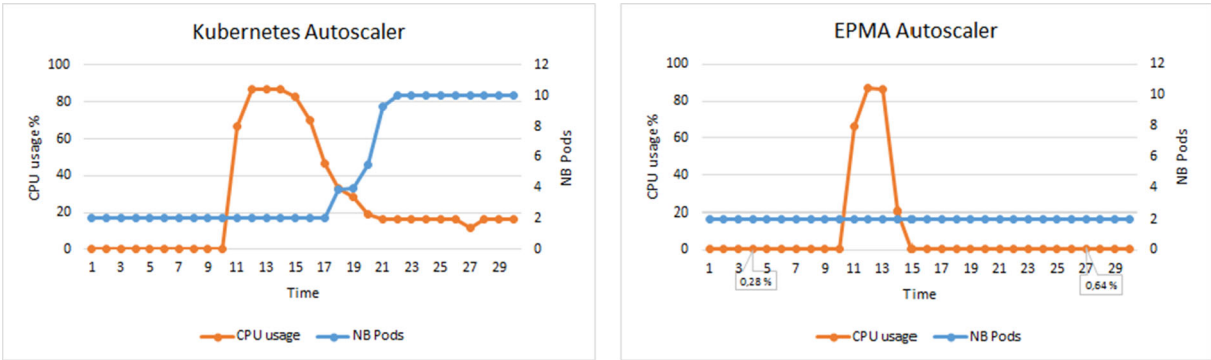
In the following, we address the behavior of EPMA when injecting the specific request.

5.2 EPMA Autoscaler Behavior

Unlike Kubernetes autoscaler, EPMA detects and identifies the root cause of the overload state using DOCKERANALYZER and launches the appropriate action plan using DOCKER-C2A planner. EPMA behavior is outlined in Fig. 8.

After injecting the specific request class B, CPU consumption increases and EPMA launches DOCKERANALYZER. OUTLIERDETECTOR is launched to detect pods outliers as pods having an abnormal behavior. The pod executing the specific request will be detected as an outlier. In this case, the next component ANOMALYIDENTIFIER is triggered to identify the root cause of the outlier behavior. A request outlier is thus detected and the root cause is a specific request problem. From this perspective, the planner will be invoked to solve the problem.

In the first step, DOCKER-C2A allocates additional resources to the pod with vertical scaling treatment. If the pod consumes an excessive amount of resources, which may lead to the overload of the VM, DOCKER-C2A launches the isolation action plan detailed in Section 3.3.2.



(a) Kubernetes Autoscaler

(b) EPMA Autoscaler

Fig. 8 Kubernetes vs EPMA

As inferred from Fig. 8, when the average CPU usage of the deployment increases as a result to the injection of request B, our EPMA autoscaler launches the action plan proposed by DOCKER-C2A. Hence, the average CPU usage of the deployment decreases after the peak owing to pods isolation, and the blue curve representing the number of pods does not increase. It is obvious that we managed to resolve the problem of resource saturation caused by the specific request without adding new instances, unlike the autoscaler of Kubernetes which adds 10 instances of pods to decrease the CPU usage. Thus, we can clearly confirm the effective result of our EPMA autoscaler compared to Kubernetes.

EPMA is a resource-efficient autoscaler, as it significantly reduces computing resources. Additionally, EPMA helps improve the performance of the application compared to Kubernetes. The next section accounts for this improvement.

5.2.1 Impact on Other Requests: Kubernetes vs EPMA

In this section, we describe the impact of specific requests ReqB on usual requests ReqA executed on the same pod. ReqA requests consume a small amount of resources.

The pod executing the specific request ReqB will be overloaded owing to high use of resources. Then, all other requests including usual requests which are ReqA requests will be disturbed while they cannot use required resources. The execution of ReqB increases the response time of ReqA. This increase is expected while the pod is overloaded and ReqA cannot continue the execution normally.

In case of EPMA autoscaler, we report better results. EPMA isolates the pod executing the specific request ReqB until ReqB finishes. Therefore, this pod will not receive other requests which are ReqA requests; the latter will be orchestrated to other pods. Hence, ReqA will use the requested resources and will be executed in a normal way. As a matter of fact, it will be faster and the response time will decrease. Thus, our EPMA autoscaler helps enhance the performance of the application by improving the response time, while with Kubernetes autoscaler, the response time has been affected and significantly increased. Our paper [18] handles in further details the impact of specific requests on usual requests.

