Multilayered Cloud Applications Autoscaling Performance Estimation

Anshul Jindal*, Vladimir Podolskiy† and Michael Gerndt‡
Chair of Computer Architecture, *Technical University of Munich*
Garching (near Munich), Germany
Email: *anshul.jindal@tum.de, †v.podolskiy@tum.de, ‡gerndt@in.tum.de

***Abstract*—A multilayered autoscaling gets an increasing attention both in research and business communities. Introduction of new virtualization layers such as containers, pods, and clusters has turned a deployment and a management of cloud applications into a simple routine. Each virtualization layer usually provides its own solution for scaling. However, synchronization and collaboration of these solutions on multiple layers of virtualization remains an open topic.**

In the scope of the paper, we consider a wide research problem of the autoscaling across several layers for cloud applications. A novel approach to multilayered autoscalers performance measurement is introduced in this paper. This approach is implemented in Autoscaling Performance Measurement Tool (APMT), which architecture and functionality are also discussed. Results of model experiments on different requests patterns are also provided in the paper.

***Index Terms*—tool to estimate autoscaling performance on multiple layers, multilayered autoscaling, performance of autoscaling, cloud applications autoscaling, cloud**

I. Introduction

Cloud computing's success is based on the virtualization technology that considers hardware as a pool of resources to be provided to users in a form of virtual machines (VMs). While such an abstraction introduces an additional overhead for the indirect usage of hardware resources, it also helps businesses scale their applications in the cloud fast. Scalability of cloud applications is their major feature and the main reason behind the cloud computing wide spread. Most cloud services providers (CSPs) provide their users autoscaling services to both capture the flash crowd traffic and save money when additional machines are unneeded.

However, for some applications using microservice architecture, virtual machines could become inconvenient either because of coarse granularity or because of the microservices management inconvenience. Therefore, recent technological improvements in cloud computing are revolving around new levels of cloud abstraction and granularity. Such virtualization entities as containers, pods, and clusters extend a cloud platform from several viewpoints: the finer granularity of cloud application components management; the easier support of established computational paradigms and architectures; the self-containment and the abscence of dependencies on external programs. The use of such technologies makes cloud application highly flexible and easy to maintain. However, with the introduction of several virtualization layers, the cloud application architect should be aware of possible tradeoffs.

Additional layers of virtualization introduce a high level of flexibility and control over a cloud application. But these layers are also responsible for the major loss in the performance of multilayered cloud applications. Yet another major problem for multilayered cloud applications is their poor manageability as a whole.

Whereas each virtualization layer introduces layer-specific management tool, almost no synchronization exists between the layers. An attempt to extract advantages of each virtualization layer for a distributed cloud application with the microservices architecture would end up with an unmanageable solution that has a big potential to break itself [1].

The first step to synchronize autoscaling solutions is to research the real performance of autoscaling solutions combinations. The measured performance would lead to understanding the major architectural drawbacks both for individual autoscaling solutions and their combinations. The general notion of the scaling performance is the time needed to scale in response to the change in amount of requests. It is also possible to consider the overall quality of a service provided by the specific autoscaled cloud application in respect to scaling effects that are taking place. Both notions were laid as a foundation for the multilayered autoscaling performance measurement approach introduced in this paper.

The key contribution of the paper is the novel approach and a tool to solve the problem of the measurement of autoscaling solutions' performance. The presented approach and the tool's output data facilitate the research of autoscaling effects both for single-layered and multilayered autoscaling of an arbitrary application. A solution of the performance measurement problem for autoscaler combinations paves a path for understanding the conceptual and technical reasons behind certain drawbacks of autoscaler combinations and, in turn, gives an outlook on possible technical solutions to the multilayered cloud application autoscaling problem.

The developed tool, namely APMT, could also be used to check the rules that are set for autoscalers at different virtualization layers to increase the performance during the autoscaling and to decrease the time necessary to scale the cloud application. Testing of various autoscaling scenarios with different autoscaling rules could be a valuable use-case of the developed tool for the industry.

