Int. J. Appl. Math. Comput. Sci., 2019, Vol. 29, No. 2, 227–244
DOI: 10.2478/amcs-2019-0017

amcs

MULTILAYERED AUTOSCALING PERFORMANCE EVALUATION: CAN VIRTUAL MACHINES AND CONTAINERS CO–SCALE?

VLADIMIR PODOLSKIY [a,*], ANSHUL JINDAL [a], MICHAEL GERNDT [a]

[a]Chair of Computer Architecture and Parallel Systems
Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany
e-mail: {v.podolskiy,anshul.jindal,gerndt}@in.tum.de

The wide adoption of cloud computing by businesses is due to several reasons, among which the elasticity of the cloud virtual infrastructure is the definite leader. Container technology allows increasing the flexibility of an application by adding another layer of virtualization. The containers can be dynamically created and terminated, and also moved from one host to another. A company can achieve a significant cost reduction and increase the manageability of its applications by allowing the running of containerized microservice applications in the cloud. Scaling for such solutions is conducted on both the virtual infrastructure layer and the container layer. Scaling on both layers needs to be synchronized so that, for example, the virtual machine is not terminated with containers still running on it. The synchronization between layers is enabled by *multilayered cooperative scaling*, implying that the autoscaling solution of the virtual infrastructure layers is aware of the decisions of the autoscaling solution on the container layer and vice versa. In this paper, we introduce the notion of cooperative multilayered scaling and the performance of multilayered autoscaling solutions evaluated using the approach implemented in ScaleX (previously known as Autoscaling Performance Measurement Tool, APMT). We provide the results of the experimental evaluation of multilayered autoscaling performance for the combination of virtual infrastructure autoscaling of AWS, Microsoft Azure and Google Compute Engine with pods horizontal autoscaling of Kubernetes by using ScaleX with four distinct load patterns. We also discuss the effect of the Docker container image size and its pulling policy on the scaling performance.

Keywords: cooperative scaling, multilayered autoscaling, autoscaling performance, autoscaling evaluation, ScaleX.

1. Introduction

Cloud computing is based on virtualization. This technology represents the hardware as a pool of resources to be sliced and provided to users in the form of virtual machines (VM): it also helps businesses to scale their applications. Scalability of cloud applications is one of the main reasons behind the wide adoption of cloud computing. Most of IaaS cloud services providers (CSP) offer autoscaling services to adapt the VMs to the changing demand.

A new type of virtual entity, the container, allows the user to design loosely coupled applications consisting of multiple small building blocks. These building block (microservices) implement a small set of functions and communicate with other microservices. The runtime environment is packed within the container allowing the execution of the microservice anywhere, both on bare metal or in a VM. Containerization extends the cloud computing paradigm from several viewpoints: finer and more accurate management of the running application; decoupling of the application from the underlying resources; self-containment of the microservices. However, the management of several virtualization layers can become very challenging.

Additional layers of virtualization introduce a higher level of flexibility and control. Aside from an obvious performance loss when introducing additional layers of virtualization, the absence of the awareness of these virtualization layers about one another could become a significant concern. With the lack of coordination between the multiple layers, one can expect that, e.g., the termination of the VM due to autoscaling may result in the termination of the running containers and, potentially, in unfulfilled requests. By increasing the awareness of the virtualization layers about their neighbours with *multilay-*

*Corresponding author

ered cooperative scaling, flexible multilayered application deployment may acquire predictable scaling behavior.

With the rich set of metrics and approaches to evaluate existing autoscaling solutions[1] and their policies,[2] it may become difficult to select them for a particular case. One should always distinguish between the assessment of the quality of the autoscaling policy and the evaluation of the autoscaling solution performance. In this paper, emphasis is put on the evaluation of the performance of autoscaling solutions with the autoscaling policy being fixed for all the cases. Particularly, the performance of multilayered autoscaling is presented for the combination of infrastructure autoscaling by AWS, Azure, and GCE with container-level autoscaling based on Kubernetes.

The key contributions of this paper are the theoretical framework for cooperative scaling involving different types of virtual entities (VMs, containers), a refined approach to QoS-based multilayered autoscaling performance evaluation based on scaling intervals, an extended description of the performance evaluation tool ScaleX and its use, results of cooperative autoscaling performance evaluation for the public IaaS clouds (AWS, Azure and GCE), autoscaling solutions combined with the autoscaling solution of Kubernetes, and results of the experiment highlighting the impact of the container image size and pulling policy type on the scaling performance.

The following section introduces the theoretical framework and the background. Section 3 covers the multilayered autoscaling performance evaluation approach. Section 4 focuses on the autoscaling performance measurement tool ScaleX. Section 5 provides experimental results for evaluating the performance of multilayered autoscaling and for estimating the potential impact of the container image size and pulling policy on the autoscaling performance. Section 6 discusses the obtained experimental results. The related works and the position of the paper in the existing research are summarized in Section 7. Section 8 concludes the paper.

2. Theoretical framework and background

2.1. Scalability and elasticity. *"The concept [of scalability] connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement"* (Bondi, 2000).

The scalability of cloud applications and the underlying virtual infrastructure means that the number or the capacity of the virtual entities forming the cloud application (containers) and the virtual infrastructure (virtual machines, VMs) can change dynamically in response to the changing workload. Often, the scalability of a cloud is specified by the term "elasticity" (Herbst *et al.*, 2013). Elasticity differs from scalability in that the state of the virtual infrastructure and cloud applications, acquired as a result of the increased demand for the capacity, lasts only until the amount of requests and resource utilization starts decreasing. Thus, cloud applications and the virtual infrastructure can return to the original state in terms of resources consumed. Elasticity could be viewed from two viewpoints—of the cloud services provider and of the cloud user.

The elasticity of the cloud as viewed by the CSP revolves around the hardware resources provided to the cloud users. The pool of hardware resources allocated for the particular user in the form of VMs can arbitrarily grow and shrink. The CSP provides virtual machines for the customer. This enables the dynamic change of the amount of resources, e.g., elasticity on the infrastructure level. This requires the provisioning of sufficient hardware as well as the scheduling of VMs. Cloud scalability on the CSP's side resides on hardware virtualization using hypervisors (e.g., Xen, Hyper-V) and on the hardware resources allocation for the virtual machines scheduling (Sotomayor *et al.*, 2009a; 2009b). The society's concerns about the ecology and the CSPs concerns about the cost of electricity introduce power consumption as another parameter to be considered when scaling (Jakobik *et al.*, 2017).

The elasticity of the cloud as viewed by the cloud user puts the cloud application in the center. A cloud application may consist of multiple microservices in containers. The elasticity from the user's prospect is in the opportunity to increase or decrease the number of microservice instances, thus regulating the capacity of the application in terms of the processed requests. Elasticity in that sense might be supported by out-of-the-box CSPs (e.g., AWS Lambda). The scalability of the application could be achieved within the IaaS cloud by running the containers on top of VMs. We assume this scenario when discussing multilayered scaling spanning several virtualization layers.

2.2. Changing the cloud capacity through scaling.

2.2.1. Types of scaling. From the point of view of the cloud user, scaling can be conducted in two ways—either by increasing the capacity of the existing virtual entities or by increasing the number of virtual entities.

Vertical scaling allows changing the resource capacity of a virtual entity. In the case of a virtual machine, it could be achieved, e.g., by increasing the amount of allocated memory or the number of virtual CPU

[1] An autoscaling solution is a piece of software that implements autoscaling.

[2] An autoscaling policy is a set of precise rules that determine how the virtual infrastructure is scaled based on its monitored parameters.

cores assigned to the VM. Vertical scaling in such a case could be represented by substituting the VM of the type with a smaller capacity for the VM of the type with a larger capacity (scale-up) or vice versa (scale-down). A container can also be vertically scaled by changing the maximal amount of resources allocated to it (e.g., the maximal amount of processor time as millicore parameter of Kubernetes pods[3]).

Horizontal scaling allows changing the capacity of the pool of virtual entities by introducing new entities or removing old ones. Horizontal scaling requires *load balancing*. Horizontal scaling for large-scale applications is preferable as it imposes a homogeneity requirement on the groups of virtual entities and does not require stopping running the application in order to change the underlying virtual machine.

2.2.2. Scaling the virtual infrastructure. The virtual infrastructure is a *"software-based IT infrastructure being hosted on another physical infrastructure and meant to be distributed as a service as in the cloud computing Infrastructure as a Service (IaaS) delivery model"*.[4]

In the case of the IaaS model, the virtual infrastructure could be represented as one or more VMs that are allocated on the physical servers in one of the CSP's data centers. From the point of view of the CSP, scaling the virtual infrastructure is always connected to allocating more or less hardware resources. From the point of view of the IaaS cloud services' user, virtual infrastructure scaling is represented either by a change in the number of virtual machines (horizontal scaling) or by a change in the type of virtual machines (vertical scaling). The IaaS model supports the automation of VM scaling via automatic scaling (or autoscaling).

A detailed discussion of autoscaling is provided in Section 2.3.

2.2.3. Scaling containerized applications. An application container could be defined as *"a controlling element for an application instance that runs within a type of virtualization scheme called container-based virtualization. [...] in container-based virtualization, the individual instances share an operating system."*[5]

Each application container can be an enclosed functional unit of the application that provides services to other containers. This viewpoint allows considering a container to be a lightweight entity that includes a relatively compact code base, though in general several microservices could be packed in the same container.