Experimental results demonstrate that using DOCKER-C2A in EPMA, greatly optimizes computing resources.

6 Conclusion

This paper presents EPMA autoscaler, an Elastic Platform for Microservice-based Applications. EPMA proves to be an efficient elastic solution that offers a cross layer analysis module that detects and identifies the root cause of the overload state at the application, container and VM levels. Moreover, EPMA provides an efficient planning module that grants different action plans according to the detected problem. As a matter of fact, EPMA avoids unnecessary resource provisioning in case of specific request issue and enables the optimization of resource allocation while allocating just needed resources for each microservice. The main objective of this autoscaler is resource optimization.

At this stage of analysis, we would assert that our research is a step that may be taken further, extended and built upon. Indeed, in future works, we plan to initiate possible improvements and extensions of the EPMA autoscaler while addressing two main parts: analysis and planning modules.

Regarding the analysis module, evaluation results demonstrated that our analyzer accurately identifies the majority of problems but in certain cases, issues are not identified correctly. In fact, we plan to improve the anomaly identification using machine learning.

With respect to the planning module, we emphasize that it displays certain limitations although it optimizes computing resources. In fact, the resource provisioning process uses PSO algorithm which requires high computational resources to generate the appropriate action plan. A possible solution may rely on improving the execution history. From this perspective, we plan to model each arriving workload and identify the required resources. This fact greatly helps optimize computing resources.

Data Availability The data sets generated during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

1. Docker. https://www.docker.com/
2. Elasticsaech: https://www.elastic.co/fr/
3. Kubernetes: https://kubernetes.io/ (2019)
4. Kubernetes autoscaler: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (2019)
5. Kubernetes autoscaler: https://istio.io/latest/docs/examples/bookinfo/ (2020)
6. Openstack cloud. https://www.openstack.org/ (2010)
7. The osgi alliance. osgi service platform. https://www.osgi.org/developer/specifications/
8. Outlier detection. https://support.infogix.com/hc/en-us/community/posts/360028941133-Outlier-Detection-Using-Modified-Z-Score
9. Scalr. https://www.scalr.com/
10. Sysdig monitor. https://sysdig.com/products/monitor/
11. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Autonomic vertical elasticity of docker containers with elasticdocker. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 472–479 (2017)
12. Ashraf, A., Byholm, B., Porres, I.: Cramp: Cost-efficient resource allocation for multiple web applications with proactive scaling. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, pp. 581–586 (2012)
13. Barna, C., Ghanbari, H., Litoiu, M., Shtern, M.: Hogna: A platform for self-adaptive applications in cloud environments. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 83–87 (2015)
14. Calheiros, R.N., Vecchiola, C., Karunamoorthy, D., Buyya, R.: The aneka platform and QoS-driven resource provisioning for elastic applications on hybrid clouds. Futur. Gener. Comput. Syst. **28**(6), 861–870 (2012)
15. Copil, G., Moldovan, D., Truong, H., Dustdar, S.: Sybl: An extensible language for controlling elasticity in cloud applications. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, Grid Computing, pp. 112–119 (2013)
16. da Silva Dias, A., Nakamura, L.H.V., Estrella, J.C., Santana, R.H.C., Santana, M.J.: Providing iaas resources automatically through prediction and monitoring approaches. In: 2014 IEEE Symposium on Computers and Communications (ISCC), pp. 1–7 (2014)
17. Fernandez, H., Pierre, G., Kielmann, T.: Autoscaling web applications in heterogeneous cloud infrastructures. In: 2014 IEEE International Conference on Cloud Engineering, pp 195–204 (2014)
18. Fourati, M.H., Marzouk, S., Drira, K., Jmaiel, M.: DOCKERANALYZER : Towards fine grained resource elasticity for microservices-based applications deployed with docker. IEEE (2019)
19. Fourati, M.H., Marzouk, S., Jmaiel, M., Guerout, T.: Docker-C2A: Cost-aware autoscaler of docker containers for microservices-based applications. Adv. Sci. Technol. Eng. Syst. J. **5**(6), 972–980 (2020)
20. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Exploring alternative approaches to implement an elasticity policy. In: 2011 IEEE 4th International Conference on Cloud Computing, pp. 716–723 (2011)
21. Giannakopoulos, I., Papailiou, N., Mantas, C., Konstantinou, I., Tsoumakos, D., Koziris, N.: Celar: Automated application elasticity platform. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 23–25 (2014)
22. Guerrero, C., Lera, I., Juiz, C.: Genetic algorithm for multiobjective optimization of container allocation in cloud architecture. J. Grid Comput. **16**(1), 113–135 (2017)
23. Hadley, J., Elkhatib, Y., Blair, G., Roedig, U.: MultiBox: Lightweight containers for vendor-independent multi-cloud deployments. In: Embracing Global Computing in Emerging Economies, pp. 79–90. Springer International Publishing (2015)
24. Han, R., Ghanem, M.M., Guo, L., Guo, Y., Osmond, M.: Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Futur. Gener. Comput. Syst. **32**, 82–98 (2014)
25. Herbst, N.R., Kounev, S., Reussner, R.: Elasticity in cloud computing: What it is,what it is not. In: 10th International Conference on Autonomic Computing (ICAC 13), pp. 23–27. USENIX Association, San Jose (2013)