In the following section of the paper we introduce a background information on multilayred virtualization. Further, in

Section III, we discuss the related work on the multilayered virtualization and the autoscaling for multilayered cloud applications. Section IV discusses the multilayered autoscaling performance measurement approach implemented in APMT. In turn, APMT is described in detail in Section V from the architectural and functional viewpoints. Section VI focuses on experimental results that were acquired with APMT on multilayer autoscaling examples with different requests generation patterns. Finally, Section VII concludes the paper and introduces a direction for the future research.

II. Background

A. Autoscaling for Virtual Machines

Addition or removal of virtual machines is the most common way to conduct autoscaling. For an autoscaling to work, it must be combined with load balancing and monitoring.

Monitoring service is used to retrieve relevant metrics on which alarms and triggers can be defined to execute custom actions such as an increment (scale-out) or a decrement (scale-in) of the resource pool size based on certain conditions. Furthermore, both the increment and the decrement of the resource pool size should be hidden from the users. This is achieved by the load balancer that allows having a static single-entry point for the application.

Each CSP provides its own native autoscaler which could not be used alongside clouds of other CSPs. For the sake of brevity, we provide a short overview only for AWS autoscaler as we further use it for the experiments.

AWS (Amazon Web Services) Auto Scaling 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). ASG is a set of different Amazon Elastic Compute Cloud (EC2) instances sharing similar characteristics and subject to the same scaling policies. Therefore, every machine in the group has the same Amazon Machine Image (AMI) and the same hardware characteristics. AWS Auto Scaling helps to maintain application availability and allows to scale automatically according to conditions defined.

B. Autoscaling for Higher-Level Entities

We'll discuss autoscaling for higher-level entities on the example of Kubernetes as it is used in experiments.

Kubernetes is an orchestration tool which provides the means to support containerized deployment atop Platform-as-a-Service (PaaS) clouds, focusing specifically on cluster-based systems. Kubernetes can schedule and run application containers on clusters of physical or virtual machines. Kubernetes pods are the smallest deployable units of computing that can be created and managed in it. Kubernetes deploys multiple pods across physical machines or VMs, enabling the scaling of an application with the dynamically changing workload. This autoscaling process of Kubernetes is called a horizontal pod autoscaling (HPA). Each pod can support multiple containers, which are able to make use of services (e.g. file system and I/O) associated with a pod. With HPA, Kubernetes automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization.

In combination with autoscaling on VMs layer, it gives a two-layered autoscaling which could be used to increase the flexibility of industry cloud applications.

III. Related Work

Research in VMs autoscaling mainly focuses on the development of better autoscaling prediction models [2], [3]. The comparison between different autoscaling prediction models is provided in [4]. In [5], an autoscaling service that can take advantage of metrics provided by different levels of the cloud stack is discussed. This work presents a custom solution instead of using native autoscalers provided by CSPs, thus introducing an entry barrier for the industry already using existing autoscalers. A Polyglot autoscaling service monitors the application attached to it and resources to further adjust the application based on the autoscaling policies of the user and on the system conditions [13].

The research in containers autoscaling focuses mainly on the development of algorithms to improve the containers deployment [7], [10], [12]. A provision of the same QoS assurance with light virtualization environment of containers is investigated in [8]. Additionally, there exist CSP-independent container orchestration solutions as Kubernetes and Docker Swarm. Such solutions use the autoscaling to share the load among containers. The autoscaling functionality of these solutions is still under development.

There is also an ongoing research in the area of performance monitoring and estimation for clouds on different layers of virtualization. Results of cross-comparison of VM instances performance to the containers are provided in [9]. Performance of containers management tool Kubernetes is discussed in [6], however the autoscaling aspect remains uncovered. One of possible approaches to investigate the behaviour of autoscalers is proposed in [11]. The main idea of the approach is to model cloud autoscaling solutions using time series data. Such an approach imposes a heavy limitation on the autoscaling research as real-world load patterns could significantly differ from the modelled data.

To the best of our knowledge, the area of multilayered autoscaling performance measurement at the moment remains mostly uncovered in the literature. Moreover, the necessity of our work is also based on the absence of a tool to monitor the performance of autoscalers on different layers of virtualization for an arbitrary application under different requests patterns. A novel approach that is used as a foundation of the APMT also contributes to the novelty of the results provided in the following sections.

