Advances in Engineering Software 140 (2020) 102734

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Research paper

Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications

Salman Taherizadeh[a,b,*], Marko Grobelnik[a]

[a] Artificial Intelligence Laboratory, Jozef Stefan Institute, Jamova cesta 39, Ljubljana 1000, Slovenia
[b] CVS Mobile, Slovenia

ARTICLE INFO

Keywords:
Auto-scaling
Key factors
Microservices
Kubernetes
Computing-intensive services
Cloud-based applications

ABSTRACT

Nowadays, different types of computing-intensive services such as mechanical, aerospace, civil and environmental applications are often deployed on the cloud since it offers a convenient on-demand model for renting resources and easy-to-use elastic infrastructures. Moreover, the modern software engineering disciplines exploit orchestration tools such as Kubernetes to run cloud applications based on a set of microservices packaged in containers. On the one hand, in order to ensure the users' experience, it is necessary to allocate enough number of container instances before the workload intensity surges at runtime. On the other hand, renting expensive cloud-based resources can be unaffordable over a long period of time. Therefore, the choice of a reactive auto-scaling method may significantly affect both response time and resource utilisation. This paper presents a set of key factors which should be considered in the development of auto-scaling methods. Through a set of experiments, a discussion follows to help shed light on how such factors influence the performance of auto-scaling methods under different workload conditions such as on-and-off, predictable and unpredictable bursting workload patterns. Due to suitable results, the proposed set of key factors are exploited in the PrEstoCloud software system for microservice-native cloud-based computationally-intensive applications.

1. Introduction

Cloud computing [1] is a preferable technology to increase computational capacity and strengthen application performance dynamically. This includes various types of advanced engineering computing applications such as scientific-numerical computations. As an example, on-demand weather forecasting is necessary to plan ahead in response to current events such as floods, storms, tornado, typhoons and lightning strikes killing thousands of people and causing extensive damages every year. Such large-scale weather forecasts consume excessive computational resources and ingest huge amounts of data in real-time, no longer possible with traditional forecasting technologies which are unable to address changing workloads at runtime. The latest paradigm shift in the modern software development of these computing-intensive applications is microservices [2].

Microservices are considered as a new software architecture for building highly modular applications deployed on the cloud. An application developed based on the microservices architecture is composed of different smaller services that each one can be deployed independently [3]. Microservices are highly decoupled services, and hence failure of one of them will not bring other microservices of the system down. When it comes to the conditions of dynamically varying workloads, this feature supports a modern software engineering practice able to offer a higher level of scalability compared to classical architectures, where the functionality of an application is packaged into only one large non-separable component. This is because individual microservices can be independently scaled separately in response to their current demand in real-time as a consequence of this new highly modular software engineering approach.

The change in the workload demands of cloud-based applications may happen in different ways. For example, an on-demand weather forecasting system unexpectedly receives a heavy workload to be processed in response to a sudden occurrence of new atmospheric events altogether at the same time such as storm formation, volcanic eruptions, rainfall and tornadoes. Another example is a cloud-based batch processing system for which requests tend to be accumulated around batch runs regularly over only short periods of time. For instance, some weather forecasting services are employed to periodically estimate the power output of wind turbine or photovoltaic systems for short time intervals. These types of services generally have short active periods, between which the application can be provided at the lowest service level.

* Corresponding author.
E-mail addresses: Salman.Taherizadeh@ijs.si (S. Taherizadeh), Marko.Grobelnik@ijs.si (M. Grobelnik).

https://doi.org/10.1016/j.advengsoft.2019.102734
Received 9 May 2019; Received in revised form 18 September 2019; Accepted 21 October 2019
Available online 08 November 2019
0965-9978/

Microservices are mainly packaged into containers as lightweight virtualisation in comparison to Virtual Machines (VMs). This is because they do not need to start or stop operating system that can take significant amount of time. Due to their lightweight nature, they can be instantiated, terminated and managed very dynamically. Exploiting such lightweight container-based virtualisation can make better auto-scaling improvements on both application response time as well as resource utilisation aspects quicker and more efficiently than employing VMs [4].

Various open-source container management platforms such as Kubernetes are currently provided in order to deploy scalable microservices. Nowadays, almost all engineering positions from software engineers to site reliability engineers one way or another deal with Kubernetes. Software engineers need to have vast experience in building microservice-based software systems orchestrated by Kubernetes, which is the most widely used management system for containers along with a massive community behind it. They decide how the system is built based on microservices and all technicalities derived from this design. Site reliability engineers develop automated solutions for operational aspects such as performance and capacity planning as well as response to Quality of service (QoS) degradation at runtime. Container management platforms such as Kubernetes provide reactive auto-scaling methods based on a set of static rules in order to operate under changing workloads over time [5]. An ordinary practice is mainly to employ fixed, infrastructure-level CPU-based auto-scaling rules to scale up or scale down the number of container instances allocated to a specific service depending on demand. Although these existing reactive auto-scaling methods with fixed rules may be appropriate for some basic scenarios of cloud-based applications, they may result in an undesirable QoS or poor resource utilisation in environments with certain dynamic workload scenarios. This is an important research area because auto-scaling methods need to continuously (i) ensure application QoS with respect to the response time as well as (ii) assign the optimal amount of resources in terms of the number of container instances. Therefore, ensuring microservice-based applications governed by existing reactive auto-scaling rules specifically offered by Kubernetes to offer favourable performance is currently a challenging issue.

More objectively, the primary goal of the present paper can be shortened as follows: (i) presenting a set of new influencing factors which have not fully received attention so far in the dynamic management of scalable resources provided by the container orchestration platforms such as Kubernetes; (ii) evaluating the choices of such factors to develop the optimum scaling strategy to be used and analysing the way how they dynamically influence the impact of reactive auto-scaling rules; and (iii) demonstrating the way to tune auto-scaling of containerised applications orchestrated by Kubernetes with regard to diverse workload patterns.

In order to recognise the effect of our proposed factors, three different workload scenarios are examined in this work, including (i) predictable bursting workload pattern, (ii) unpredictable bursting workload pattern, and (iii) on-and-off workload pattern. Based on experiments conducted for each workload pattern, the results of our evaluation show that the Kubernetes auto-scaling method is highly sensitive to changes in the value of our proposed factors. In other words, the results imply that there are significant factors which need to be considered in the implementation of auto-scaling methods, while dealing with different workload scenarios.

The rest of the paper is organised as follows. Section 2 presents a background related to the microservices architecture. Section 3 discusses related work on existing reactive auto-scaling methods which have been proposed by academia and industry. Key factors to be considered while developing auto-scaling rules used by provisioning methods for scalable microservices are illustrated in Section 4. Section 5 presents empirical evaluation along with experimental results. Section 6 contains a critical discussion. Finally, conclusion appears in Section 7.

2. Microservices architecture background

Microservices are small, loosely coupled processes capable of communicating together via language-independent Application Programming Interfaces (APIs) to create the whole application in a cloud-native architecture. In other words, an application includes small, self-contained deployable microservices, each one acting as an individual function application, working together through APIs, which are not dependent on a particular language, library, framework and more [6]. In this way, decomposing one application into small microservices enables cloud-based application providers to distribute the computational workload of services among various resources. Besides this, each of the services can be easily developed and operated by different software engineering teams, and hence, this architecture affects both organisational forms of cooperation as well as technological decisions which can be made locally by each team [7]. Compared to Service Oriented Architecture (SOA) [8], microservices are typically designed around business capabilities and priorities, and independent deployability is a key characteristic of them [9]. Microservices usually exploit simple interfaces known as Representational State Transfer (REST).

Resilience to failure is an important feature of microservices because each application request is divided and translated to different service calls in this software architecture. Therefore, a bottleneck in a specific service operation will not bring the entire system down and it affects only that service. In such a situation, other services are able to carry on processing requests as usual. Accordingly, the microservices architecture addresses necessary requirements including distribution, modularity as well as fault-tolerance [10]. Another useful capability of the microservices architecture is improving the reusability of software components. It means that a single microservice can be reused or shared in several applications or even in various parts of the same application.

An important cloud-native application property named scalability is another advantage of microservices in order to exploit computing infrastructures in a feasible manner to build distributed, large-scale and extensively scalable cloud-based applications [11]. A microservice may be composed of multiple runtime instances depending on workload variations over time. This means one microservice consists of one or more runtime instances during execution time to react to workload fluctuations. Scalable cloud-native systems, in which execution environments are constantly dynamic and workloads change over time, comprise such independently replicable instances.

Fig. 1 depicts an example of the microservices architecture in which various services may have different amount of demands at runtime to accomplish their own particular tasks, and thus there is the possibility of scaling each service dynamically at distinct level. As shown in this exemplary figure, the on-demand weather forecasting system receives the sensor data representing the presence of three different tornadoes along with an area of high-pressure cloud going to cause a violent storm. In this case, microservices are required to perform two different functions, including Service A to construct models for tornadoes and Service B to derive a pattern for the existing high-pressure cloud area. Therefore, due to different amounts of workloads for various services, three instances are allocated to Service A since each instance is constructing a distinct model for the associated tornado, whereas Service B consists of only one instance deriving a pattern for the area of high-pressure cloud. This fact implies enormous demands are put on Service A and a lower level of Service B.