Although containers support both types of scaling, the most widely used is horizontal scaling. Container scaling can be automated by using container orchestration tools (e.g., DockerSwarm or Kubernetes). Vertical scaling of a container is not explicitly implemented, although it could be simulated by increasing or decreasing container resource limits.[6] The automation of container on-the-fly vertical scaling is being researched (Al-Dhuraibi *et al.*, 2017). Kubernetes' abstraction of pods as a group of containers sharing the network and the storage[7] leverages the opportunity to decrease the degree of container isolation, allowing access to the shared resources. It allows scheduling and running application containers on clusters of physical or virtual machines. The abstraction of a pod serves as a basis to scale groups of containers. Automatic horizontal scaling thereof is supported by out-of-the-box Kubernetes by monitoring CPU utilization and changing the number of pods in the replication controller. The vertical autoscaling feature is also being actively developed.[8]

By leveraging the ability to scale both the virtual infrastructure and containers, we may introduce the notions of multilayered and cooperative scaling.

2.2.4. Multilayered and cooperative scaling. Simultaneous scaling on several layers of virtualization introduces additional issues that do not appear when scaling either the virtual infrastructure or a containerized application. When putting containers on virtual machines, it is necessary to ensure that during the scale-out the necessary amount of resources is available in the form of VMs. It could also occur that a VM is terminated with containers running on top of it. In the case of Kubernetes, accidental termination of the master VM during the scale-down may yield a fault of the whole deployment. To avoid such problems, each virtualization layer should be aware scaling actions happening on the other layer. Multilayered scaling with enabled awareness of scaling actions on other layers can be called *cooperative scaling*.

Cooperative scaling supposes the presence of several virtualization layers. By putting virtual entities of one layer on top of virtual entities of the other layer, a dependency is established. Though the multilayered structure allows more flexibility, the presence of a dependency yields scaling challenges. To reduce damaging effects of scaling on multiple layers, each layer should receive updates about scaling actions taken on another one. Allowing scaling solutions on different layers to communicate, the availability of the application

[3]https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.

[4]https://www.techopedia.com/definition/30459/virtual-infrastructure.

[5]https://www.techopedia.com/definition/31114/application-container.

[6]https://docs.docker.com/config/containers/resource_constraints/.

[7]https://kubernetes.io/docs/concepts/workloads/pods/pod/.

[8]https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

during and after scaling can be ensured.

Cooperative scaling exhibits (a) support for scaling on multiple layers induced by triggering the same rule of the scaling policy, (b) coordinated scheduling of scaling activities on all virtualization layers, (c) the availability of data about the planned scaling to the scaling solution of the neighbour layer, (d) scaling flexibility, e.g., the ability to use different scaling policies/thresholds to conduct scaling on different virtualization layers.

Currently, production-level cooperative scaling requires an additional functionality to enable the selection of scaling policies parameters for each layer.

2.3. Autoscaling.

2.3.1. Reactive autoscaling. Autoscaling (automatic scaling, auto scaling, auto-scaling) is the technology that enables automatic provisioning and termination of virtual entities to adapt the resource capacity to changes in the demand. The key difference from the previous discussion of scaling is in the *automation* of this process. The automation is achieved by using a monitoring service to retrieve relevant resource utilization metrics on which alarms and triggers can be defined.

State-of-the-art autoscaling solutions by public CSPs are of the reactive type. Reactive autoscaling is the type of autoscaling that either deploys or terminates the predefined amount of virtual entities as a response to the change of some metric. Hence, reactive autoscaling takes into account only the *given parameters*, which are either provided by the cloud administrator or are measured by the monitoring solution. The amount of change, i.e., the number of virtual entities to be allocated or terminated, is encoded as a set of rules. The most severe limitation of reactive autoscaling is the small amount of time allocated for the scaling action in the case of an increased workload. The most recent research tries to overcome it on the CSP's side by employing continuous (online) updates of the optimal autoscaling configuration (Guo *et al*., 2018).

Each IaaS/PaaS/FaaS CSP provides its own native reactive autoscaling solution. Following, we will introduce brief information on autoscaling solutions of AWS, Microsoft and Google that were used for conducting the experiments in the paper.

AWS (Amazon Web Services) Auto Scaling[9] is a part of the services offered by Amazon in its IaaS public cloud. The core concept of AWS Auto Scaling is an Auto Scaling Group (ASG). An ASG is a set of different Amazon Elastic Compute Cloud (EC2) instances (VMs) sharing similar characteristics and being subject to the same scaling policies. Therefore, every VM in the group has the same Amazon Machine Image (AMI) and the same hardware characteristics. The load distribution among the VMs is automated by Elastic Load Balancer (ELB). Amazon CloudWatch provides the performance data used in the scaling rules.

Microsoft Azure Autoscale[10] comes in two modes: metric-based and scheduled. The metric-based autoscale service of Microsoft represents the common way of autoscaling as in the AWS case. The scheduled mode allows the user to write a scaling schedule to adjust the infrastructure according to time markers. Similarly to AWS, Azure also groups VM instances into a group that is managed by its autoscaling solution. These groups are called scaling sets. Despite the fundamental similarity of scaling sets to auto scaling groups of AWS, they are slightly different, e.g., the user is not allowed to attach the shell script to the VM template—the necessary file should be provided directly in the VM image. Each scaling set is scaled based on the autoscale settings. They determine the capacity and the set of scaling rules identifying the thresholds for different metrics.

Google Compute Engine (GCE) autoscaling[11] is based on a managed instance group. It is a scalable group of the same virtual machines that behaves as a uniform entity. Each group contains a load balancer. GCE autoscaling is based on the metrics provided by Google Stackdriver.[12] Out-of-the-box it supports autoscaling based solely on the average CPU utilization. Moreover, Stackdriver introduces additional metrics as well as the ability to create custom ones.

2.3.2. Scheduled autoscaling. Reactive autoscaling is appropriate for most practical cases, though the time between the decision to scale-out and the new instances being able to serve the requests may impact the quality of service (QoS). A particular example could be the Christmas season for a web shop. With the rising number of customers, the virtual infrastructure may not meet the demand, and reactive autoscaling tries to adapt the virtual infrastructure to the growing demand. However, during the autoscaling process, the web site may not have enough capacity to serve all the Christmas orders, which will result in a revenue decrease. On the other hand, loosening a reactive autoscaling policies could result in the overprovisioning and increase of the costs. A partial solution to this problem is provided in the form of scheduled autoscaling.

The concept of *scheduled autoscaling* is simple. Based on the knowledge of load patterns, the cloud administrator devises a scaling schedule which

[9] `https://aws.amazon.com/autoscaling/`.

[10] `https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling`.

[11] `https://cloud.google.com/compute/docs/autoscaler/`.

[12] `https://cloud.google.com/stackdriver/`.

contains the information on how many virtual entities should be added to or removed from the virtual infrastructure/containerized application at the specific time. Most CSPs offer scheduled autoscaling along with reactive autoscaling. AWS, for example, started to provide the scheduled autoscaling service for applications in 2017.[13] Kubernetes also supports pods scheduling.[14] Scheduled autoscaling could be combined with reactive autoscaling, both to capture the expected load changes and to react to spontaneous variations.

2.3.3. Predictive autoscaling. Certain drawbacks of the widely used types of autoscaling could be avoided by incorporating a smarter approach to autoscaling that is able to extract a value from the monitoring data. *Predictive autoscaling* (also known as proactive autoscaling) leverages historical data about the application and the virtual infrastructure collected by the monitoring solution. The collected historical data can be in various forms: application traces, logs, time series, etc. These data are needed for the derivation of the models used to extrapolate the future values of the specific metrics. For example, the collected requests per second time series could be used to derive a model and forecast (predict, extrapolate) the request per second value for a specific service at some moment in the future.

In addition to the forecast, performance models of the software and the virtual entity as well as a management component implementing autoscaling are required (Bauer *et al.*, 2017). With these components, the predictive autoscaling solution (i) collects monitoring data for the forecasted parameter, (ii) derives forecasting models, (iii) derives application and virtual infrastructure performance models, (iv) derives the scaling policy that ensures the provision of such an amount of virtual entities that would be able to serve the forecast workload, (v) executes scaling actions.

Predicitive autoscaling has to be dynamically adapted to changes in the application and in user demand patterns. This includes updating the prediction model, the performance models and the derived scheduling policy.

Though predictive autoscaling is yet to be provided by CSPs, various solutions are already widely represented in the research literature (Roy *et al.*, 2011; Nikravesh *et al.*, 2015; Moore *et al.*, 2013). Moreover, some orchestration solutions contain predictive autoscaling as work-in-progress.[15]

As unpredictable changes in the load may also happen in such a dynamic environment, predictive and reactive autoscaling might be combined (Liu *et al.*, 2015).

2.4. Evaluation of autoscaling. The evaluation of autoscaling could be conducted from different points of view. First of all, one needs to distinguish between the *evaluation of autoscaling policies* and the *evaluation of the autoscaling solution*.

The autoscaling policy is a set of rules that governs the autoscaling process, be it reactive, scheduled, or predictive. An autoscaling policy evaluation framework should provide the set of autoscaling solution implementation-independent metrics that allow comprehensive evaluation of the quality of the specific autoscaling policy, e.g., the tendency to over- or underprovision the resources, the frequency of the scaling events, and the cost of the scaled virtual infrastructure (**?**). Although the evaluation of autoscaling policies with various metrics provides a useful decision-making framework allowing selection of the appropriate autoscaling policy based on multiple criteria, it does not capture the implementation-specific characteristics of the autoscaling solution, e.g., its performance.

The performance could be measured by the time to take a scaling decision and the time for starting additional virtual entities. Another approach could be based on user-level performance metrics, e.g., the number of QoS violations for the application (Jindal *et al.*, 2017). The evaluation of autoscaling solution performance using both techniques is discussed further in the paper.

3. Approach to evaluate autoscaling performance

3.1. Single-layered autoscaling performance evaluation. The main purpose of reactive autoscaling is to react to a change in the application/virtual infrastructure load. The *reaction time* or *autoscaling latency* is the time between the decision of the autoscaler and the final adaptation of the resources (Jindal *et al.*, 2017). For application level metrics, the fraction of the autoscaler latency where QoS requirements were violated is an important metric.