IV. Multilayered Autoscaling Performance Measurement Approach

A. Single-Layered Autoscaling Performance Measurement

As autoscaling's main purpose is to react to the change in the amount of requests, *the reaction time* or the autoscaling latency, i.e. the time between the decision of the autoscaler to

scale the resources and the final adaptation of the resource, is an important metric for the performance of the autoscaler. Additional metrics can be derived from the service's QoS requirements. These are the fraction of the autoscaler latency where QoS requirements were not matched. Let us first define autoscaling latency based on two metrics of the autoscaler: CAI and DAI.

In order to measure the autoscaling latency, we use two metrics provided by autoscalers: *Current Amount of Instances (CAI)* and *Desired Amount of Instances (DAI)*. Each autoscaler contains autoscaling rules that determine conditions triggering autoscaling actions (scale-in or scale-out). As the deployment of a VM (or another autoscaling entity) takes time for resource allocation, booting and configuring, DAI will differ from CAI for a certain time interval. After autoscaling is completed, CAI and DAI will be equal. So, we consider the described time interval to be the autoscaling latency. If t_{start} is the start time and t_{finish} is the finish time of autoscaling then the *autoscaling interval* $T_{autoscale} = [t_{start}, t_{finish}]$ is the interval such that $\forall t \in T_{autoscale} : CAI(t) \neq DAI(t)$. If $\forall t \in T_{autoscale}$ we see that $CAI(t) < DAI(t)$ then $T_{autoscale}$ is a *scale-out* interval. If $\forall t \in T_{autoscale}$ we have $CAI(t) > DAI(t)$ then $T_{autoscale}$ is a *scale-in* interval. $CAI = DAI$ represents ordinary functioning of the cloud application. The autoscaling latency of an autoscaling interval $T_{autoscale}$ is defined as $t_{finish} - t_{start}$.

Other performance measures for autoscaling solutions are based on the notion of the quality of service for the cloud application. Basically, each quality of service parameter depends on the performance that the cloud service demonstrates. However, instead of looking at all the QoS parameters, we have chosen two that directly get influenced by the autoscaler's performance. These parameters are - a *cloud service latency* and the *amount of failed requests*. These metrics have corresponding QoS limitations imposed on them by the cloud application users:

- Cloud application latency: should be below the given service latency threshold.
- Failure rate: should be below the given service failure rate threshold.

The definition of service latency is the time for an individual request and for failure rate is the fraction of the requests that failed, where failing is given, when the latency exceeds a certain failure-threshold. To measure the performance of the autoscaler, we now define the following two metrics: service latency violation and service failure rate violation. Both are the fraction of the autoscaling interval where the corresponding QoS requirement was violated.

As the performance metric for autoscaling based on the cloud service latency, we identify the fraction of the autoscaling interval $T_{autoscale}$ when the latency was above the predefined maximum. If $T_{autoscale} = [t_a, t_b]$ and $T_{highlatency} = [t_c, t_d]$ where $t_a \leq t_c \leq t_d \leq t_b$, then $\forall t \in T_{highlatency}$ we see that $latency(t) > latency_{required}$. Thus, this performance measure can be computed as:

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

The value of 1 for $HL(T_{autoscale})$ means that during the whole autoscaling interval, the requirement on the latency wasn't met. In more complex cases, $T_{highlatency}$ could be a set of intervals, e.g. $T_{highlatency} = \{T^{(1)}_{highlatency}, T^{(2)}_{highlatency}, ...T^{(p)}_{highlatency}\}$. Then it would be necessary to compute $T_{highlatency}$ as a sum of these intervals lengths.

Analogously to $HL(T_{autoscale})$, we compute the fraction of autoscaling interval during which the amount of failed requests was higher than a predefined value. So, if $T_{autoscale} = [t_a, t_b]$ and $T_{highfaults} = [t_e, t_f]$ where $t_a \leq t_e \leq t_f \leq t_b$, then $\forall t \in T_{highfaults}$ we see that $requests_{failed}(t) > limit_{failed}$ where $limit_{failed}$ is the highest amount of failed requests according to the service level agreement (SLA). In the simplest case, the performance measure is computed as:

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

Presented autoscaling performance metrics (1), (2) are simple yet efficient as they allow the direct estimation of the autoscaling quality in a form relevant both for research and business tasks.