In essence, the REST API Gateway acts as a proxy to microservices. It is a single-entry point into the whole system. Furthermore, the REST API Gateway can support other functional capabilities for example to come up with caching, security and monitoring operations at one place.

The logical continuation of the microservices architecture for cloud-native applications has been discussed comprehensively in a research work by Kratzke and Quint [11] in various cloud contexts. The authors explained that the term microservice is deeply aligned to cloud-native

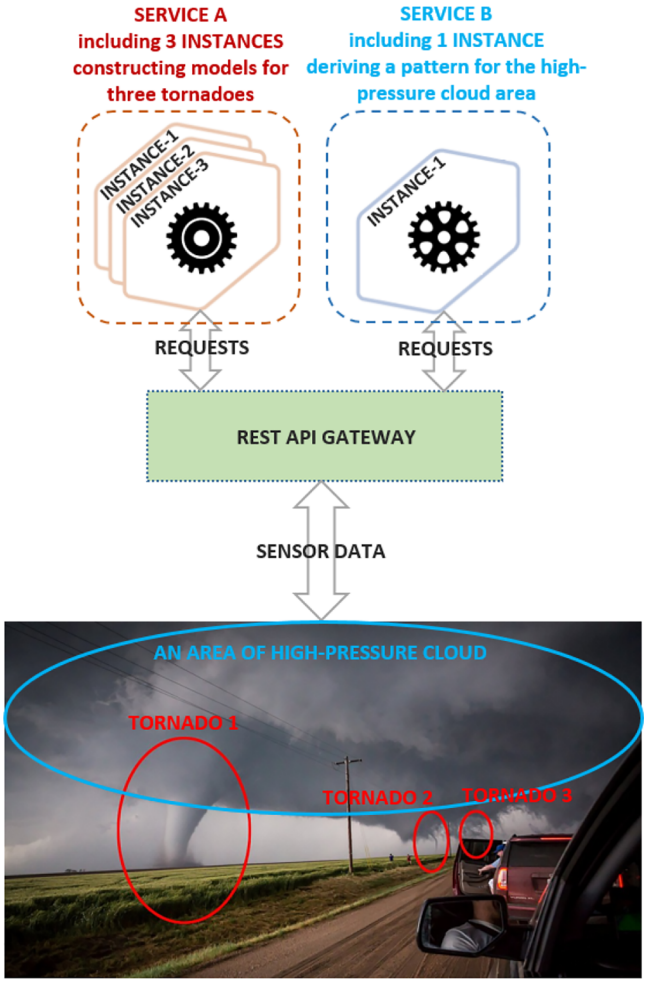

Fig. 1. An example of the microservices architecture used for an on-demand weather forecasting system.

architectures, regardless of being micro or macro but the right size. Microservices generally are packaged using container-based virtualisation and deployed in the cloud. The main interest in this approach is for the sake of having the opportunity to build self-contained deployment units so-called containers in a standardised form. Furthermore, containers are highly lightweight, and hence they can be exploited to deploy microservices quickly. However, microservices could be deployed completely without containers, for example through VMs. Nevertheless, it should be noted that resource utilisation of VMs is relatively high, and consequently in this way they are not considered as the main option to be deployed on resource-constrained edge computing devices such as Raspberry-Pi, BeagleBoard and pcDuino3-Nano.

3. Related work

Reactive auto-scaling rules are considered as purely static, threshold-based adaptation policies. In this way, adaptation actions are commenced according to the value of some parameters based upon a set of predefined thresholds. The most important advantage of such rule-based auto-scaling approaches is their simplicity since these rules are easy-to-set-up for the scalability of underlying cloud-based infrastructures. In this section, existing reactive methods in auto-scaling of cloud-based applications are explained in detail in order to include their advantages and limitations. These approaches are chosen for comparison together because they are mainly rule-based and considered as advanced reactive auto-scaling solutions.

Al-Sharif et al. [12] proposed a framework named Autonomic Cloud Computing Resource Scaling (ACCRS) in order to provision a sufficient number of Virtual Machines (VMs) to address the changing resource requirements of an application running on the cloud. The proposed adaptation method employs a set of fixed thresholds for resource-level metrics, for example CPU utilisation. In this approach, the workload can be identified as a light or heavy weight if the resource usage violates the associated threshold. The presented resource scaling framework uses hypervisor-based virtualisation technologies, which are able to support only VMs. A VM is a service instance, which has its own operating system as well as a set of libraries, within an emulated environment offered by the hypervisor. Different from VMs, the utilisation of containers does not need an operating system to boot up that has gained increasing popularity in the cloud computing frameworks. Resource usage of VMs is extensive and thus typically they cannot be easily developed on small servers or resource-constrained devices.

Kukade and Kale [13] demonstrated a master-slave auto-scaling architecture for containerised applications. Slaves represent the nodes where containers can be deployed, while there is a master that is responsible for receiving arrival requests and routing them to running container instances. The master also includes a self-adapter module that is able to check two different scaling rules in order to increase or decrease the number of running containers. Firstly, if the request rate exceeds a pre-defined fixed threshold, a new container instance will be started. If the memory load of containers reaches a threshold, then a new replica of container instance will be launched. However, CPU utilisation which is the most important metric for computing-intensive applications in real-world auto-scaling platforms has not been considered by the study.

Kan [14] introduced a container-based elastic cloud platform called DoCloud. This platform incorporates proactive and reactive models to calculate the number of containers to be added for the scale-out, while the proactive model is applied for the scale-in to remove unnecessary containers from the service cluster. DoCloud exploits static thresholds for CPU and memory utilisation, and uses the Auto-Regressive Moving Average (ARMA) method to predict the number of incoming requests for the application.

Baresi et al. [15] proposed an auto-scaling approach that employs an adaptive discrete-time feedback controller which enables a container-based application to dynamically scale necessary resources. In this work, a component called ECoWare agent needs to be deployed in each VM. An ECoWare agent is in charge of the collection of container-specific monitoring data, such as containers' usage of CPU usage, etc. This component is also responsible for instantiating or terminating a container in the VM, or changing the resources assigned to a container. This auto-scaling method is limited only to Web applications. In addition, it adds overhead by deploying ECoWare agents for each container and VM.

A static auto-scaling method which is called THRESHOLD or THRES (Metric, UP%, DOWN%) [16] can horizontally add a container instance if an aggregated metric (such as the average CPU utilisation of the cluster) reaches the predetermined UP% threshold. Moreover, it is able to remove a container instance when it falls below the predefined DOWN% threshold for a default number of consecutive intervals, e.g. two intervals. The approach named "THRES1 (CPU, 80%, 50%)" is an example for such a static auto-scaling method. This auto-scaling method is not flexible enough in order to adjust itself to dynamic changes of the operational environment, and it leads to too much resource waste over time.

Tsoumakos et al. [17] presented a resource provisioning approach which is called TIRAMOLA in order to identify the number of VM instances necessary to meet user-defined objectives for a NoSQL database cluster. This resource provisioning approach continuously decides the most beneficial state that can be achieved at runtime, and thus specifies possible actions in each state that can either do nothing, or add/remove NoSQL nodes. The principle of TIRAMOLA is acting in an expected style of operation when the regular workload scenario can be specified.

Accordingly, previously unobserved workloads are considered as the major obstacle to the fast adaptation of the entire system to meet the performance objective of cloud-based interactive services. Furthermore, TIRAMOLA is restricted to the elasticity of a specific type of application such as NoSQL databases. Moreover, the monitoring part needs to collect client-side statistics in addition to server-side metrics. In this regard, clients of such applications should be modified so that each one can report its own monitoring statistics, which is not an operational solution for many real-world use cases.

Kubernetes is a widely used lightweight open-source container management platform which is capable of orchestrating containers and automatically providing horizontal scalability of applications. In Kubernetes, a group of one, or a small number of containers which are tightly coupled together with a shared IP address and port space can be defined as a pod. Therefore, a pod simply indicates one single instance of an application which can be replicated, if more instances are helpful to handle the increasing workload. The Kubernetes auto-scaling approach [18] is a control loop algorithm basically based upon CPU utilisation. The Kubernetes auto-scaling algorithm which is presented in Algorithm 1 can increase or decrease the number of containers to keep the average CPU utilisation at, or close to, a target value such as 80%.

Inputs:
$Target_{cpu}$: Targeted CPU resource usage, e.g. 80%
CLTP: Control Loop Time Period in seconds, e.g. 30 seconds
Outputs:
P#: Number of pods to be running

```
do{
    Cluster = [Pod1,..., PodN];
    SumCpu = SUM (cpu_usage_of_pod1,..., cpu_usage_of_podN);
    P# = ⌈SumCPU / Target_cpu⌉;
    Execute (P#);
    Wait (CLTP);
} while (true);
```

In the Kubernetes auto-scaling algorithm, the SUM function employed for calculating the total sum of the CPU utilisation of the cluster. The auto-scaling period of the Kubernetes auto-scaler is half a minute (30 s) by default that can be changed. At each auto-scaling iteration, Kubernetes' controller may add or remove a number of containers according to P# (number of pods to be running).