The autoscaling latency can be defined based on *Current Amount of Instances (CAI)* and *Desired Amount of Instances (DAI)*. Each autoscaling solution contains autoscaling rules that determine conditions triggering autoscaling actions (scale-in or scale-out in the horizontal scaling case). As the deployment of the virtual entity takes time for resource allocation, booting and configuring, DAI will differ from CAI during some time. When autoscaling is completed, CAI and DAI will be equal. Thus, we consider the described time interval to be the autoscaling latency. If t_{start} is the start time and t_{end} is

[13] https://aws.amazon.com/about-aws/whats-new/2017/11/scheduled-scaling-now-available-for-application-auto-scaling/.

[14] https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/.

[15] https://github.com/mattjmcnaughton/kubernetes/tree/add-predictive-autoscaling.

the end time of autoscaling then the *autoscaling interval* $T_{\text{autoscale}} = [t_{\text{start}}, t_{\text{end}}]$ is an interval such that $\forall t \in T_{\text{autoscale}} : CAI(t) \neq DAI(t)$. If $\forall t \in T_{\text{autoscale}} : CAI(t) < DAI(t)$, then $T_{\text{autoscale}}$ is a *scale-out interval*. If $\forall t \in T_{\text{autoscale}} : CAI(t) > DAI(t)$, then $T_{\text{autoscale}}$ is a *scale-in interval*. The *autoscaling latency* is defined as $t_{\text{end}} - t_{\text{start}}$.

Other performance measures for autoscaling solutions are based on the notion of the quality of service for the cloud application. For autoscaling solution performance evaluation, two user-level quality metrics were chosen: the *cloud application response time* and the *maximal failure rate*. They are directly influenced by the quality of the corresponding autoscaling solution services. These metrics have the corresponding QoS limitations which may be imposed by cloud application users:

- required response time (RRT),
- required maximal failure rate (RMFR).

To measure the performance of the autoscaling solution, we now define the following two metrics: response time violation and maximal failure rate violation. Both are the fraction of the autoscaling interval where the corresponding QoS requirement was violated.

As the performance metric for autoscaling is based on the cloud application response time, we identify such a metric as the fraction of the autoscaling interval $T_{\text{autoscale}}$ with the service response time (RT) above the threshold. So, if $T_{\text{autoscale}} = [t_a, t_b]$ and $T_{\text{high RT}} = [t_c, t_d]$, where $t_a \leq t_c \leq t_d \leq t_b$, then the autoscaling solution performance metric can be computed as (Jindal *et al.*, 2017)

$$\text{RTV}(T_{\text{autoscale}}) = \frac{t_d - t_c}{t_b - t_a}, \tag{1}$$

where $\text{RTV}(T_{\text{autoscale}})$ is the fraction of the autoscaling latency where the response time requirement is violated. In more complex cases, $T_{\text{high RT}}$ could be a set of intervals, e.g., $T_{\text{high RT}} = \{T^{(1)}_{\text{high RT}}, T^{(2)}_{\text{high RT}}, \dots, T^{(p)}_{\text{high RT}}\}$.

Similarly to $\text{RTV}(T_{\text{autoscale}})$, we compute the fraction of the autoscaling interval with the maximal failure rate (MFR) higher than a predefined threshold. If $T_{\text{autoscale}} = [t_a, t_b]$ and $T_{\text{high MFR}} = [t_e, t_f]$, where $t_a \leq t_e \leq t_f \leq t_b$, then $\forall t \in T_{\text{high MFR}}$ and the performance metric is computed as in (Jindal *et al.*, 2017)

$$\text{MFRV}(T_{\text{autoscale}}) = \frac{t_f - t_e}{t_b - t_a}. \tag{2}$$

The presented autoscaling performance metrics (1) and (2) allow us to evaluate the performance of the autoscaling solution directly without considering the particular autoscaling policy implemented by the solution.

3.2. Performance evaluation of multilayered autoscaling. The general idea behind multilayered autoscaling performance evaluation is to measure it as a fraction of autoscaling time with violated QoS requirements spanning multiple layers of virtualization. Figure 1 illustrates the concept of a multilayered autoscaling interval and the multilayered autoscaling performance measurement.

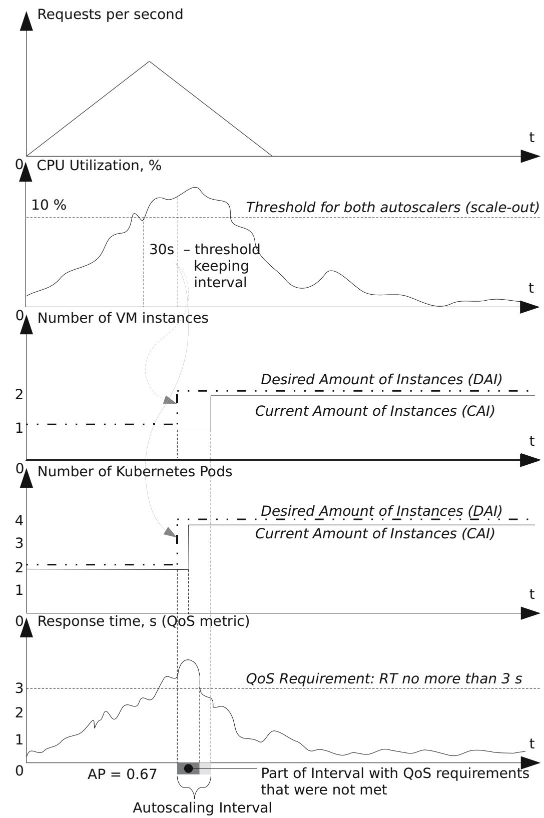

Fig. 1. Abstract example of a response time-based multilayered autoscaling performance metric.

The multilayered autoscaling interval is a set of time intervals on different layers of virtualization covering the whole autoscaling event. The first two graphs in Fig. 1 show how CPU utilization changes in response to change in the amount of requests. The two following graphs highlight change in the number of VMs and pods in response to increased CPU utilization (the rule was set to increase the number of pods and VMs by one after reaching a 10% CPU utilization and sustaining it for 30 seconds). If we assume the QoS requirement of the response time is 3 seconds as shown in the last graph, then we can identify the fraction of the autoscaling time interval during which the requirement on the response time was not met. The approach for multilayered autoscaling performance evaluation closely follows this

example.

The main research problem in multilayered autoscaling performance evaluation is to identify a set of autoscaling events on different virtualization layers to be considered as a single autoscaling event spanning multiple layers. In the methodology defined by Jindal *et al.* (2017), this challenge is resolved using the notion of time locality, i.e., the scaling events on multiple layers of virtualization are considered to be part of the same multilayered autoscaling event if autoscaling events on the previous virtualization layer and on the next one have the same direction of scaling (scale-in or scale-out) and the event of the dependent virtualization layer (e.g., containerized application) follows the scaling event on the lower layer (e.g., virtual infrastructure).

$T_{\text{as}}^{(1)} = \{T_1^{(1)}, T_1^{(2)}, \ldots, T_1^{(n)}\}$ is a set of autoscaling intervals, with $T_i^{(1)}$ being an autoscaling interval on the virtualization layer that is closest to the hardware (e.g., native CSP's autoscalers are on this level); superscripts denote the layers, subscripts signify that an interval in the set belongs to the particular multilayered autoscaling interval, i.e., $T_{\text{as}}^{(1)}$. In turn, each element of the set could also be a set of intervals on the corresponding layer of virtualization. The intervals of time between the autoscaling intervals are not taken into account when computing multilayered autoscaling duration in order to reduce the effect of the performance of the underlying hardware, though, depending on the goals of performance evaluation, one might attribute these intervals to autoscaling duration.

In general, a single-layer autoscaling interval $T_1^{(i)}$ is considered an element of the set of autoscaling intervals for a *single case* of multilayered autoscaling if and only if $\forall j : j > i$ and we have the following conditions fulfilled:

$$T_1^{(i)} \prec T_2^{(i)}, \tag{3}$$

$$T_1^{(i)} \preceq T_1^{(j)}, \tag{4}$$

$$\frac{CAI(t_j) - DAI(t_j)}{|CAI(t_j) - DAI(t_j)|} \cdot \frac{CAI(t_i) - DAI(t_i)}{|CAI(t_i) - DAI(t_i)|} = 1, \tag{5}$$

with $\forall t_j \in T_2^{(j)}$ and $\forall t_i \in T_1^{(i)}$. Thus, in order to be considered a part of a single multilayered autoscaling event, autoscaling on level i should (i) occur when all previous layers have entered a stable state, i.e., $CAI = DAI$, after the corresponding previous j-th autoscaling has already occurred, and (ii) be of the same direction (scale-in or scale-out) as each of the autoscalings on previous layers.

Example CAI/DAI plots for the two-layered autoscaling case are presented in Fig. 2.