All the above definitions like autoscaling latency could be extended to average values over several autoscaling intervals.

B. Multilayered Autoscaling Performance Measurement

A main research problem for multilayered autoscaling performance measurement is to identify, which set of scaling events on different layers to consider a single scaling case.

$T_{as} = \{T^{(1)}_1, T^{(2)}_1, ...T^{(n)}_1\}$ is a set of autoscaling intervals, whereas $T^{(1)}_1$ is an autoscaling interval on the level of scaling closest to the hardware (native CSP's autoscalers are on this level); superscript denotes the level. $T^{(i)}_2, i = 1 : n$ would be next autoscaling intervals on the same layers. An autoscaling interval $T^{(i)}_1$ is considered as an element of the set of autoscaling intervals for a single case of the multilayered autoscaling if and only if $\forall j : j < i$ we have the following conditions fulfilled:

$$T^{(i)}_1 \prec T^{(j)}_2 \tag{3}$$

$$T^{(i)}_1 \preceq T^{(j)}_1 \tag{4}$$

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

$\forall t_j \in T^{(j)}_2$ and $\forall t_i \in T^{(i)}_1$. Thus, in order to be considered a part of the single autoscaling an autoscaling on level i should 1) occur when all previous layers have entered a stable state, i.e. $CAI = DAI$, after the corresponding previous j^{th} autoscaling has already occured[1], and 2) be of the same

[1]In case of scale-out autoscaling we consider previous layers based on their distance to hardware, i.e. how many layers are between the current layer and the hardware. In the scale-in case, we consider the layer farthes from the hardware to be the first, so the enumeration starts at the topmost virtualization layer.

direction (scale-in or scale-out) as each of autoscalings on previous layers.

An example of the multilayered autoscaling case consisting of autoscaling intervals on different layers is presented in Fig. 1.

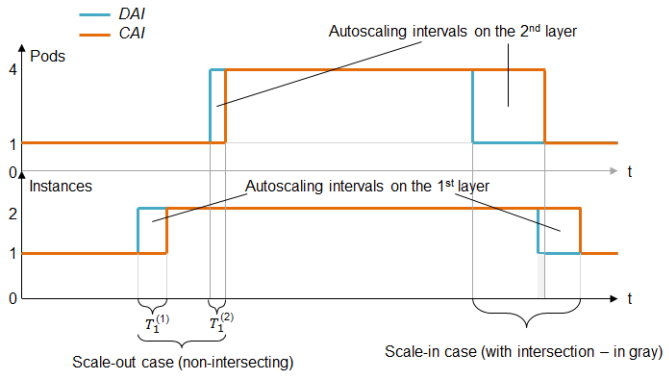

Fig. 1. Example of identification of multilayered autoscaling cases for two-layered architecture.

After we've defined the notion of the multilayered autoscaling, we will provide a formula to compute the autoscaling duration for a single autoscaling event of the multilayered cloud application. If we consider $A = \{a_1, a_2, ...a_m\}$ a set of indices that enumerates all the members of T_{as} then we can compute the duration of the multilayered autoscaling with the following formula:

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

Formula (6) takes into account a possible intersection of 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), (4), and (5), gives us an estimate for the duration of the single autoscaling event for an arbitrary multilayered cloud application.

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

$$T_{autoscale} = \bigcup_{i=1}^{m} T_1^{(a_i)} \quad (7)$$

The presented multilayered autoscaling performance measurement approach is implemented in Autoscaling Performance Measurement Tool which is discussed in the following section.

V. Autoscaling Performance Measurement Tool

A. Autoscaling Performance Measurement Tool Overview

APMT has a user-friendly web-interface. The tool is developed mostly in Node.js. An overall architecture of the APMT and communications between components thereof in a typical use case is shown in Fig. 2. APMT is divided into back-end and front-end layers. Each layer consists of components implementing micro-services architecture; these components can be scaled in case of changing requirements.