It should be mentioned that the auto-scaler of container orchestration tools such as Kubernetes follows a broadly accepted reference model named MAPE-K (Monitor, Analyse, Plan and Execute over a shared Knowledge Base) used in various autonomic computing systems [19]. In essence, the Kubernetes auto-scaler is considered as a classical MAPE-K loop feedback instance proposed to offer as a guideline to build self-adaptive software systems. The Monitor phase describing the execution environment generates the input data for the Analyse phase, which aims at decision-making on if any adaptation is required in given conditions. The Plan phase provides appropriate actions to adapt the target system with regard to feasible adaptation strategies such as scaling up or down. The Execute phase which receives the change plan involves the adaptation operation. The Knowledge Base is also employed to store all information about the whole execution environment.

4. Proposed key influencing factors

This section explains three significant factors, which may influence the efficiency of container management platforms such as Kubernetes to provide reactive auto-scaling methods based on a set of static rules. These influencing factors, which have not been fully met by any of the existing cloud-based auto-scaling technologies, include (i) conservative constant called α, (ii) adaptation interval called CLTP, and (iii) stopping at most one container instance in each CLTP. These factors are derived from the literature analysis described in details for each one in the next subsections.

4.1. Conservative constant (α)

In some experience studies [20–22], there are certain situations, where it can be observed that cloud-based applications enter to unstable state with a little fluctuation even under predictable bursting workload scenarios, while considering the dynamics of the underlying computing environments. One particular reason for this is that frequent auto-scaling actions for a given workload may cause too much auto-scaling overhead, which will render the system unstable. In some cases, researchers suggested to use larger time intervals to update the number of containers so-called cluster size. However, large adaptation periods may result in missing dynamics in the execution environment, and hence the system may not be accurate and agile enough to track timely all changes in the workload. This is a significant issue, which may cause a huge software failure for computing-intensive microservice-native cloud-based applications. In order to understand how this challenge can be addressed in this subsection, we focus on how the auto-scaling of containerised applications can be precisely tuned under the condition of predictable bursting workloads.

An ordinary auto-scaling practice exploited by existing cloud resources is to use fixed, infrastructure-level rules. For instance, a CPU-based auto-scaling policy can be specified in a way that more container instances should be instantiated if the average CPU utilisation reaches a fixed threshold such as 80%; while some container instances may be stopped if the average CPU utilisation is below 80%. As a consequence, the average CPU usage, which varies close to the specified threshold, will lead to frequent unnecessary container instantiations or terminations. For example, let us assume that there are two container instances in the cluster. CPU usage of the first container is 40%, and CPU usage of the second container is 39%. In this case, the cluster size will be changed to one instance according to the Kubernetes auto-scaling algorithm presented in Algorithm 1. This is because $\lceil (40\% + 39\%) \div 80\% \rceil = \lceil 0.9875 \rceil = 1$ that means one container instance should be eliminated from the cluster because of the downscaling Kubernetes principle. Experimenting with the same workload density after the container termination, the CPU usage of the cluster, which now includes only one container, would be almost 80%, and with a minor variation possibly it would be 81%. Hence, the auto-scaling system adds one container instance due to the upscaling Kubernetes principle. This is because $\lceil (81\%) \div 80\% \rceil = \lceil 1.0125 \rceil = 2$ that means the cluster size will change again to two container instances. This is an example that minor fluctuations in the workload density may lead to frequent unnecessary changes in the cluster size.

In order to achieve a stable operational environment, we propose a

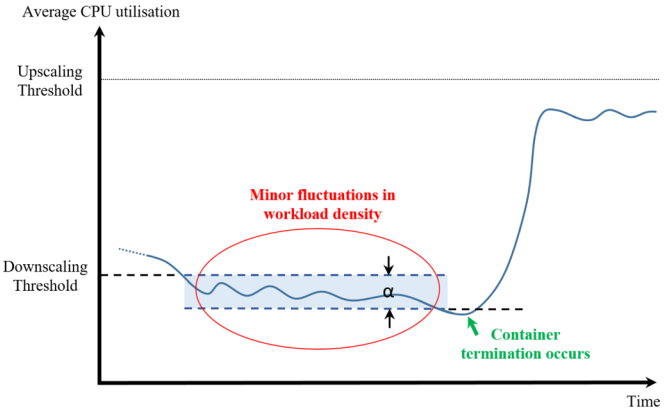

Fig. 2. Avoiding frequent changes in the cluster size due to minor fluctuations in the workload density.

conservative constant called α, which should be used by the auto-scaling system. The constant α is aimed at avoiding frequent changes in the cluster size due to minor fluctuations in the workload. Fig. 2 shows how this constant tries to sustain the number of running containers unchanged when fluctuations in the number of requests are not severe at runtime. During the time period highlighted in blue, the constant α provides the expected robustness of auto-scaling method while there exists a trembling workload which does not vary drastically. Afterwards, when the workload density drops more, a running container instance is terminated to improve the resource utilisation without any application performance degradation.

The constant α should have a reasonable value, neither too large to cause over-provisioning problem nor too small to lead to serious performance drops. A big value for α may reduce the efficiency of the auto-scaling method because, in such a case, redundant container instances have less possibility of being terminated from the cluster in general. Therefore, a higher value of α would possibly cause longer periods of overprovisioned resources. Experimenting with equal computational requirements and workload density within operational environments, an up to almost 10% variation in the average CPU utilisation can be always seen. This difference is the effect of runtime variations in running conditions that are out of the application providers' control, for example time-varying processing delays, CPU or I/O load factors, etc. With respect to this rationale, the maximum value for α can be set to the value of 10%. Therefore, the constant α can have a value between 0% and 10%, that helps the auto-scaling method conservatively make sure that the termination or instantiation of container instances will not result in an unstable situation. Along this line, the Kubernetes auto-scaling algorithm with respect to the conservative constant (α) needs to be updated as presented in Algorithm 2.

Inputs:
$Target_{cpu}$: Targeted CPU resource usage, e.g. 80%
CLTP: Control Loop Time Period in seconds
α: Conservative constant
Outputs:
P#: Number of pods to be running

```
do{
    Cluster = [Pod1,..., PodN];
    Cluster_size = N;
    SumCpu = SUM (cpu_usage_of_pod1,..., cpu_usage_of_podN);
    If (SumCPU > (N * Target_cpu))
        P# = ⌈SumCPU / (Target_cpu + α)⌉;
    else
        P# = ⌈SumCPU / (Target_cpu − α)⌉;
    end if
    Execute (P#);
    Wait (CLTP);
} while(true);
```

4.2. Adaptation interval called control loop time period (CLTP)

The adaptation interval called "Control Loop Time Period (CLTP)" is the minimum duration between two successive adaptation actions over time. The adaptation interval should be defined longer than the time period taken to start up a container instance in the execution environment. This is because the auto-scaling method needs to make adaptation decisions when the system is quite stable. In such manner, if any auto-scaling action happens, the whole system can continue operating properly without losing control over container instances running in the cluster. The default adaptation interval adopted by the Kubernetes platform is 30 s, which also can be changed. At each iteration, Kubernetes' auto-scaling controller may increase or decrease the number of container instances in the cluster.

Some remarkable studies [23–26] have analysed auto-scaling methods for container-based applications recently. Such research works present a wide range of parameters to be considered by their proposed container-based auto-scaling platforms in practice, including the threshold of resource utilisation, the present utilisation of resources, the current number of containers in the cluster, etc. In order to make such proposed auto-scaling solutions capable of operating also in highly dynamic environments under unpredictable bursting workload scenarios, one step further could be taken into account by considering various adaptation intervals rather than a fixed period. Besides, the Kubernetes auto-scalers, which has been applied to enhance the performance of containerised applications in cloud computing studies [27–30], employ only the default CLTP. However, the length of adaptation interval is required to be evaluated not only to rapidly scale the system capacity at any size, but also to avoid losing control over the execution environment due to unpredictable variations in the workload density at runtime. Along this line, the length of the adaptation interval, whether CLTP = 15 s, CLTP = 30 s or CLTP = 60 s employed by the auto-scaling approach may influence the overall performance of the application under the condition of unpredictable bursting workloads in the execution environment.

4.3. Stopping at most one container instance in each CLTP

Batch processing systems are computing applications that process on-and-off workloads in which requests are accumulated around batch runs usually over short time periods. On-and-off workloads typically represent such applications which are repeatedly required for a while and later shut off for a short period of time.