If $A = \{a_1, a_2, \ldots, a_m\}$ is a set of indices that enumerates all the members of T_{as}, then the duration of multilayered autoscaling can be determined with the following formula:

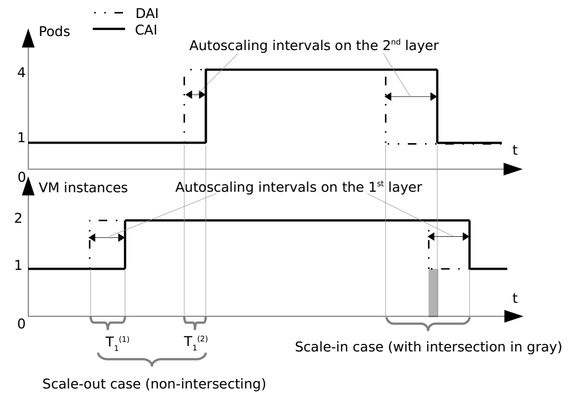

Fig. 2. Example of multilayered autoscaling event identification for two-layered virtualization (virtual infrastructure and containerized application).

$$\Delta T_{as} = |T_1^{(a_1)}| + \sum_{i=1}^{m} (|T_1^{(a_{i+1})}| - |T_1^{(a_{i+1})} \cap T_1^{(a_i)}|). \tag{6}$$

The formula (6) takes into account a possible intersection of the autoscaling intervals on different layers by adding a delta of the interval further in time (if a pair of consecutive intervals overlaps) or the whole interval (if a pair of consecutive intervals does not overlap, i.e., $T_1^{(a_{i+1})} \cap T_1^{(a_i)} = \emptyset$). The formula (6) with respect to constraints (3)–(5) gives us an estimate for the duration of a single autoscaling event for an arbitrary multilayered cloud application. In the simplest case of two layers, the formula becomes

$$\Delta T_{as} = |T_1^{(1)}| + (|T_1^{(2)}| - |T_1^{(2)} \cap T_1^{(1)}|). \tag{7}$$

With respect to metrics, the previously introduced formulas (1) and (2) are still in use, but the notion of the autoscaling interval on which they are computed is changed:

$$T_{\text{autoscale}} = \bigcup_{i=1}^{m} T_1^{(a_i)}. \tag{8}$$

The presented multilayered autoscaling performance measurement approach is implemented in the autoscaling performance measurement tool ScaleX, which is discussed in the following section.

Comparison with the existing evaluation schemes. The presented user-side autoscaling performance evaluation approach and metrics should be considered complimentary to the existing approaches and metrics (Ilyushkin *et al.*, 2017; Evangelidis *et al.*, 2017).

It is not designed to be the only one used when evaluating the performance of autoscaling solutions. Such metrics as the overprovisioning, underprovisioning, instability, or cost of the overprovisioned virtual infrastructure should also be measured and calculated when evaluating an autoscaling solution. The proposed metrics and approach are different from the existing in that (i) several virtualization levels are considered when evaluating the performance, and (ii) the metrics are designed to capture the user-side performance, i.e., what the user will encounter during the autoscaling process. Thus, the approach and metrics are necessary for *comprehensive evaluation of the autoscaling solution's performance*.

4. ScaleX: An autoscaling performance measurement tool

4.1. ScaleX overview. ScaleX is a user-friendly web service-based horizontal single-layered and multi-layered autoscaling performance measurement tool designed and implemented by the authors (Jindal *et al.*, 2017). The tool is implemented in Node.js. The architecture of ScaleX and the communications between its modules in a typical use case are shown in Fig. 3.

ScaleX is composed of multiple modules, overall matching the microservice architecture. Each module consists of components for handling a particular task.

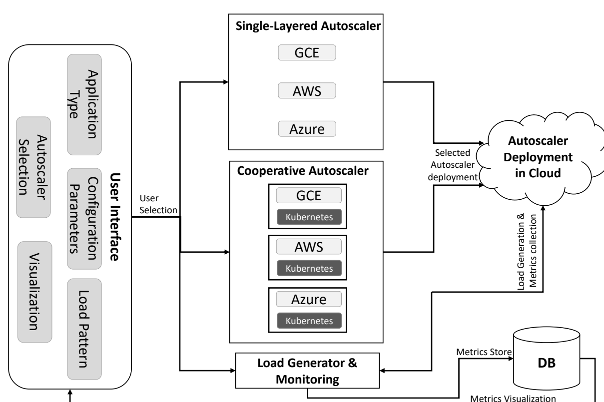

Fig. 3. High-level architecture of ScaleX.

Apart from measuring the performance of the deployed autoscaling solution, ScaleX allows cloud application deployment for multiple CSP native autoscalers with a single command. With the same command, the user may configure autoscaling parameters. The following subsections present the modules of ScaleX.

4.2. User interface. The user interface (UI) module of ScaleX interacts with the user and provides him or her with an opportunity to select or configure different tool parameters. It comprises five components, which are discussed in the following paragraphs.

Application Type Setting Component enables the selection of an application to be deployed for testing from a list of predefined applications.[16]

The user can select an application from any of these categories to conduct the tests of the autoscaling solution(s) on it using a particular autoscaling decision metric. At the moment, ScaleX supports only CPU utilization as an autoscaling metric.

Configuration Parameters Component provides the user with the autoscaling solution configuration functionality. This component configures the chosen autoscaling solution using parameter values specified by the user. Each autoscaling solution supported by ScaleX provides all important autoscaling configuration parameters.

Single-Layered Autoscaler Configuration Component allows configuring horizontal scaling of the virtual infrastructure based on a numbers of parameters: the type of instance, minimal and maximal number of instances, a scaling decision metric and its threshold, or an autoscaling policy.

Multilayered Autoscaler Configuration Component allows configuring horizontal scaling for both the virtual infrastructure and the containerized application.[17] The configuration parameters for both virtualization layers are configured by this component:

1. CSPs IaaS Autoscaler: all the parameters as listed for the Single-Layered Autoscaler Component.

2. Kubernetes Horizontal Pod Autoscaler (HPA): the minimal and maximal numbers of pods and the pod scaling decision metric with its threshold. As of now, only CPU utilization is supported as the autoscaling decision metric in Kubernetes.

Load Pattern Configuration Component provides an interface to the workload request generator integrated in the ScaleX. It allows the user to select a preconfigured load pattern in order to test the performance of the autoscaling solution. At the moment, four load patterns are supported by ScaleX:

- *Linear Increase Load Pattern* corresponds to the linearly increasing number of requests per second during the test time.

- *Linear Increase and Constant Load Pattern* corresponds to the number of requests per second pattern that linearly increases during the first half of

[16]This component also supports arbitrary application.

[17]The autoscaling of the containerized application is enabled via the Kubernetes orchestration tool.

the test time and then stabilizes for the rest of the test.

- *Random Load Pattern* corresponds to the randomly increasing and decreasing number of requests per second during the test time.

- *Triangle Load Pattern* corresponds to the number of requests per second pattern that linearly increases during the first half of the test time and then decreases with the same slope during the rest of the test.

The user can configure the following parameters of the load pattern selected: the number of concurrent clients, the maximal number of requests, the maximal duration of a test, the request timeout, the HTTP request method, the request body, the content type, and the number of requests per second.

Autoscaler Selection Component allows the user to pick a supported autoscaling solution from the list. A single-click functionality to deploy and undeploy an autoscaling solution is also provided.

Visualization Component shows plots and tables for all the performance metrics. The user can use these plots and tables to compare the autoscaling solutions.

4.3. Single-layered autoscaler interface. ScaleX comprises interfaces to single-layered CSP native IaaS autoscaling solutions. This module supports the deployment process for different CSP native autoscaling solutions. The purpose of this module is to combine configuration parameters with the selected application and to deploy it using the chosen autoscaler. Currently, 3 CSP native autoscaling solutions are supported by ScaleX: GCE, AWS, and Azure. The following highlights the deployment procedure of these CSPs native autoscaling solutions.

GCE Autoscaler. A common VM instance template is created with a start script to deploy the application on a VM start. This template is used as a basis to form the managed instance group. The GCE autoscaling solution is configured using the parameters provided by the user. Following, a load balancer is created to direct the load to the managed instance group. Stackdriver logging and GCE monitoring are used to collect the metrics data for this group.

AWS Autoscaler. At the starting point, an instance launch configuration is created with the same start script that is used with the GCE autoscaler to deploy the application on a VM start. The launch configuration is used to form an Auto Scaling Group (ASG) in AWS Cloud using the user parameters. The scaling parameters and policies are added to the ASG in the next step. To direct the load to this ASG, an Elastic Load Balancer (ELB) is added. It serves as a single endpoint for the load generation workload—internally the load is distributed among the ASG instances. AWS Cloud Watch is used to collect the metrics data for the whole ASG as well as for the individual EC2 instances.

Azure Autoscaler. At the beginning, a customized VM image is created with a start script to deploy the application when the VM finishes booting. This image is then used in the VM scale set for replication along with the user-defined autoscaling configuration parameters. The VM scale set serves the same purpose as the managed instance group in GCE or an ASG in the AWS virtual infrastructure. The load balancer is also added. Azure monitoring APIs are used to collect the metrics for the VM scale set.

Figure 4 shows the deployment to test the single-layered virtual infrastructure autoscaling solution provided by IaaS CSP.

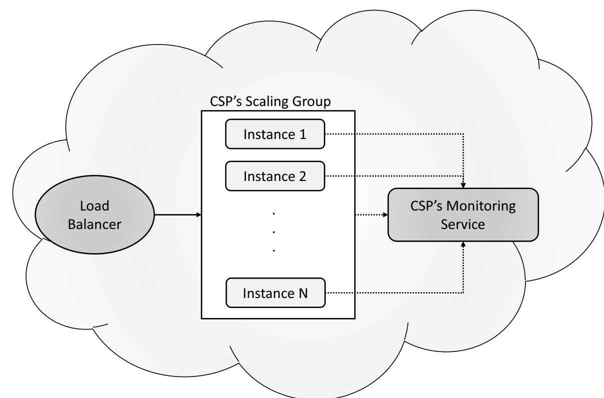

Fig. 4. Deployment to test single-layered virtual infrastructure autoscaling.

4.4. Multilayered autoscaler interface. In the scope of this module, ScaleX combines CSP native IaaS autoscaling solutions with the Kubernetes horizontal pod autoscaler to form a two-layered autoscaling solution. Additionally, the module combines configuration parameters with the selected application and deploys in the selected IaaS cloud as a Kubernetes cluster. A monitoring service is attached as part of the deployment to collect the performance metrics and store these data in a database.