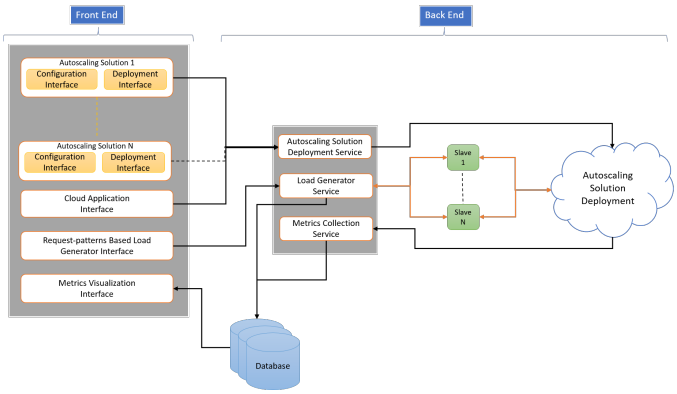

Fig. 2. Autoscaling Performance Measurement Tool Simplified Architecture.

Currently, APMT integrates three autoscaling options, namely, AWS Auto Scaling, Kubernetes horizontal pod scaling, and the combination of these two autoscalers. Each option has its own configuration and deployment strategy. In further subsections we present two layers of the developed tool.

B. Front-End Layer

Front-end layer represents a part of APMT designated to interact with user and provide him or her with an opportunity to select different options. Front-end comprises five components with user interfaces. In the following paragraphs we provide detailed description of these components.

Configuration Interface. The Configuration Interface provides user with the autoscaler configuration functionality. This component deploys and terminates autoscalers automatically using different user configurations. Each autoscaling option provided by the Configuration Interface support a number of configuration parameters.

AWS Auto Scaling allows to scale the number of VM instances based on the configurable parameters: type of instance, min. and max. number of instances, scaling decision metric (with its threshold) and autoscaling policy.

Kubernetes Horizontal Pod Autoscaling scales the number of pods whereas the number of VMs remains fixed in the cluster. As of now, only the CPU utilization is supported as the autoscaling decision metric in Kubernetes. For this autoscaling option there are also a number of parameters to set: type of instance, number of instances to include in a cluster, scaling decision metric (with its threshold).

Combined autoscaling combines previously described autoscaling options. It comprises the combined parameters of these aforementioned autoscaling options.

Deployment Interface. This interface allows the selected autoscaling option to be automatically deployed on the CSP's cloud using its native instances types. A single-click functionality to undeploy all instances is also provided.

Cloud Application Interface. Cloud Application Interface enables the selection from a list of different types of applications[2]. Supported pre-defined applications categories:

- CPU intensive Applications.
- I/O Applications.
- High Memory Usage Applications.

User can select an application in any of these categories to check the performance of the autoscaling solution on a particular autoscaling decision metric. At the moment, the tool implementation supports only CPU intensive applications.

Request Patterns-based Load Generator Interface. This component provides an interface to the workload request generator integrated in the APMT. It allows user to select any of the pre-configured load patterns in order to test the performance of the specific autoscaling option. At the moment, only four load patterns are supported by APMT.

Linear Increase Load Pattern represents linearly increasing number of requests per second in the scope of the load test time. *Linear Increase and Constant Load Pattern* represents linearly increasing number of requests per second which becomes constant in the second half of the load test time. *Random Load Pattern* represents randomly increasing and decreasing number of requests per second in the scope of the load test time. *Triangle Load Pattern* represents linearly increasing number of requests per second which then linearly decreases with the same speed in the second half of the load test time thus forming a triangle-like shape on a graph.

In the scope of the component, user can also configure the load pattern setting a number of options: number of concurrent clients, maximal number of requests, maximal duration of a test, request timeout, HTTP request method, request body, content type, a number of requests per second.

Metrics Visualization Interface. This interface shows graphs and tables for all the performance metrics. User can look at these graphs and tables to compare autoscaling options and choose the best one.

C. Back-End Layer

Back-end layer comprises components tightly integrated with the front-end components to provide a seamless user experience. In the following paragraphs we describe their functionality and structure in more details.

Autoscaling Solution Deployment Service. This service is linked with Configuration Interface, Deployment Interface and Cloud Application Interface components of the front-end layer. The purpose of this service is to combine the configuration parameters with the selected application and further to deploy it using the chosen autoscaling option. As a part of deployment, a monitoring service is also attached to collect the performance metrics and store this data in a database. Each autoscaler has different deployment procedure. In the following paragraphs we provide a brief information on each autoscaler's deployment procedure.