Batch processing systems have recently received substantial attention as an important cloud computing research area due to their widely used nature in modern engineering software such as business analytics, stock control, payroll processing, etc. Relevant papers that have been published in this area introduced cloud-based computing solutions able to receive a batch of requests over short periods of time. Tamrakar et al. [31] presented two different data processing algorithms to scale services in the cloud. The first algorithm deploys new service instances on-demand only in case of QoS degradation caused by termination of previous instances. Hence, applying this algorithm may become impractical due to the time constraints imposed by some time-critical applications such as disaster early warning systems. The second algorithm uses a higher number of service instances than required to accomplish the work that apparently results in unnecessary over-provisioning of resources. Zhang et al. [32] proposed a cloud-based video batch processing platform named Video Cloud integrated with Batch processing and Fast processing (ViCiBaF). Their idea is to allow the users to share computing resources in order to extend the infrastructure at runtime. This means that the scalability of cloud-based batch processing applications can be provided through adding computing devices dynamically by end-users that makes the system usually less predictable and less reliable. This is because such resources may be withdrawn from the running environment at any time.

Moreover, proactive auto-scaling methods [33,34] for batch processing systems have been developed in 2019 to predict the amount of resources required in the near future based on collected historical monitoring data, current intensity of workload, etc. Such proactive auto-scaling approaches generally use learning algorithms such as reinforcement learning, neural network to scale up or scale down the cloud resources. It should be noted that these methods require enough historical data to train a performance model and some time to converge towards a stable driven model. Therefore, if proactive auto-scaling methods have a large enough training data set reflecting characteristics of all different possible operational situations, they are capable of generalising that means they can react to unseen changing workload scenarios. As a consequence, if the training data set is not comprehensive enough, such proactive approaches may suffer from their

imprecision limit which may result in whether over-provisioning problem or serious performance drops.

The auto-scaling method for batch processing systems should ensure the QoS of the application, while stopping or starting container instances in each adaptation interval. In other words, after the termination or instantiation of container instances, the auto-scaling method has to provide acceptable service responses within continuously uncertain environments under the condition of on-and-off workloads at runtime. The point is that the Kubernetes auto-scaling algorithm may fail to provide the expected application performance during on-and-off workload scenarios. In such workload patterns, stopping most of the container instances running in the cluster at once when the number of requests instantly drops down a lot is not a suitable adaptation action. This is because more container instances running into the pool of resources will be required very soon. In this case, terminating container instances for the inactive periods may cause too many changes in the cluster size with the consequent QoS degradation.

To come up with a solution, the auto-scaling method may terminate at most one container instance in each adaptation interval. This is an auto-scaling strategy which can be adopted to handle on-and-off workload patterns in which peak spikes appear periodically in short time intervals. While having on-and-off workload scenarios, the Kubernetes auto-scaling algorithm with respect to stopping at most one container instance should be updated as included in Algorithm 3.

Inputs:
$Target_{cpu}$: Targeted CPU resource usage, e.g. 80%
CLTP: Control Loop Time Period in seconds
Outputs:
P#: Number of pods to be running

```
do{
    Cluster = [Pod1,..., PodN];
    Cluster_size = N;
    SumCpu = SUM (cpu_usage_of_pod1,..., cpu_usage_of_podN);
    P# = ⌈SumCPU / Target_cpu⌉;
    If (P# < N)
        P# = N – 1;
    end if
    Execute (P#);
    Wait (CLTP);
} while(true);
```

5. Empirical evaluation

A set of experiments was performed to evaluate the choices of influencing factors (conservative constant (α), adaptation interval (CLTP) and stopping at most one container instance), and accordingly the sensitivity of the Kubernetes auto-scaling method to changes in the value of these parameters was analysed.

Each experiment was repeated for three iterations to achieve the average values of important properties and to verify the obtained results and thus to have a greater validity. Accordingly, the results reported are mean values over three runs for each experiment.

All host machines applied in our experiments belong to the Academic and Research Network of Slovenia (ARNES) which is a non-profit cloud infrastructure provider. In our experiments, all machines allocated to the cluster which provides the service have the same hardware characteristics: CPU cores: 4, CPU MHz: 2397, Memory: 4GB and Speed: 1000 Mbps.

An application was developed and containerised to provide numerical computations widely used within engineering problems as a pilot use case. The application specifically solves sparse systems, which are ubiquitous in various scientific and technical computations. A system of equations can be considered as sparse if only a relatively small number of its multi-dimensional matrix elements are non-zero. The time needed to solve sparse equations basically makes up a large share of the whole numerical computations, for example the prediction of complex systems such as weather forecasting applications. As far as further precision of the prediction result is attainable, the size and the number of such equations become larger, and consequently the amount of computation will significant increased. For example, our developed application is also capable of solving a sparse problem to find a narrow area which minimises distance from different points (e.g. three points in our experiments as an incoming request) with one extra constraint—the area should include an additional, specific point. These jobs can provide a benchmark which is a perfect match for our available computing infrastructures with aforementioned hardware characteristics provided by ARNES. In our use case, a single task request normally takes 130 ms with our used experimental setup in conditions where the system is not overloaded.

The REST API GATEWAY as Load-Balancer was implemented by *HAProxy* [35], which provides high-availability support for cloud-based applications by spreading requests across multiple container instances. HAProxy is widely used by a number of auto-scaling research works [36–46] and also high-profile commercial solutions including GoDaddy [47], GitHub [48], Stack Overflow [49], Reddit [50], Speedtest [51], Bitbucket [52], Twitter [53], W3C [54] as well as the AWS OpsWorks [55] product from Amazon Web Services. The auto-scaler is also able to dynamically determine host machines that are not overloaded at runtime, so that the Load–Balancer would distribute requests to those nodes which have currently more computing capabilities. In other words, the Load–Balancer distributes all incoming requests across the cluster of instances able to perform extra jobs. In our experiments, service instances in the cluster are running on different host machines. This is because there is no point to start additional instances of the same service on the same host since all incoming requests are identical in our experiments.

Moreover, the *httperf* [56] tool has been employed in order to build a load generator which is able to produce different workload patterns for various analyses in our empirical evaluation. To this end, three different workload scenarios have been inspected, including (i) predictable bursting workload scenario, (ii) unpredictable bursting workload scenario and (iii) on-and-off workload scenario. In every experiment, results are analysed to ensure if the auto-scaling method is able to meet the application performance, while optimising the resource allocation. In this context, each auto-scaling approach is investigated primarily according to the average response time and the average number of container instances.

Such as many cloud resource management systems [57–62], the targeted CPU resource usage was set to the value of 80%. Because the unpredictable bursting and on-and-off workload scenarios examined in our experiments are considered neither even nor predictable. Therefore, auto-scaling methods have enough chance to react to runtime fluctuations in the workload since the targeted threshold is not very close to

Table 1
Comparing various auto-scaling methods with respect to different workload scenarios.

Workload scenario	Auto-scaling method	Avg. response time	Avg. number of container instances
Predictable bursting workload	α = 0 (Default K8S)	142.69 ms	1.50 containers
	α = 5	139.59 ms	1.67 containers
	α = 10	139.64 ms	2.00 containers
Unpredictable bursting workload	CLTP = 15	148.42 ms	1.67 containers
	CLTP = 30 (Default K8S)	140.33 ms	1.67 containers
	CLTP = 60	155.87 ms	1.57 containers
On-and-off workload	Default K8S	219.96 ms	2.30 containers
	Stop at most one	190.77 ms	2.60 containers

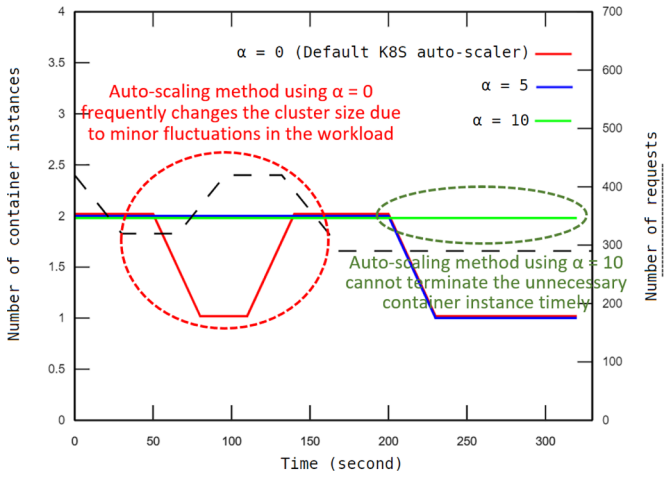

Fig. 3. Number of container instances allocated by three different Kubernetes (K8S) auto-scaling methods using $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$.

100%.

Table 1 presents the average response time as well as the average number of container instances allocated by various auto-scaling methods in every workload pattern examined in this work. The average response time represents the application QoS, while the average number of container instances conveys the concept of the resource utilisation offered by auto-scaling methods. It is also worth noting that having fewer container instances compared to other situations with more instances is preferred. Moreover, faster response time is an important determinant of comparison among various auto-scaling methods because it is transparent to the end-user. Table 1 is described in detail in the following subsections.

5.1. Predictable bursting workload scenario

In an operational environment with predictable bursting workload in which there are minor fluctuations in the number of requests, the conservative constant (α) is used to avoid an unstable operational situation. To demonstrate the practical applicability of this key factor, we performed various experiments with three different alpha: $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$. To this end, a workload scenario has been examined that includes trembling number of requests between 290 and 420 over time. Fig. 3 shows the number of container instances allocated at runtime by three different Kubernetes auto-scaling methods using $\alpha = 0$ (default Kubernetes auto-scaler), $\alpha = 5$ and $\alpha = 10$.