A multilayered autoscaler needs to enable synchronization between autoscaling solutions on different virtualization layers. A sequence of actions is undertaken by the module in order to enable the synchronization. A separate VM instance is created for the Kubernetes master before starting the nodes (formerly

known as minions) as part of a managed instance group in GCE, an ASG in AWS or a VM scale set in Azure. Such a configuration allows scaling the nodes based on the observed workload. ScaleX employs *kubeadm*[18] to deploy the Kubernetes cluster. Kubeadm is used to initialize the master. Once the master becomes ready, the nodes can join it by running a specific command. A start script is added as part of the each CSP IaaS autoscaling solution. This start script is modified to include all the configurations and commands required by the VM instance to join the Kubernetes cluster. Hence, when a new instance is created in a managed instance group in GCE, an ASG in AWS or a VM scale set in Azure, it automatically joins the Kubernetes cluster. During the scale-in, a VM instance to be terminated is removed safely by the master from the Kubernetes cluster so that no further pods are scheduled to run there, whereas the pods that already run there are rescheduled to run on other nodes. This mechanism adds the awareness of the virtual infrastructure about containerized virtualization with pods, therefore synchronization between layers hinders premature termination of the VM with running pods by the native autoscaling solution of the IaaS CSP.

The user parameters are used to configure the Kubernetes cluster along with the scaling parameters and policies in the CSPs IaaS autoscaler. The master IP-address is used as a single endpoint for the generated workload and internally the master distributes the load among Kubernetes cluster nodes. For the collection of the metrics from the Kubernetes cluster, ScaleX uses Heapster[19] coupled with the InfluxDB.[20] Both are deployed in the cluster with the chosen application. Metrics data are continuously fetched by ScaleX from both Heapster (data about the Kubernetes cluster) and the native CSPs monitoring service (data for CSPs instances and the autoscaling group), and stored in the database for visualization and analysis. Figure 5 shows the deployment to test the multilayered autoscaling solution.

4.5. Load generator and monitoring. This module generates the workload of the desired pattern and directs it to the IP address of the deployed application. ScaleX employs the customized version of Node.js-based Loadtest[21] for load generation. The number of clients that generate the load and the load generation time are configured for the selected load pattern. To prevent a single load generation node from becoming the bottleneck, the module implements the master-slave architecture—the generation of the requests is distributed

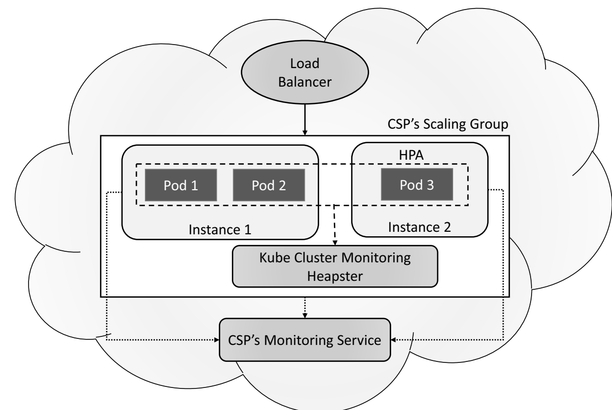

Fig. 5. Deployment to test multilayered autoscaling.

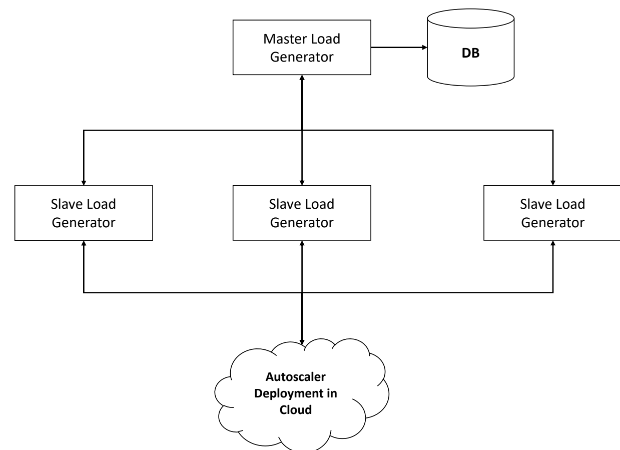

Fig. 6. Load generator architecture.

among the slave nodes. After completion of each request, the performance results are sent back to the master node and stored in the database. The monitoring part of the component periodically fetches the data from different monitoring services deployed as part of the autoscaling solutions and stores them in the database for further performance analysis. Figure 6 depicts the architecture of the ScaleX load generator module.

4.6. Database. ScaleX uses MongoDB to store the performance data. The reason to choose MongoDB as the storage for the performance data and generated workload parameters is the support for high insert rates. The drawback, however, is the missing transaction safety, which in principle is not relevant for ScaleX. An additional advantage of this NoSQL database is that it supports storing measurements with varying data schema as the monitoring solutions used to collect the performance and requests data employ different data schemes.

[18]https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/.
[19]https://github.com/kubernetes/heapster.
[20]https://www.influxdata.com/time-series-platform/influxdb/.
[21]https://www.npmjs.com/package/loadtest.

5. Multilayered autoscaling performance evaluation

5.1. Experimental setting. In our experiments, we used all workload patterns supported by ScaleX. The total time for each test was 20 minutes; the request timeout was 6.5 seconds. The number of simulated concurrent clients for each load generation was 50; they were deployed on a single VM instance not participating in the experiment. For each pattern except for random, the start value of the request rate was 1 request per second, whereas the increase/decrease was set to 3. The random load pattern starts at 50 requests per second and increases/decreases randomly. Load generation is distributed among the concurrent clients. The computer-intensive test application computes the sum of prime numbers between 1 and 1000000 when called. Executing this computation from multiple clients increases CPU utilization. Hence we can observe the autoscaling solution effect on the deployment.

The VM configuration for experiments on different clouds is provided in Table 1. The image of the operating system used in every configuration was Ubuntu 16.04 LTS.

Table 2 highlights the configuration of Kubernetes autoscaling. The target number of pods in a deployment or replication controller is automatically adjusted based on the formula

$$N_{\text{tgt.pods}} = \left\lceil \frac{\sum_{\text{pods}} U_{\text{cur.CPU}}}{U_{\text{tgt.CPU}}} \right\rceil, \tag{9}$$

where $\sum_{\text{pods}} U_{\text{cur.CPU}}$ is the overall current pod CPU utilization and $U_{\text{tgt.CPU}}$ is a target CPU utilization. Table 3 contains the configuration settings for CSP native autoscalers.

The experiment was conducted several times for each combination of autoscaling solutions using the Multiple Consecutive Trials (MCT) methodology. As the results demonstrated relative stability in performance and in scaling patterns, and we wanted to highlight autoscaling behavioral features that might be lost by averaging, we chose to show in the paper the results of a *single* experiment. In the future, ScaleX will be extended to support the more accurate Randomized Multiple Interleaved Trials (RMITs) methodology (Abedi and Brecht, 2017).

Table 1. Experimental VM configuration.

CSP	Instance type	Memory	vCPUs
GCE	–	2 GB	1 vCPU
AWS	t2.small	2 GB	1 vCPU
Azure	A1_V2 Standard	2 GB	1 vCPU

Table 2. Experimental configuration: Kubernetes autoscaling solution.

Instances	Min. pods	Max. pods	Scaling metric	Threshold
1(master) 3(nodes)	1	10	CPU Utilization	20 %

Table 3. Experimental configuration: CSP autoscaling solution.

Min. instances	Max. instances	Scaling metric	Threshold
1	3	CPU Utilization	20 %

5.2. Experimental results.

5.2.1. Evaluating the performance of multilayered autoscaling.
AWS Auto Scaling + Kubernetes. The data collected in the scope of AWS Auto Scaling/Kubernetes, experiment demonstrate that the scale-out action conducted by the native AWS autoscaling solution lags the scale-out action by Kubernetes which results in the deployment of new pods on a single VM (see rows B and C in Fig. 7). This behavior indicates coordination problems between multiple virtualization layers. The lack of coordination leads to the deployment of new pods on the old VM instances whereas the newly added VM could only have a single pod (in particular, rows B and C in Fig. 7 show that all the pods were started when the number of running VMs was 1). Such a disproportion leads to load balancing issues and results in a latency increase, as shown in row D. The scale-in times for AWS Auto Scaling are larger than the scale-out times which is indicated by row C. A possible explanation is that AWS Auto Scaling conducts more time-consuming actions during the termination of the VM. In B-1 and B-3 the current number of pods is reduced, although Kubernetes did not request this. The reason is that infrastructure scaling decided to decommission VMs although pods were still running there.