[2]This component also supports arbitrary cloud application if necessary for testing.

AWS Auto Scaling. An Elastic Load Balancer (ELB) and an Auto Scaling Group (ASG) are created in AWS Cloud using selected parameters. Afterwards, scaling parameters and polices are added to the ASG, and the chosen application is deployed on the created VMs. AWS Cloud Watch is used to collect the metrics data for the whole ASG as well as for the individual EC2 instances. Fig. 3 depicts the AWS Auto Scaling option deployment architecture.

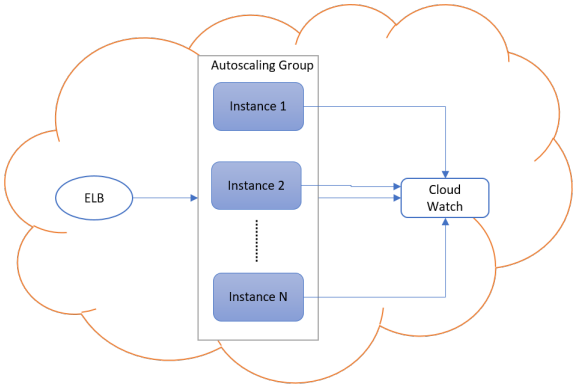

Fig. 3. Deployment of the AWS autoscaler.

Kubernetes Horizontal Pod Scaling. For the deployment of a Kubernetes cluster, APMT uses **kubeadm**[3]. The selected parameters are used to create the Kubernetes cluster in the AWS cloud using the EC2 instances. The cluster consists of a master node and a certain number of minion nodes (node refers to a VM instance). For collection of the metrics, APMT uses **Heapster**[4] coupled with the **InfluxDb**[5] used to store metrics in the form of time series natively. Metrics data is continuously fetched by APMT and stored in the database. Both kubeadm and Heapster are deployed in the scope of the cluster alongside the chosen application. Fig 4 depicts the Kubernetes horizontal pod autoscaler deployment in the cloud.

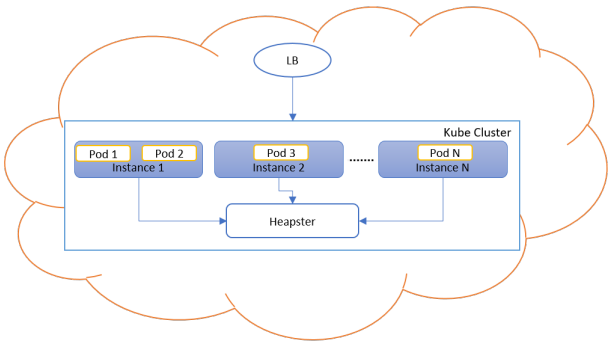

Fig. 4. Deployment of the Kubernetes horizontal pod autoscaler.

Combined autoscaling. This option provides the combined version of the discussed autoscalers. Firstly, as Kubernetes cluster requires a master node to be started before starting

[3]https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/
[4]https://github.com/kubernetes/heapster
[5]https://www.influxdata.com/time-series-platform/influxdb/

the minion nodes, we change the AWS ASG capacity to a single instance. This single instance serves as a master node for the Kubernetes cluster. This change enables us that no other instances will be started until the master node is ready. Once the master node is ready, we update the ASG configuration with user-provided parameters values. This change makes AWS autoscaler check whether any instance needs to be added to the group; to each new instance a launch configuration is attached. This launch configuration includes an instance start script which has all the configuration and commands required by the VM instance to join the Kubernetes cluster. Hence when a new instance has to be added to ASG, it automatically joins the Kubernetes cluster. In the scale-in phase, the VM instance to be terminated is drained safely by the master node from the Kubernetes cluster so that no further pods are scheduled on it and the pods already running on it are scheduled on other nodes. As there is a chance of the master node getting terminated as a part of ASG, APMT uses AWS instance protection and instance termination polices to protect the master node from termination. Furthermore, we have used ELB to direct the load to ASG which internally gets distributed to Kubernetes pods. For collecting the metrics, we have used both the AWS Cloud Watch (to get the metrics for ASG and EC2 instances) and the Heapster service(to get the metrics about the Kubernetes cluster). These metrics are continuously fetched by APMT and stored in the database. Fig. 5 depicts the combined autoscalers deployment in the cloud.