Moreover, Fig. 4 depicts the response time offered by three different auto-scaling methods using $\alpha = 0$ (default Kubernetes auto-scaler), $\alpha = 5$ and $\alpha = 10$.

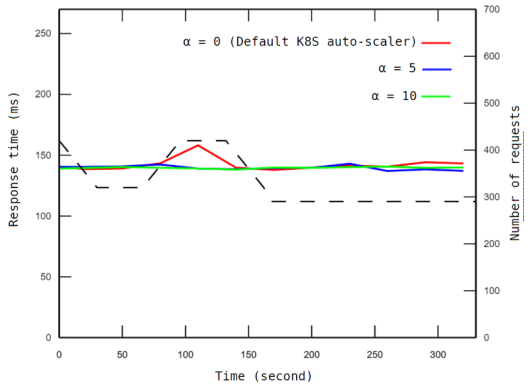

Fig. 4. Response time provided by three different Kubernetes (K8S) auto-scaling methods using $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$.

Fig. 3 showed that the value of $\alpha = 0$ (default Kubernetes auto-scaler) failed to provide the expected robustness of auto-scaling method. Since due to minor fluctuations in the workload, this auto-scaling approach stopped a container instance, and afterwards shortly started a new one again. This fact, for a while, negatively affected the response time offered by the auto-scaling method using $\alpha = 0$, shown in Fig. 4. Therefore, the slowest average response time (142.69 ms) was offered by the auto-scaling method using $\alpha = 0$ (default Kubernetes auto-scaler). A value of $\alpha = 10$ decreased the efficiency of the auto-scaling method because, in this case, the unnecessary container instance was not eliminated from the cluster at the right time. This is the reason why the auto-scaling method using $\alpha = 10$ provided the worst resource utilisation since it allocated more container instances (2 containers) than the other two approaches. Therefore, a higher value of α would result in longer periods of over-provisioned resources. For the experimentation in such predictable bursting workloads, the value of α has to be set to 5, that results in neither unnecessary over-provisioning of resources, nor too frequent changes in the number of running container instances.

5.2. Unpredictable bursting workload scenario

In order to choose the best time period for the adaptation interval or so-called Control Loop Time Period (CLTP) in an operational environment with unpredictable bursting workload in which there are unpredictable fluctuations in the number of requests, we performed a set of experiments according to three different time length: CLTP = 15 s, CLTP = 30 s (default Kubernetes auto-scaler) and CLTP = 60 s. To this end, a workload scenario has been inspected that includes a rising workload, a sudden inactive workload, an instantaneously increasing workload and finally a falling workload, respectively by passage of time. Fig. 5 shows the number of new container instances allocated by three different Kubernetes auto-scaling methods using 15-s adaptation interval, 30-s adaptation interval (default Kubernetes auto-scaler) and 60-s adaptation interval.

Fig. 5 demonstrates that the auto-scaling approach using 15-s adaptation interval is the fastest method since it was able to allocate a new container instance sooner than other methods in response to slowly rising workload from 240 to 700 requests. It also terminated the container added to the cluster when the workload density suddenly decreases in the off period. However, this adaptation action is not appropriate since there is an upcoming drastically increasing workload from 240 to suddenly 700 requests. In such a situation, the response time provided by the auto-scaling method using CLTP = 15 s was slow when there is a drastic workload after a while, shown in Fig. 5. Fig. 5

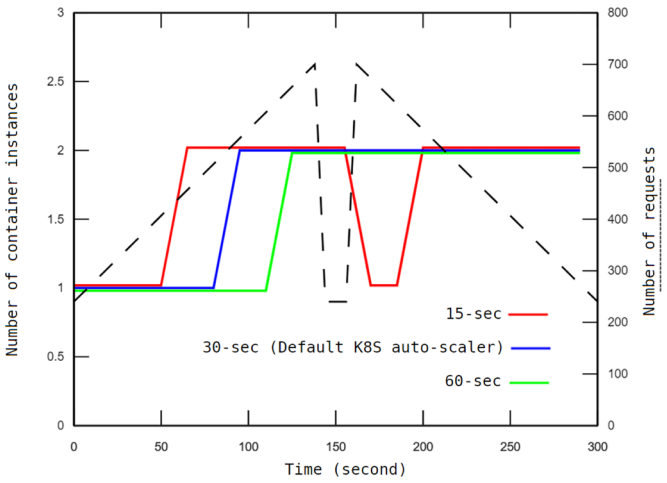

Fig. 5. Number of container instances allocated by three different Kubernetes (K8S) auto-scaling methods using CLTP = 15 s, CLTP = 30 s and CLTP = 60 s.

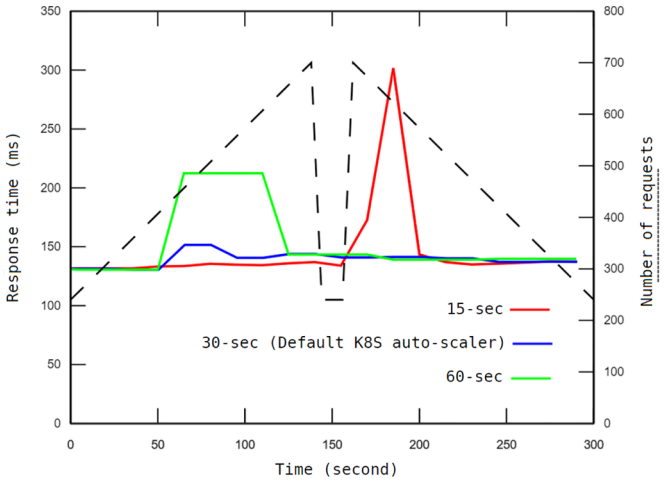

Fig. 6. Response time provided by three different Kubernetes (K8S) auto-scaling methods using CLTP = 15 s, CLTP = 30 s and CLTP = 60 s.

also shows that the auto-scaling approach using 60-s adaptation interval offers a slow response time during the gradually rising workload since it was not agile enough to recognise runtime changes in the workload density. The auto-scaling approach using 30-s adaptation interval (default Kubernetes auto-scaler) was the method which provided the fastest average response time almost steady over time (140.33 ms) on average in comparison with auto-scaling approaches using 15-s adaptation interval (148.42 ms) and 30-s adaptation interval (155.87 ms). Auto-scaling methods using CLTP = 15 s and CLTP = 30 s employed more container instances (1.67) during the experiment compared with another approach using CLTP = 60 s (1.57). Therefore, it can be concluded that in such unpredictable bursting workloads for the experimentation, the value of CLTP has to be set to 30 s to ensure that there would be no issue if any auto-scaling event takes place, (Fig. 6).

5.3. On-and-off workload scenario

In order to make sure if stopping at most one container instance in each adaptation interval can be a helpful auto-scaling strategy to be be adopted to handle on-and-off workload patterns, we analysed a set of experiments. The goal was to compare the default Kubernetes auto-scaling method with the strategy of stopping at most one container

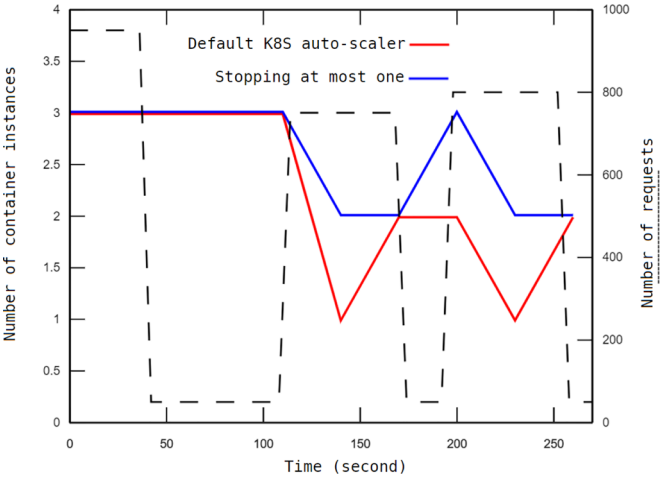

Fig. 7. Number of container instances allocated by two different Kubernetes (K8S) auto-scaling methods using the default algorithm and the strategy of stopping at most one container in each CLTP.