Microsoft Azure Autoscale + Kubernetes. Microsoft Azure Autoscale demonstrates the slowest autoscaling behavior (see row C in Fig. 8). Both scale-out and scale-in times are significantly larger than for GCE and AWS. Based on plots in rows D and E, we can conclude that the overall performance of a single Azure VM instance is better than that of the competitors since with the later scale-out (compared with AWS and GCE) Azure is still able to keep the performance on par with other tested configurations. The most probable cause could be the newer hardware used to host the VMs at Microsoft datacenters. It is possible to notice in the Azure graphs in rows B–E for all tested patterns that the performance

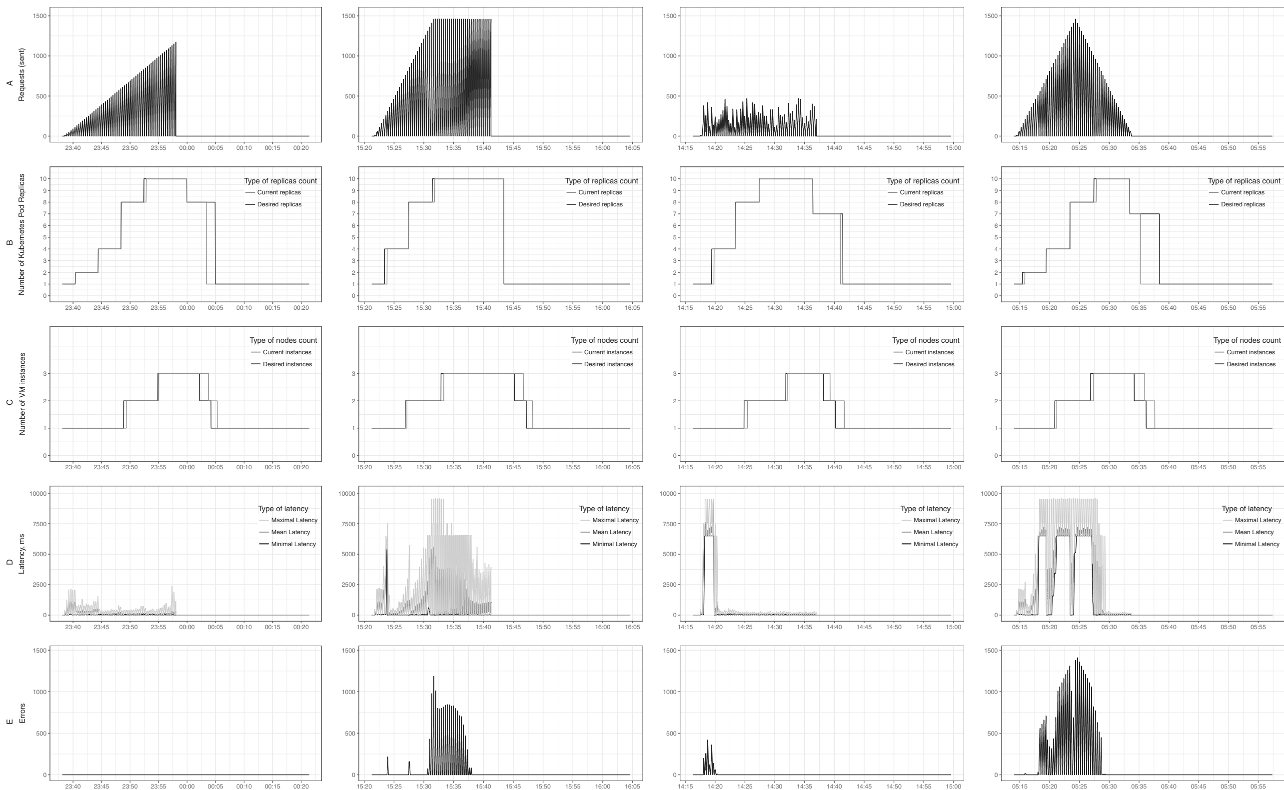

Fig. 7. Graphical representation of the AWS/Kubernetes multilayered autoscaling solution. Columns (load patterns): (first) linearly increasing, (second) linearly increasing and constant, (third) random, (fourth) triangle. Rows: (A) total number of requests sent, (B) DAI and CAI of Kubernetes pods, (C) DAI and CAI of AWS VM instances, (D) response time, (E) requests failure rate.

mostly depends on the underlying hardware and on the scaling of Kubernetes pods, and not on the actual number of VMs.

Google Compute Engine (GCE) autoscaling + Kubernetes. Based on rows D and E in Fig. 9, we can conclude that the GCE/Kubernetes deployment exhibits the best performance. Looking deeper at row C, we can see a cause for such behavior—most part of the experiment interval is covered by scaled-out VM instances. If pod replicas are distributed over more VMs, they can take a higher load. However, there is always a tradeoff between the size of VMs, their number and the number of pod replicas. For example, an early VM scale-out will result in a cost increase. The experiments also show that GCE autoscaling is faster at making scaling decisions and providing VM instances than AWS and Azure in the scope of the evaluated case. Additional VMs are added early and thus the new pods are better distributed, which is illustrated by rows B and C.

Discussion. The comparison of the AWS, Azure, and GCE deployments for the tested case is conducted using two metrics: (i) the amount of QoS violations, (ii) the fraction of the autoscaling interval where the QoS requirements were violated. In Tables 4 and 5 we summarize the number of QoS violations by a load pattern. A response time QoS requirement violation is identified by the mean response time being higher than 6.5 s. The maximal failure rate QoS violation is arbitrarily indicated by the amount of errors higher than 10. Table 6 goes into more details on the multilayered autoscaling performance of the studied deployments.

The results in Tables 4 and 5 show that the GCE/Kubernetes deployment in the tested case outperforms both AWS and Azure. The initial cause for this is the fast decision-making process for VMs instances scale-out that allows distributing the new pods more or less evenly. However, with respect to the amount of requests ending up in an error, the results show no clear leader. For example, the GCE/Kubernetes deployment shows problems handling the Linear Increase and Random Load patterns in the tested case. If we refer to the error plot E-2 in Fig. 9, we might notice really small intervals with a high amount of errors.

The conclusion of the comparison might be formulated in such a way that the GCE/Kubernetes deployment provides the best autoscaling performance

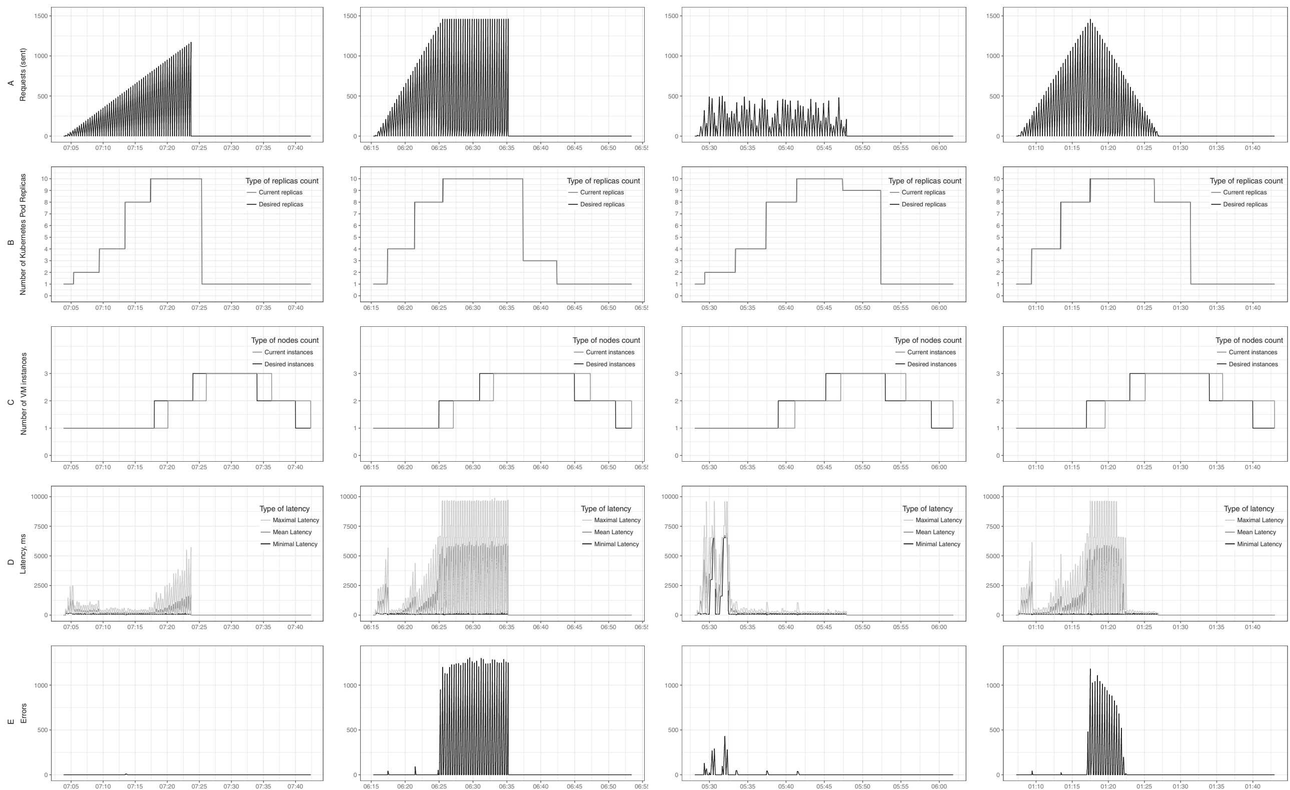

Fig. 8. Graphical representation of the Azure/Kubernetes multilayered autoscaling solution. Columns (load patterns): (first) linearly increasing, (second) linearly increasing and constant, (third) random, (fourth) triangle. Rows: (A) total number of requests sent, (B) DAI and CAI of Kubernetes pods, (C) DAI and CAI of Azure VM instances, (D) response time, (E) requests failure rate.

Table 4. Performance comparison based on the amount of response time requirement QoS violations.

Load Pattern	Amount of RT requirement violations		
	AWS	Azure	GCE
Linear Increase	0	0	0
Linear and Constant	17962	40934	**250**
Random	1545	2570	**1127**
Triangle	6418	15222	**76**

Table 5. Performance comparison based on the amount of maximal failure rate QoS violations.

Load Pattern	Amount of MFR requirement violations		
	AWS	Azure	GCE
Linear Increase	**0**	**0**	382
Linear and Constant	25707	42725	**1251**
Random	**1720**	2570	1835
Triangle	9954	16368	**845**

for the experimental case without careful selection of the parameters for autoscaling policy (e.g., the CPU threshold).