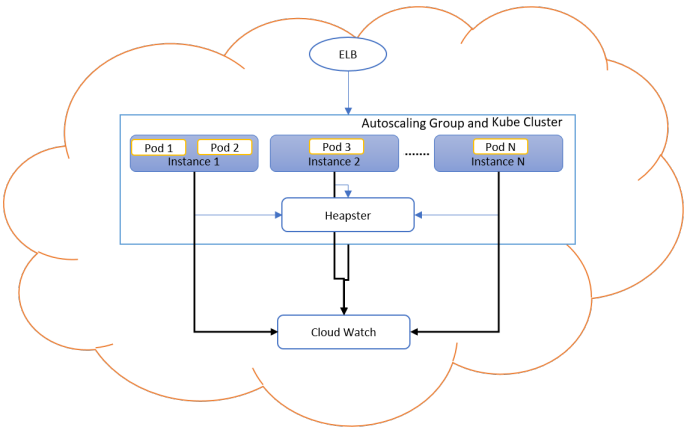

Fig. 5. Combined autoscalers deployment.

Load Generator Service. This service generates the desired amount of requests sent to the deployed application IP-address based on the workload pattern selected by the user. Load Generator Service is basically the customized version of Node.js based **Loadtest**[6]. For a selected load pattern, a certain number of clients and a load generation time are configured. A single load generation node does not become the bottleneck if the amount of requests to be generated is too high. After completion of each request, performance parameters are stored in the database for the purposes of visualization and analysis.

[6]https://www.npmjs.com/package/loadtest

Fig 6 depicts the overall architecture of the Load Generator Service.

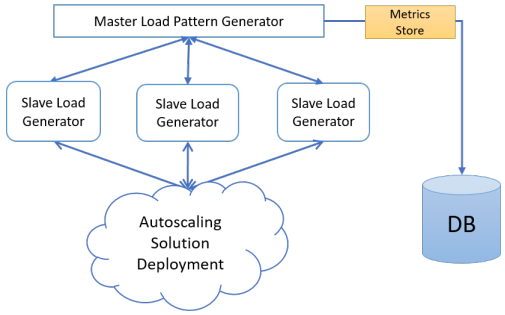

Fig. 6. Load Generator Service architecture.

Metrics Collection Service. This service periodically fetches the data from different monitoring services deployed as part of autoscalers and stores them in the database for further performance analysis.

NoSQL Database. APMT uses MongoDB to store the performance data and whereas MongoDB demonstrates the preference towards high insert rate over the transaction safety. Data schemes differ across different autoscaling options therefore MongoDB being schema-less database makes it an acceptable choice.

VI. Experimental Results

A. Experimental Setting

In our experiments, we use four different workload patterns linear increase, linear increase further becoming a constant, random increase/decrease, and linear increase followed by linear decrease forming a triangle-shaped load. The total time for each test was **20** minutes; request timeout is **6.5** seconds. Additional configuration for each load pattern is described in Table I.

TABLE I
Experimental Configuration: Load Generator Service

Pattern type	Requests per second	Concurrent clients
Linear Increase	Starts with 1, increases by 3 every second.	50
Linear Increase and Constant	Starts with 1, increases by 3 every second, remains constant after half of the test time has passed.	50
Random	Starts with 50, either increases or decreases randomly every second	50
Triangle	Starts with 1, increases by 3 every second, decreases by 3 every second after half of the test time has passed.	50

A test application computes the sum of prime numbers starting with **1** and up to **1000000** when called using a

particular API call. Executing this computation from multiple clients will increase the CPU utilization. Hence we can see the autoscaling option effect on the deployment.

Each autoscaler is deployed on the AWS cloud with the VM configuration mentioned in the Table II.

TABLE II
EXPERIMENTAL VM CONFIGURATION

Instance type	Memory	vCPUs
t2.micro	1 GB	1 vCPU

Table III depicts the configuration of Kubernetes autoscaler whereas Table IV contains AWS autoscaler configuration. For the combined case, we APMT uses configurations from both tables.