26. Hoenisch, P., Weber, I., Schulte, S., Zhu, L., Fekete, A.: Four-fold auto-scaling on a contemporary deployment platform using docker containers. In: Service-Oriented Computing, pp. 316–323. Springer Berlin Heidelberg (2015)
27. IBM: An architectural blueprint for autonomic computing (2005)
28. Iglewicz, H.: modified-z-scores. https://www.itl.nist.gov/div898/handbook/eda/section3/eda35h.htm
29. Iqbal, W., Dailey, M.N., Carrera, D., Janecek, P.: Adaptive resource provisioning for read intensive multi-tier applications in the cloud. Futur. Gener. Comput. Syst. **27**(6), 871–879 (2011)
30. Kachele, S., Hauck, F.J.: Component-based scalability for cloud applications. In: Proceedings of the 3rd International Workshop on Cloud Data and Platforms - CloudDP 13. ACM Press (2013)
31. Kan, C.: Docloud: An elastic cloud platform for web applications based on docker. In: 2016 18th International Conference on Advanced Communication Technology (ICACT), pp. 1–1 (2016)
32. Kan, C.: Docloud: An elastic cloud platform for web applications based on docker. In: 2016 18th International Conference on Advanced Communication Technology (ICACT), pp. 1–1 (2016)
33. Kaur, P.D., Chana, I.: A resource elasticity framework for QoS-aware execution of cloud applications. Futur. Gener. Comput. Syst. **37**, 14–25 (2014)
34. Lewis, J., Fowler, M.: Microservices https://martinfowler.com/articles/microservices.html (2014)
35. Liu, B., Buyya, R., Toosi, A.N.: A fuzzy-based auto-scaler for web applications in cloud computing environments. In: Service-Oriented Computing, pp. 797–811. Springer International Publishing (2018)
36. Paraiso, F., Challita, S., Al-Dhuraibi, Y., Merle, P.: Model-driven management of docker containers. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp. 718–725 (2016)
37. Petcu, D., Sandru, C.: Towards component-based software engineering of cloud applications. In: Proceedings of the WICSA ECSA 2012 Companion Volume on - WICSA ECSA 12. ACM Press (2012)
38. Pokahr, A., Braubach, L.: Towards elastic component-based cloud applications. In: Intelligent Distributed Computing VIII, pp. 161–171. Springer International Publishing (2015)
39. Powers, DM.W.: Evaluation: From precision, recall and f-measure to roc, informedness markedness and correlation (2020)
40. Kukade, GKPP.: Auto-scaling of micro-services using containerization. In: International Journal of Science and Research (IJSR), vol. 4 (2015)
41. Sauvanaud, C., Kaâniche, M., Kanoun, K., Lazri, K., Da Silva Silvestre, G.: Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. J. Syst. Softw. **139**, 84–106 (2018)
42. Subraya, B.M.: Integrated Approach to Web Performance Testing IGI Global (2006)
43. Yu, G., Chen, P., Zheng, Z.: Microscaler: Automatic scaling for microservices with an online learning approach. In: 2019 IEEE International Conference on Web Services (ICWS). IEEE (2019)
44. Zhang, F., Tang, X., Li, X., Khan, S.U., Li, Z.: Quantifying cloud elasticity with container-based autoscaling. Fut Gen Comput Syst **98**, 672–681 (2019)