Table 6 summarizes parameters of all CSP layer *scale-out intervals*. The column RTV represents a fraction of the autoscaling interval with the violated response time requirement, whereas MFRV represents the same for the requirement on the maximal failure rate. Since not all the deployments have exposed the clear synchronized multilayered behavior, the autoscaling performance was evaluated only during the scaling of VM instance groups.

We can observe a clear autoscaling performance problem for Azure. It is not only the slowest, but also exhibits more performance problems during the scaling time. Scale-out times of other deployments for all the patterns are mostly in the 5—30 second interval, which could be considered appropriate, although even these times can exhibit performance problems (refer to high

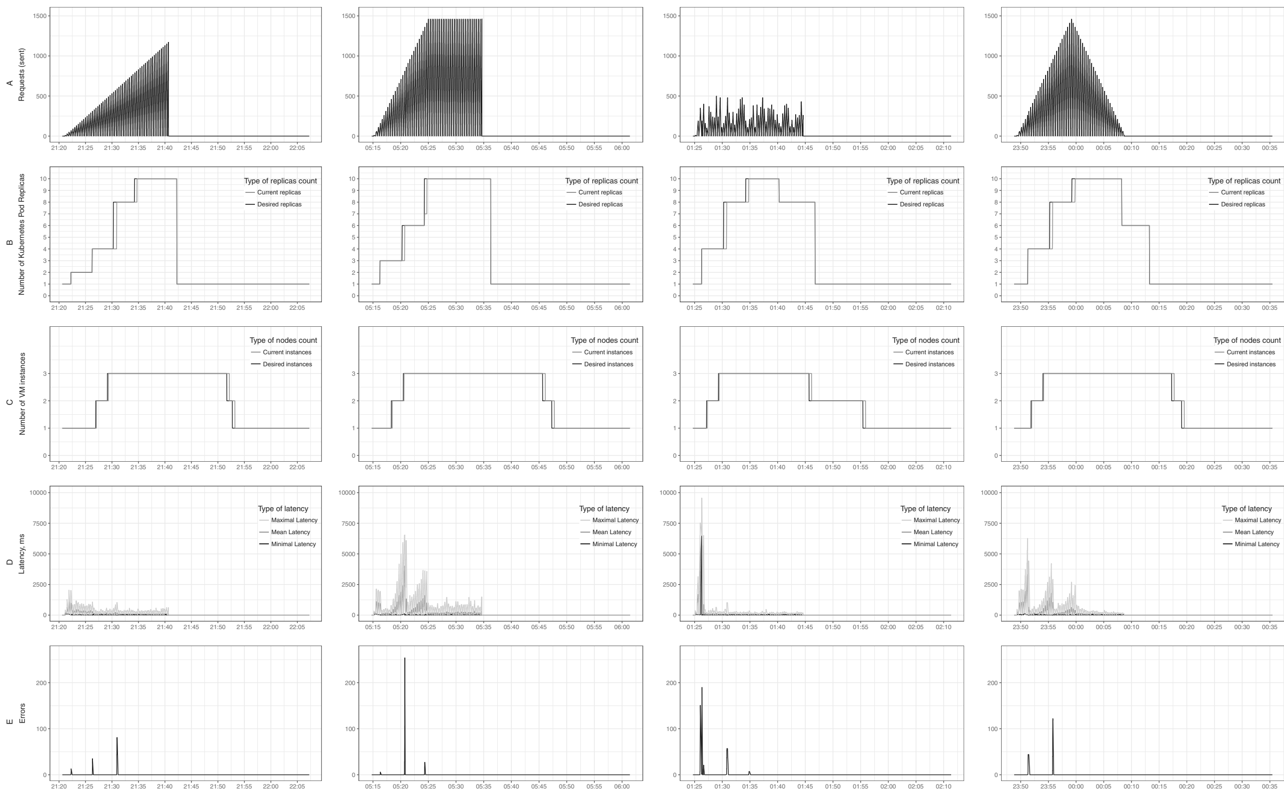

Fig. 9. Graphical representation of the GCE/Kubernetes multilayered autoscaling solution. Columns (load patterns): (first) linearly increasing, (second) linearly increasing and constant, (third) random, (fourth) triangle. Rows: (A) total number of requests sent, (B) DAI and CAI of Kubernetes pods, (C) DAI and CAI of GCE VM instances, (D) response time, (E) requests failure rate.

values of RTV and MFRV for the 2nd AWS scale-out interval for Linear Increase and Constant).

The results shown in Table 6 indicate that the performance issues in the autoscaling solutions can be the reason for QoS violations. Making autoscaling intervals smaller and loosening thresholds in the autoscaling rules do not necessarily help to increase the performance of the solution during autoscaling as the old infrastructure remains exposed to the arriving requests. Moreover, the time necessary for the application services to become available for the requests was not taken into account.

5.2.2. Evaluating the effect of the container image size and pulling policy on the scaling performance. Creation and termination of microservice replicas is a mechanism that enables scaling on the containerized application virtualization layer. With the differences in the types of software used inside the containers, the actual scaling time in the multilayered autoscaling case may differ as the additional time is required to receive the image that is used to create the container. This difference could be attributed to the size of the image as well as its actual location. For example, many users prefer to use the centralized DockerHub repository of images. It contains a vast number of container images with different software combinations. Therefore, one needs to take into account the time that will be required to get the container image over the network considering the latency. This issue is usually resolved by adding a local repository with the Docker container images. In this subsection of the paper we tried to make a point that the parameters of container images and pulling policies can be responsible for the changes in the scaling time of the containerized applications.

In order to prove the hypothesis that the pulling policy and the container image size should be considered when studying the scaling of the containerized application, a small-scale experiment was conducted on 11 container images taken from Docker Hub. During the experiment, the time between scheduling the container and its start was measured. The measurements were conducted multiple times for the *Always Pull* and *If Not Present* policies for container images supported by Kubernetes. The averaged results are provided in Table 7.

The pulling time in the conducted experiment does not give a clear indication of the connection between the pulling time when the image resides on Docker Hub and its size, though, for example, the average pulling time for

Table 6. Performance of the AWS, Azure, and GCE autoscaling solutions for scale-out events.

Load Pattern	CSP	Scale -out	Scale -out time [s]	RTV	MFRV
Linear Increase	AWS	1st	28.06	0.00	0.00
		2nd	9.03	0.00	0.00
	Azure	1st	**128.00**	0.00	0.00
		2nd	126.00	0.00	0.00
	GCE	1st	8.01	0.00	0.00
		2nd	12.01	0.00	0.00
Linear Increase and Constant	AWS	1st	17.03	0.00	0.00
		2nd	28.08	0.73	0.88
	Azure	1st	**128.00**	0.85	0.92
		2nd	123.00	**0.92**	**0.94**
	GCE	1st	26.02	0.00	0.00
		2nd	11.95	0.02	0.02
Random	AWS	1st	33.08	0.00	0.00
		2nd	15.01	0.00	0.00
	Azure	1st	**131.00**	0.00	0.00
		2nd	117.00	0.00	0.00
	GCE	1st	11.01	0.00	**1.00**
		2nd	7.99	0.00	0.00
Triangle	AWS	1st	6.01	0.00	0.00
		2nd	18.02	0.00	0.00
	Azure	1st	**155.00**	**0.86**	**0.91**
		2nd	128.00	0.00	0.00
	GCE	1st	7.98	0.00	0.00
		2nd	8.01	0.00	0.00

Table 7. Docker image pulling time for different pulling policies: image present and image on Docker Hub.

Image	Size [Mb]	Pulling time [s] (present)	Pulling time [s] (Docker Hub)
hello-world	0.00185	13	101
redis	27.8	12	66
nodejs-alpine	69.7	13	78
python-alpine	89.9	13	234
mongodb	368	13	98
mysql	445	12	55
java	584	14	300
nodejs	674	14	181
r	701	14	191
golang	715	14	272
python	912	14	183

six smallest images is more than twice larger than the average pulling time for five largest images. The outlier pulling times for the *hello-world*, *python-alpine* and *java* images may be caused by the popularity of these images and the limited capacity of Docker Hub. Though such a small experiment does not provide a clear support for the initial hypothesis, it highlights the fluctuations in the pulling time, which could be a reason for the slowdown on the containerized application virtualization layer. As expected, in case of the locally present image, the pulling time is significantly lower, staying in the interval between 12 and 14 seconds for the tested cases. A local container images repository may increase the performance of the multilayered scaling though the additional resources will be required to establish such a configuration.

6. Related works

Fundamental principles of the autoscaling policies performance evaluation were introduced by Papadopoulos *et al*. (2016). The researchers describe an autoscaling policy performance evaluation approach based on a chance constrained optimization problem solved using scenario theory. The approach was implemented in *Performance Evaluation Framework for Auto-Scaling (PEAS)* and tested on several existing autoscaling policies using 796 real workload traces. The paper also introduced several distinct metrics to evaluate the autoscaling performance with the core metrics of the average number of under- and over-provisioned resources.

The major contribution of the study by Ilyushkin *et al*. (2017) is a set of performance metrics to estimate an autoscaling policy. The set includes under- and over-provisioning accuracy, wrong-provisioning timeshare, instability, as well as other user-oriented metrics, e.g., wait time, response time, elastic slowdown, average number of resources, or average task throughput. Upon the listed metrics, the authors have built an approach to compare autoscalers using pairwise comparison, fractional difference comparison, and aggregated elasticity, as well as some user metrics. The developed approach and the metrics were applied to selected existing autoscaling policies. Although the presented approach and metrics allow comparing different autoscaling policies, even the authors admit that the type of performance considered also heavily relies on the type of application under consideration.