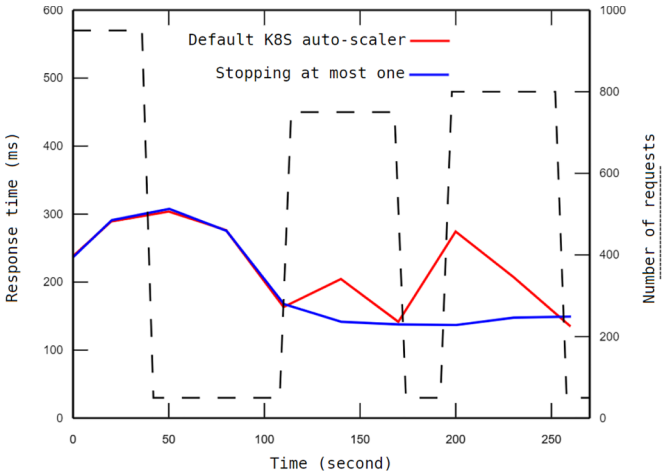

Fig. 8. Response time provided by two different Kubernetes (K8S) auto-scaling methods using the default algorithm and the strategy of stopping at most one container in each CLTP.

instance in each adaptation interval. To this end, an on-and-off workload scenario in which three different peak spikes appear periodically in short time intervals, as shown in Fig. 7. The first peak spike includes 950 requests, the second one consists of 750 requests, and finally 800 requests come into the last peak spike. The auto-scaling method, which stops at most one container in each adaptation interval, is more capable of timely provisioning an adequate number of containers to address peak spikes. This is because this proposed method does not terminate most of the container instances immediately when each peak spike disappears.

The auto-scaling method, which stops at most one container instance in each adaptation interval, allocated more microservices on average (2.6) than the default Kubernetes auto-scaling mechanism (2.3) in the on-and-off workload scenario. Moreover, the difference between these two distinct Kubernetes auto-scaling methods with regard to service response time can be taken into consideration enormous in this workload pattern, as depicted in Fig. 8.

In the on-and-off workload scenario, the average response time achieved by the strategy of stopping at most 1 container in each adaptation interval in this examined experiment were 190.77 ms that is almost 30 ms faster than the average response time provided by the default Kubernetes approach (219.96 ms). As depicted in Fig. 8, sudden active periods inappropriately result in an increase in the service time of the requests offered by the default auto-scaling approach.

6. Discussion

Over the course of our study, different types of threats to the validity of the theoretical basis and empirical results have been identified and a brief explanation of these threats is discussed below:

The microservices architecture is still evolving and growing, and hence there is much yet to be discovered [63]. Accordingly, there is generally a lack of consensus and limited guidance within the industry on not only what this architecture actually is, but also how it can be implemented. From this perspective, the present research work is upon the reviewed literature and based on conducting experiments considered to represent the common principles and best practices of scalable systems. This paradigm through containerised execution environments has been adopted by leading companies such as Netflix, Uber and Amazon.

If the value of the targeted CPU resource usage threshold is set closer to $Target_{cpu}$ = 100%, therefore the auto-scaling method has no chance to timely react to runtime changes in the workload density before a performance degradation arises. If the value of this threshold is

set less than 80%, then this may cause an over-provisioning issue which wastes costly resources. In the execution environment, if the workload trend is very even and predictable, this threshold for the utilisation of CPU resource can be pushed higher than 80%.

Different additional external factors (e.g. the mobility of the clients, unstable network conditions at the side of end-users, and client' network channel diversity, and so forth) may influence the end-users' experience. In fact, cloud-based applications may be adopted by various users from all around the world. This type of service quality issues due to connectivity problems are currently addressed by edge computing [64] that is out of our research scope in this paper.

It should be also noted that QoS properties of cloud-based infrastructure such as availability, bandwidth quality, etc. may vary at runtime, independent from the auto-scaling policies and the workload features, which the application experiences. Thus, when a container instance should be launched and deployed on a host machine, the cloud-based application provider requires to ensure that the host machine is capable of fulfilling the needed requirements of container instances. In this regard, the performance of running infrastructures also needs to be continuously characterised. This function is facilitated by our developed PrEstoCloud monitoring platform [65] at present.

The conducted experiments in this work are only based on Docker [66], although the execution environment may be implemented in other container-based virtualisation technologies such as LXC [67] and OpenVZ [68]. The reason is that all functionalities specified in the presented auto-scaling mechanisms are entirely independent characteristics from containerisation technologies, hardware features, providers of underlying cloud infrastructure, etc.

It is worth pointing out that if a certain service instance starts working at a host, it will expose its interfaces on specified port numbers, which should not conflict with the port numbers of other instances running on the same host machine. Thus, the Load–Balancer employs the information from the port numbers alongside protocols from incoming requests to distribute the traffic among appropriate instances running on the same host. All instances containerised running on a host get the same proportion of CPU cycles by default. In such condition, if computation in one container is idle, other containers can exploit the leftover CPU cycles of the host. If the Docker containerisation technology is used to containerise the service instances, it is possible to adjust different proportions allocated to running containers by employing a relative weighting approach. In this way, if containers running on a same host altogether try to exploit the whole 100% of the CPU time, these relative weights provide each container specifically access to the determined proportion of the host's CPU cycles because CPU time now is restricted.

Existing auto-scaling policies used by the current open-source container management platforms are primarily specified based upon the CPU utilisation. Some of the platforms claim that they can also consider the memory usage in their proposed auto-scaling methods by specifying scaling policies for memory resources that are similar to CPU-based auto-scaling methods. However, in practice, for memory-intensive applications like in-memory databases (for example HSQLDB), such proposed auto-scaling methods may not be very useful because of some key reasons:

- In order to scale a memory-intensive application, the data in memory should be shifted from one node to another one which is very time and bandwidth consuming. Therefore, unlike CPU, scaling memory is much more expensive.
- A lot of applications are not designed for dynamic memory size. For example, memory-intensive applications primarily determine the buffer cache size when starting, which is then fixed for the rest of time, and hence changing memory size does not make sense for them. We can certainly rely on swapping of the operating system to make it transparent, but performance will not be appropriate nevertheless.
- In order to run memory-intensive applications, using high-memory machines in advance needs to be considered as a significant requirement to deliver fast performance for workloads that process large data sets in memory.

As a further complement to this research work, the Kubernetes auto-scaler can be extended to also offer vertical auto-scaling for the allocation of disk and bandwidth resources to storage-intensive and network-intensive services, respectively. Vertical auto-scaling is a method to resize disk capacity or bandwidth assigned to a container on the current host machine depending on variations in the workload at runtime. This mechanism can dynamically increase disk and bandwidth resources allocated to an existing container instance when they are required or reduce resources when they are no longer needful. To this end, cloud-based infrastructure providers such as Amazon EC2 [69] and Microsoft Azure [70] provide vertical auto-scaling mechanism to change the bandwidth and database instance size on-the-fly. However, vertical auto-scaling is limited by the maximum hardware capacity of the individual host machine. Moreover, the service still has a single point of failure. For that reason, the combination of horizontal and vertical auto-scaling mechanisms can be exploited to the same service. When different microservices in an application system share one single monolithic database, it is extremely challenging to scale the whole database horizontally based on the traffic load. To overcome this problem, it is critical to allow each individual microservice to have its own separate database. Accordingly, a specific database can be replicated automatically if any performance issue is going to come up, and it can be horizontally scaled down to optimise the resource utilisation when the load decreases. In this way, there is a primary database instance to both write and read data, and there are read replicas which are just a read-only copy of the primary database instance. Each and every update to the primary database instance should be automatically reproduced on the read replicas immediately.

7. Conclusions

Cloud computing [71] has become the prevalent approach for offering many different types of services over the Internet based on a set of microservices packaged in containers. However, auto-scaling of computing-intensive microservice-native cloud-based applications has been a challenging issue due to runtime variations in the quantity and computational requirements of arrival requests. An auto-scaling method which is unable to address changing workload intensity over time will result in either resource over-provisioning situation where the usage of allocated resources is unacceptably low or resource under-provisioning situation where the application suffers from poor performance. In essence, the main problem is that existing auto-scaling methods, such as the one provided by the Kubernetes container orchestration platform, mainly use auto-scaling rules which cannot be very helpful for certain workload scenarios.

To improve the Kubernetes auto-scaler to be capable of satisfying application performance requirements (e.g. response time constraints), while optimising the utilisation of resources allocated to the application (e.g. number of containers), we proposed various auto-scaling factors. These influencing factors, which should be taken into consideration to handle different workload patterns, consist of (i) conservative constant (α), (ii) adaptation interval or Control Loop Time Period (CLTP), and (iii) stopping at most one container instance in each adaptation interval. In this study, the Kubernetes auto-scaler has been chosen for comparison to the methods using our proposed key influencing factors since it is considered as the most popular orchestration framework for containers in the advanced cloud-based production systems [72].

In order to analyse the impact of our proposed factors, we examined three different workload scenarios including (i) predictable bursting workload pattern, (ii) unpredictable bursting workload pattern, and (iii) on-and-off workload pattern.

The results of our evaluation demonstrated that the proposed factors significantly influence the performance of the Kubernetes auto-scaling method in order to ensure the QoS with respect to the response time and other benefits such as optimal resource utilisation. The key factor required to be considered for predictable bursting workload patterns is the conservative constant (α), which was concluded to be set to 5. In other words, in environments with predictable bursting workloads, considering a value of 5 as conservative constant along with the targeted CPU resource usage will lead to neither unnecessary over-provisioning of resources, nor too frequent changes in the number of running container instances. It was also concluded that on unpredictable bursting workload patterns, the CLTP needs to be specified as 30 s not only to be agile enough to recognise runtime fluctuations in the workload over time, but also to ensure that there would be no QoS degradation if any auto-scaling action occurs. Moreover, it was concluded that the default Kubernetes auto-scaler can be significantly improved if the policy of stopping at most one container instance in each CLTP is taken into account for on-and-off workload patterns.

CRediT authorship contribution statement

Salman Taherizadeh: Conceptualization, Data curation, Formal analysis, Writing - original draft, Writing - review & editing. **Marko Grobelnik:** Conceptualization, Data curation, Formal analysis, Writing - original draft, Writing - review & editing.

Declaration of Competing Interest

The authors declare no conflicts of interest or financial ties.

Acknowledgement

The authors would like to thank Dr. Zhiming Shen at Cornell University and Mr. Matjaz Rihtar at the Jozef Stefan Institute for their helpful suggestions and comments. This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732339(PrEstoCloud project: Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing).

References

[1] Ari I, Muhtaroglu N. Design and implementation of a cloud computing service for finite element analysis. Adv Eng Softw 2013;60–61:122–35. https://doi.org/10.1016/j.advengsoft.2012.10.003.
[2] Suram S, MacCarty NA, Bryden KM. Engineering design analysis utilizing a cloud platform. Adv Eng Softw 2018;115:374–85.
[3] Taherizadeh S, Stankovski V, Grobelnik M. A capillary computing architecture for dynamic internet of things: orchestration of microservices from edge devices to fog and cloud providers. Sensors 2018;18(9):2938.
[4] Kratzke N. About microservices, containers and their underestimated impact on network performance. Proc. of the Sixth International Conference on Cloud Computing, GRIDs, and Virtualization. Nice, France: IARIA; 2015. p. 165–9.
[5] Kovacs J, Kacsuk P, Emodi M. Deploying docker swarm cluster on hybrid clouds using occopus. Adv Eng Softw 2018;125:136–45.
[6] Kratzke N, Peinl R. Clouns - a cloud-native application reference model for enterprise architects. Proc. of 2016 IEEE 20th international enterprise distributed object computing workshop (EDOCW). Vienna, Austria: IEEE; 2016. p. 1–10. https://doi.org/10.1109/EDOCW.2016.7584353
[7] Kratzke N. A brief history of cloud application architectures. Appl Sci 2018;8(8):1368. https://doi.org/10.3390/app8081368.
[8] Mackie RI. Application of service oriented architecture to finite element analysis. Adv Eng Softw 2012;52:72–80.
[9] Stubbs J, Moreira W, Dooley R. Distributed systems of microservices using docker and serfnode. 2015 7th international workshop on science gateways. Budapest, Hungary: IEEE; 2015. p. 34–9.
[10] Thönes J. Microservices. IEEE Softw 2015;32(1). 116–116
[11] Kratzke N, Quint PC. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. J Syst Softw 2017;126:1–16. https://doi.org/10.1016/j.jss.2017.01.001.
[12] Al-Sharif ZA, Jararweh Y, Al-Dahoud A, Alawneh LM. Accrs: autonomic based cloud computing resource scaling. Cluster Comput 2017;20(3):2479–88. https://doi.org/10.1007/s10586-016-0682-6.
[13] Kukade PP, Kale G. Auto-scaling of micro-services using containerization. Int J Sci Res(IJSR) 2015;4(9):1960–3.
[14] Kan C. Docloud: An elastic cloud platform for web applications based on docker. Proc. of 2016 18th international conference on advanced communication technology (ICACT). Pyeongchang, South Korea: IEEE; 2016. p. 478–83. https://doi.org/10.1109/ICACT.2016.7423439
[15] Baresi L, Guinea S, Leva A, Quattrocchi G. A discrete-time feedback controller for containerized cloud applications. Proc. of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. Seattle, WA, USA: ACM; 2016. p. 217–28. https://doi.org/10.1145/2950290.2950328
[16] Dube P., Gandhi A., Karve A., Kochut A., Zhang L. Scaling a cloud infrastructure. 2016. US Patent 9,300,553.
[17] Tsoumakos D, Konstantinou I, Boumpouka C, Sioutas S, Koziris N. Automated, elastic resource provisioning for nosql clusters using tiramola. Proc. of 2013 13th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid). Delft, Netherlands: IEEE; 2013. p. 34–41. https://doi.org/10.1109/CCGrid.2013.45
[18] Kubernetes horizontal pod auto-scaling, 2019 (accessed september 15). 2019. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[19] Rutten E, Marchand N, Simon D. Feedback control as mape-k loop in autonomic computing. Software engineering for self-adaptive systems III. Assurances. Springer; 2018. p. 349–73.
[20] Khazaei H., Barna C., Litoiu M. Performance modeling of microservice platforms considering the dynamics of the underlying cloud infrastructure. 2019. ArXiv preprint arXiv:1902.03387, 1–15.
[21] Wang N, Varghese B, Matthaiou M, Nikolopoulos DS. Enorm: a framework for edge node resource management. IEEE TransServComput 2017:1–14. https://doi.org/10.1109/TSC.2017.2753775.
[22] Cardellini V, Grassi V, Presti FL, Nardelli M. Optimal operator placement for distributed stream processing applications. Proc. of the 10th ACM international conference on distributed and event-based systems. ACM; 2016. p. 69–80.
[23] Zhang F, Tang X, Li X, Khan SU, Li Z. Quantifying cloud elasticity with container-based autoscaling. Future Gen Comput Syst 2019;98:672–81.
[24] Nadgowda S, Suneja S, Kanso A. Comparing scaling methods for linux containers. Proc. of 2017 IEEE international conference on cloud engineering (IC2E). Vancouver, Canada: IEEE; 2017. p. 266–72.
[25] Xu M, Toosi AN, Buyya R. Ibrownout: an integrated approach for managing energy and brownout in container-based clouds. IEEE Trans Sustain Comput 2018;4(1):53–66.
[26] Rossi F, Nardelli M, Cardellini V. Horizontal and vertical scaling of container-based applications using reinforcement learning. Proc. of 2019 IEEE 12th international conference on cloud computing (CLOUD). Milan, Italy: IEEE; 2019. p. 329–38.
[27] Zhao A, Huang Q, Huang Y, Zou L, Chen Z, Song J. Research on resource prediction model based on kubernetes container auto-scaling technology. Proc. of IOP conference series on materials science and engineering, Vol. 569. China: IOP Publishing; 2019. P. 052092
[28] Rodriguez M.A., Buyya R. Containers orchestration with cost-efficient autoscaling in cloud computing environments. 2018. ArXiv preprint arXiv:1812.00300, 1–22.
[29] Liu H, Chen S, Bao Y, Yang W, Chen Y, Ding W, Shan H. A high performance, scalable dns service for very large scale container cloud platforms. Proc. of the 19th international middleware conference industry. Rennes, France: ACM; 2018. p. 39–45.
[30] Ogawa K, Kanai K, Nakamura K, Kanemitsu H, Katto J, Nakazato H. Iot device virtualization for efficient resource utilization in smart city iot platform. Proc. of 2019 IEEE international conference on pervasive computing and communications workshops (PerCom Workshops). Kyoto, Japan: IEEE; 2019. p. 419–22.
[31] Tamrakar K, Yazidi A, Haugerud H. Cost efficient batch processing in amazon cloud with deadline awareness. Proc. of 2017 IEEE 31st international conference on advanced information networking and applications (AINA). Taipei, Taiwan: IEEE; 2017. p. 963–71.
[32] Zhang W, Xu L, Duan P, Gong W, Liu X, Lu Q. Towards a high speed video cloud based on batch processing integrated with fast processing. Proc. of 2014 international conference on identification, information and knowledge in the internet of things. Beijing, China: IEEE; 2014. p. 28–33.
[33] Wen Y, Wang Z, Zhang Y, Liu J, Cao B, Chen J. Energy and cost aware scheduling with batch processing for instance-intensive iot workflows in clouds. Future Gen Comput Syst 2019:39–50.
[34] Taher NC, Mallat I, Agoulmine N, El-Mawass N. An iot-cloud based solution for real-time and batch processing of big data: application in healthcare. Proc. of 2019 3rd international conference on bio-engineering for smart technologies (BioSMART). Paris, France: IEEE; 2019. p. 1–8.
[35] Li W, Liang J, Ma X, Qin B, Liu B. A dynamic load balancing strategy based on haproxy and tcp long connection multiplexing technology. Proc. of the Euro-China conference on intelligent data analysis and applications. Springer; 2019. p. 36–43. https://doi.org/10.1007/978-3-030-03766-6_5
[36] Qu C, Calheiros R, Buyya R. Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. Concurr Comput 2017;29(12). https://doi.org/10.1002/cpe.4126.
[37] Nadgowda S, Suneja S, Isci C. Paracloud: bringing application insight into cloud operations. Proc. of the 9th USENIX workshop on hot topics in cloud computing (HotCloud 17), USENIX association, Santa Clara, California. 2017.
[38] Toosi A, Qu C, de Assunção M, Buyya R. Renewable-aware geographical load balancing of web applications for sustainable data centers. J Netw Comput Appl 2017;83:155–68. https://doi.org/10.1016/j.jnca.2017.01.036.
[39] Grozev N, Buyya R. Dynamic selection of virtual machines for application servers in cloud environments. Research advances in cloud computing. Springer; 2017. p. 187–210.