The technical report by Versluis (2017) highlights fundamental research questions regarding the performance of different autoscaling policies. This paper may be viewed as a comprehensive extension of the one by Ilyushkin *et al.* (2017). Using four different workloads from scientific, industrial, and engineering domains, the authors were able to prove that the application domain actually heavily influences the quality of the autoscaling results.

The performance estimation approach presented by Evangelidis *et al*. (2017) is based on probabilistic

discrete-time Markov chain model checking. The checking is conducted using the PRISM tool, which is a probabilistic model checker for the formal modeling and analysis of systems with random or probabilistic behavior. Each policy is encoded in the PRISM tool with a set of user-defined model parameters. By specifying the auto-scaling policy, the model parameters, and running PRISM, the user would be able to obtain the estimates of the probability that various estimated performance parameters are lying in specified intervals for different values of model parameters. The identified most appropriate values of model parameters could be used to adjust the autoscaling policy. The authors also conducted the validation of the study by using the AWS public cloud testbed and ROC analysis.

Hwang *et al.* (2016) outlined the generic performance model for clouds of any type. It encompasses a total of 19 metrics divided into 3 abstraction levels: basic performance metrics, cloud capabilities, cloud productivity. Multiple metrics were applied to comprehensively estimate the performance of scale-out, scale-up, and mixed scaling modes on some real-world benchmarks and on the public cloud providers.

The presented works concentrate on the evaluation of autoscaling on the level of virtual machines. These works also consider the evaluation of policies and not of the implementation of autoscaling in commercial clouds. Our approach extends the works towards multiple autoscaling layers and investigates the important aspect of overheads of real implementations.

7. Conclusion and future work

In this paper we tried to summarize our theoretical understanding of the autoscaling area and shed the light on performance evaluation for multilayered autoscaling solutions. The paper incorporates the methodology and tools originally presented by Jindal *et al.* (2017). The multilayered autoscaling performance evaluation methodology and ScaleX were applied in the scope of the paper to enable the comparison of several multilayered autoscaling solutions based on the virtual infrastructure provided by public CSPs.

The results of the conducted comparison show that for multilayered autoscaling the performance is not only determined by the time taken for autoscaling but rather also by the time that the decision to scale takes, by the real hardware underlying VM instances, and by the degree of synchronization between autoscaling on different virtualization layers. With the unadjusted autoscaling policies, the GCE/Kubernetes solution showed the best overall performance on the tested case which could mostly be attributed to the overprovisioning of VMs. Additionally, with another small experiment, the influence of the container image pulling time on the autoscaling quality was highlighted in the paper.

The study indicated several future work directions. The addition of support for the Randomized Multiple Interleaved Trials (RMITs) testing methodology (Abedi and Brecht, 2017) to ScaleX will increase the accuracy of the evaluations made using the tool. The support for automated autoscaling policies testing with the specialized metrics given by ScaleX (**?**) will extend the evaluation capabilities of the tool allowing evaluation of both autoscaling solutions and policies. By making load generation in ScaleX more flexible (generating the load based on data from logs and traces), the behaviour of autoscaling solutions could be studied in conditions close to industrial cases. Yet another necessary step would be the extension of ScaleX's functionality to support the evaluation of vertical autoscaling and autoscaling in hybrid cloud environments. With all these possible extensions, ScaleX may find use even in business environments willing to test scaling capabilities of the virtual infrastructure and the containerized application to find out the scaling bottlenecks and get rid of them. Addition of the recommending service to the evaluation tool may allow the user to receive insights from ScaleX on how autoscaling solutions should be set or which autoscaling policy should be selected.

By implementing these extensions in ScaleX and further polishing the evaluation methodology, we aim to continue testing autoscaling capabilities of the existing solutions in different settings. Such tests will enable us to better understand the complex area of elastic cloud applications. Nevertheless, the provided approach may become inappropriate for the serverless paradigm. Function-as-a-Service (FaaS) tries to hide the elasticity of the virtual infrastructure from the user highlighting only scaling on the application level by changing the number of function instances backed with the change in the number of running containers (Lloyd *et al.*, 2018). Adapting the autoscaling solution performance evaluation methodology to this paradigm may become another challenging research direction.

The next important step is to identify the infrastructure and application parameters that influence the quality of scaling. Yet another barely covered research problem is the influence of the application structure on the actual scaling capabilities of the application. The research in this area may uncover the particular microservice application structures that have a bigger scaling potential than the other applications.

8. Availability

The source code of ScaleX and the video manual are available in the Git repository at `https://github.com/ansjin/ScaleX`.

Acknowledgment

The authors would like to express their gratitude to the anonymous reviewers of the paper, who provided their valuable comments to improve the contribution. This work was supported by the German Research Foundation (DFG) and the Technical University of Munich within the funding programme *Open Access Publishing*.

References

Abedi, A. and Brecht, T. (2017). Conducting repeatable experiments in highly variable cloud computing environments, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE'17, L'Aquila, Italy*, pp. 287–292.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. and Merle, P. (2017). Autonomic vertical elasticity of docker containers with elasticdocker, *2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA*, pp. 472–479.

Bauer, A., Herbst, N. and Kounev, S. (2017). Design and evaluation of a proactive, application-aware auto-scaler: Tutorial paper, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE'17, L'Aquila, Italy*, pp. 425–428.

Bondi, A.B. (2000). Characteristics of scalability and their impact on performance, *Proceedings of the 2nd International Workshop on Software and Performance, WOSP'00, Ottawa, Canada*, pp. 195–203.

Evangelidis, A., Parker, D. and Bahsoon, R. (2017). Performance modelling and verification of cloud-based auto-scaling policies, *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid'17, Madrid, Spain*, pp. 355–364.

Guo, Y., Stolyar, A. and Walid, A. (2018). Online VM auto-scaling algorithms for application hosting in a cloud, *IEEE Transactions on Cloud Computing*, pp. 1–1, (early access), `https://ieeexplore.ieee.org/document/8351912`.

Herbst, N.R., Kounev, S. and Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not, *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), San Jose, CA, USA* , pp. 23–27.

Hwang, K., Bai, X., Shi, Y., Li, M., Chen, W.G. and Wu, Y. (2016). Cloud performance modeling with benchmark evaluation of elastic scaling strategies, *IEEE Transactions on Parallel and Distributed Systems* **27**(1): 130–143.

Ilyushkin, A., Ali-Eldin, A., Herbst, N., Papadopoulos, A.V., Ghit, B., Epema, D. and Iosup, A. (2017). An experimental performance evaluation of autoscaling policies for complex workflows, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE'17, L'Aquila, Italy*, pp. 75–86.

Jakobik, A., Grzonka, D. and Kolodziej, J. (2017). Security supportive energy aware scheduling and scaling for cloud environments, *European Conference on Modelling and Simulation, ECMS 2017, Budapest, Hungary,* pp. 583–590.

Jindal, A., Podolskiy, V. and Gerndt, M. (2017). Multilayered cloud applications autoscaling performance estimation, *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2), Kanazawa, Japan*, pp. 24–31.

Versluis, L. and Neacsu, A.I. (2017). A trace-based performance study of autoscaling workloads of workflows in datacenters, *Technical Report 1711.08993v1*, Vrije Universiteit Amsterdam, Amsterdam.

Liu, Y., Rameshan, N., Monte, E., Vlassov, V. and Navarro, L. (2015). Prorenata: Proactive and reactive tuning to scale a distributed storage system, *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzen, China*, pp. 453–464.

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance, *2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, USA*, pp. 159–169.

Moore, L.R., Bean, K. and Ellahi, T. (2013). Transforming reactive auto-scaling into proactive auto-scaling, *Proceedings of the 3rd International Workshop on Cloud Data and Platforms, CloudDP'13, Prague, Czech Republic*, pp. 7–12.

Nikravesh, A.Y., Ajila, S.A. and Lung, C.-H. (2015). Towards an autonomic auto-scaling prediction system for cloud resource provisioning, *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'15, Florence, Italy*, pp. 35–45.

Papadopoulos, A.V., Ali-Eldin, A., Arzen, K.-E., Tordsson, J. and Elmroth, E. (2016). PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications, *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* **1**(4): 15:1–15:31.

Roy, N., Dubey, A. and Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting, *2011 IEEE 4th International Conference on Cloud Computing, Washington, DC, USA*, pp. 500–507.

Sotomayor, B., Montero, R.S., Llorente, I.M. and Foster, I. (2009a). Resource leasing and the art of suspending virtual machines, *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications, HPCC'09, Seoul, South Korea,* pp. 59–68.

Sotomayor, B., Montero, R.S., Llorente, I.M. and Foster, I. (2009b). Virtual infrastructure management in private and hybrid clouds, *IEEE Internet Computing* **13**(5): 14–22.

Vladimir Podolskiy is a PhD student at TUM and a DAAD scholar. His research interests are in the area of predictive cloud applications, autoscaling, evaluation of autoscaling solutions and scalable middleware for the Internet of things. He graduated in 2014 from Bauman Moscow State Technical University (BMSTU), Russia. Prior to starting his PhD, he had worked for several years at IBS Group as an analyst and a software architect.

Anshul Jindal is a PhD student at TUM. His research interests include cloud computing, autoscaling and performance predictions of microservices. He completed his MSc in informatics in 2018 at TUM, Germany. Prior to starting his studies, he had worked for 2 years at Samsung Research Institute in Bangalore, India, as a senior software engineer.

Michael Gerndt received a PhD in computer science in 1989 from the University of Bonn. In 1990 and 1991, he held a postdoc position at the University of Vienna, and joined Julich Research Centre in 1992. He habilitated in 1998 at the Technical University of Munich (TUM). Since 2000 he has been a professor of architecture of parallel and distributed systems there. His research focuses on resources management in cloud environments and on programming models and tools for scalable parallel architectures.

Received: 18 July 2018
Revised: 12 December 2018
Accepted: 1 February 2019