[40] Chen H, Wang Q, Palanisamy B, Xiong P. Dcm: Dynamic concurrency management for scaling n-tier applications in cloud. Proc. of 2017 IEEE 37th international conference on distributed computing systems (ICDCS). Atlanta, GA, USA: IEEE; 2017. p. 2097–104. https://doi.org/10.1109/ICDCS.2017.22
[41] Kampars J, Pinka K. Auto-scaling and adjustment platform for cloud-based systems. Proc. of the 11th international scientific and practical conference, Vol. 2, Rezekne, Latvia. 2017. p. 52–7. https://doi.org/10.17770/etr2017vol2.2591
[42] Sangpetch A., Sangpetch O., Juangmarisakul N., Warodom S. Thoth: automatic resource management with machine learning for container-based cloud platform. 2017. https://doi.org/10.5220/0006254601030111.
[43] Singh V, Peddoju SK. Container-based microservice architecture for cloud applications. Proc. of 2017 international conference on computing, communication and automation (ICCCA). Greater Noida, India: IEEE; 2017. p. 847–52. https://doi.org/10.1109/CCAA.2017.8229914
[44] Nitu V, Teabe B, Fopa L, Tchana A, Hagimont D. Stopgap: elastic vms to enhance server consolidation. Software 2017;47(11):1501–19. https://doi.org/10.1002/spe.2482.
[45] Wajahat M., Karve A., Kochut A., Gandhi A. Mlscale: a machine learning based application-agnostic autoscaler. Sustain Comput;. 10.1016/j.suscom.2017.10.003.
[46] Xu X-L, Jin H, Wu S, Wu X-L, Li Y. Fama: a middleware for fast deploying and auto scaling towards multitier applications in clouds. J Internet Technol 2015;16(6):987–97. https://doi.org/10.6138/JIT.2015.16.6.20130506.
[47] Foley F, Lemm M. Designing performance metrics at godaddy. 217-004. Harvard Business School Case; 2016. p. 1–17.
[48] Cosentino V, Izquierdo JLC, Cabot J. Findings from github: methods, datasets and limitations. Proc. of 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). Austin, USA: IEEE; 2016. p. 137–41. https://doi.org/10.1109/MSR.2016.023
[49] Cai L, Wang H, Xu B, Huang Q, Xia X, Lo D, Xing Z. Answerbot: an answer summary generation tool based on stack overflow. Proc. of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. Tallinn, Estonia: ACM; 2019. p. 1134–8. https://doi.org/10.1145/3338906.3341186
[50] Gilbert E. Widespread underprovision on reddit. Proc. of the 2013 conference on computer supported cooperative work. Texas, USA: ACM; 2013. p. 803–8. https://doi.org/10.1145/2441776.2441866
[51] Khatouni AS, Mellia M, Marsan MA, Alfredsson S, Karlsson J, Brunstrom A, Alay O, Lutu A, Midoglu C, Mancuso V. Speedtest-like measurements in 3g/4g networks: the monroe experience. Proc. of 2017 29th international teletraffic congress (ITC 29), Vol. 1. Genoa, Italy: IEEE; 2017. p. 169–77. https://doi.org/10.23919/ITC.2017.8064353
[52] Pham C, Cao P, Kalbarczyk Z, Iyer RK. Toward a high availability cloud: techniques and challenges. Proc. of IEEE/IFIP international conference on dependable systems and networks workshops (DSN 2012). Boston, USA: IEEE; 2012. p. 1–6. https://doi.org/10.1109/DSNW.2012.6264687
[53] Leitner P, Inzinger C, Hummer W, Satzger B, Dustdar S. Application-level performance monitoring of cloud services based on the complex event processing paradigm. Proc. of 2012 Fifth IEEE international conference on service-oriented computing and applications (SOCA). Taipei, Taiwan: IEEE; 2012. p. 1–8. https://doi.org/10.1109/SOCA.2012.6449437
[54] Sciullo L, Aguzzi C, Felice MD, Cinotti TS. Wot store: enabling things and applications discovery for the w3c web of things. Proc. of 2019 16th IEEE annual consumer communications & networking conference (CCNC). Las Vegas, USA: IEEE; 2019. p. 1–8. https://doi.org/10.1109/CCNC.2019.8651786
[55] Shackelford A. Working with aws opsworks. Beginning Amazon Web services with node. js. Springer; 2015. p. 31–59. https://doi.org/10.1007/978-1-4842-0653-9_2
[56] Apte V. What did i learn in performance analysis last year?: teaching queuing theory for long-term retention. Proc. of the 10th ACM/SPEC international conference on performance engineering (ICPE 2019). Mumbai, India: ACM; 2019. p. 71–7. https://doi.org/10.1145/3302541.3311526
[57] Han R, Guo L, Ghanem MM, Guo Y. Lightweight resource scaling for cloud applications. Proc. of 2012 12th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid). Ottawa, ON, Canada: IEEE; 2012. p. 644–51. https://doi.org/10.1109/CCGrid.2012.52
[58] Lv Z-H, Wu J, Bao J, Hung PC. Ocrem: openstack-based cloud datacentre resource monitoring and management scheme. Int J High Perform Comput Netw 2016;9(1–2):31–44. https://doi.org/10.1504/IJHPCN.2016.074656.
[59] Singhi G, Tiwari D. A load balancing approach for increasing the resource utilisation by minimizing the number of active servers. Int J Comput SecuritySource Code Anal 2017;3(1):11–5.
[60] Monil MAH, Rahman RM. Implementation of modified overload detection technique with vm selection strategies based on heuristics and migration control. Proc. of 2015 IEEE/ACIS 14th international conference on computer and information science (ICIS). Las Vegas, NV, USA: IEEE; 2015. p. 223–7. https://doi.org/10.1109/ICIS.2015.7166597
[61] Alonso A, Aguado I, Salvachua J, Rodriguez P. A metric to estimate resource use in cloud-based videoconferencing distributed systems. Proc. of 2016 IEEE 4th international conference on future internet of things and cloud (FiCloud). Vienna, Austria: IEEE; 2016. p. 25–32. https://doi.org/10.1109/FiCloud.2016.12
[62] Alonso A, Aguado I, Salvachua J, Rodriguez P. A methodology for designing and evaluating cloud scheduling strategies in distributed videoconferencing systems. IEEE Trans Multimedia 2017;19(10):2282–92. https://doi.org/10.1109/TMM.2017.2733301.
[63] Hamzehloui MS, Sahibuddin S, Ashabi A. A study on the most prominent areas of research in microservices. Int J Mach LearnComput 2019;9(2):242–7. https://doi.org/10.18178/ijmlc.2019.9.2.793.
[64] Taherizadeh S, Jones A, Taylor I, Zhao Z, Stankovski V. Monitoring self-adaptive applications within edge computing frameworks: a state-of-the-art review. J Syst Softw 2018;136:19–38. https://doi.org/10.1016/j.jss.2017.10.033.
[65] PrEstoCloud monitoring platform, 2019 (Accessed September 15, 2019). https://github.com/salmant/PrEstoCloud/blob/master/Monitoring-Agent/InfrastructureMonitoringAgent.java.
[66] Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux J 2014(239).
[67] Qiu Y, Lung C-H, Ajila S, Srivastava P. Lxc container migration in cloudlets under multipath tcp. Proc. of 2017 IEEE 41st annual computer software and applications conference (COMPSAC), Vol. 2. Turin, Italy: IEEE; 2017. p. 31–6. https://doi.org/10.1109/COMPSAC.2017.163
[68] Jaikar A, Shah SAR, Bae S, Noh SY. Performance evaluation of scientific workflow on openstack and openvz. Proc. of international conference on cloud computing. Guangzhou, China: Springer; 2016. p. 126–35.
[69] Portella G, Rodrigues GN, Nakano E, Melo AC. Statistical analysis of amazon ec2 cloud pricing models. Concurr Comput 2018:1–16. https://doi.org/10.1002/cpe.4451.
[70] Persico V, Marchetta P, Botta A, Pescape A. On network throughput variability in microsoft azure cloud. Proc. of 2015 IEEE global communications conference (GLOBECOM). San Diego, USA: IEEE; 2015. p. 1–6. https://doi.org/10.1109/GLOCOM.2015.7416997
[71] Lovas R, Nagy E, Kovacs J. Cloud agnostic big data platform focusing on scalability and cost-efficiency. Adv Eng Softw 2018;125:167–77.
[72] Shah J, Dubaria D. Building modern clouds: Using docker, Kubernetes & google cloud platform. Proc. of 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC). Las Vegas, USA: IEEE; 2019. p. 0184–9. https://doi.org/10.1109/CCWC.2019.8666479