TABLE III
EXPERIMENTAL CONFIGURATION: KUBERNETES AUTOSCALER

Instances	Min. pods	Max. pods	Scaling metric	*Threshold*
2	1	10	CPU Utilization	10 %

TABLE IV
EXPERIMENTAL CONFIGURATION: AWS AUTOSCALER

Min. instances	Max. instances	Scaling metric	*Threshold*
1	3	CPU Utilization	10 %

B. Autoscaling Performance Measurements

The main results of the analysis of the data acquired by APMT for the combination of AWS native autoscaler with Kubernetes pods cluster resizing on the test case are provided in the Table V. For the combined case, we have used the presented approach to identify connected autoscaling intervals and compute the scaling times. To compute autoscaling performance metrics presented in the table we have used the following QoS requirements: mean latency should be no more than **2.5** seconds, errors rate should not be more than **10** errors per discretion interval. In principle, these results prove the possibility to use the provided approach and the tool to measure the performance of autoscaling. In the following table, HL represents the fraction of an autoscaling interval with a high mean latency, whereas FR stands for the fraction of an autoscaling interval with a high failed requests rate. Both HL and FR were computed for the multilayered scale-out case as each pattern contains single multilayered scale-out case and latency and errors metrics are collected during the scale-out autoscaling cases only.

For the sake of brevity, full experimental results with performance data (latencies and errors) are presented graphically in Fig. 7. On plots **F**, **L**, **R**, **X** request failures are caused by the inability of instances to cope with the growing request numbers considering the limitation of **6.5** seconds to wait for processing of the single request.

TABLE V
EXPERIMENTAL RESULTS FOR TWO-LAYERED AUTOSCALING (AWS + KUBERNETES)

Layer	Load Pattern	Scale-Out Time, *seconds*	Scale-In Time, *seconds*	HL	FR
AWS	Linear Increase	15.9	375.6	0.00	0.45
	Linear Increase and Constant	4.0	376.2	0.01	0.01
	Random	10.0	379.7	0.00	0.01
	Triangle	11.9	379.2	0.02	0.05
Kubernetes	Linear Increase	30.0	30.0	0.00	0.00
	Linear Increase and Constant	30.0	30.0	0.66	0.68
	Random	30.0	30.0	0.38	0.62
	Triangle	30.0	30.0	0.47	0.47
Combined	Linear Increase	76	412.6	0.00	0.13
	Linear Increase and Constant	34.0	409.2	0.65	0.70
	Random	40.0	-	0.00	0.46
	Triangle	41.9	400.3	0.05	0.15

On the example of linearly increasing requests generation pattern (plots **A**, **B**, **C**, **D**, **E**, **F**), we can notice a peak latency on plot **E** during the first scale-out of instances (VMs layer, see plot **D**). After this peak, latency rapidly decreases and stays on the low level during the rest request generation time. Moreover, failures peak from on the plot **F** resembles the latency peak on plot **E**. This illustrates the low QoS during the autoscaling on the level of VMs. From this and other requests generation patterns we can see that scaling of pods has almost no influence on performance what could be explained as pods increase and decrease not representing an actual change in hardware resources, so pods scaling is not critical for the experimental application's performance.

Plots for other requests generation patterns depict more or less the same behaviour of autoscalers. It is important to note that the scale-in time for AWS is larger than its scale-out time. This behavour could be explained by the number of activities that AWS executes during undeployment of an instance. Nevertheless, this pattern will be investigated in the future paper. States of pods readiness is yet another point for future investigation as it may lead to adjustments in the presented approach.

VII. CONCLUSIONS

Synchronization challenges and performance issues occur during the scaling of multilayered cloud applications. In some cases interference of an autoscaler on another level leads to the loss of the functionality. Therefore it is important to capture performance and functionality issues and track down architectural reasons for this. The aproach and the tool proposed in the paper serve to solve the problem of the autoscaling performance measurement both for single- and multilayered cloud applications.

The presented approach and the tool could be applied in the cloud-powered industry. A key example would be the APMT use to check the autoscaling rules at different virtualization

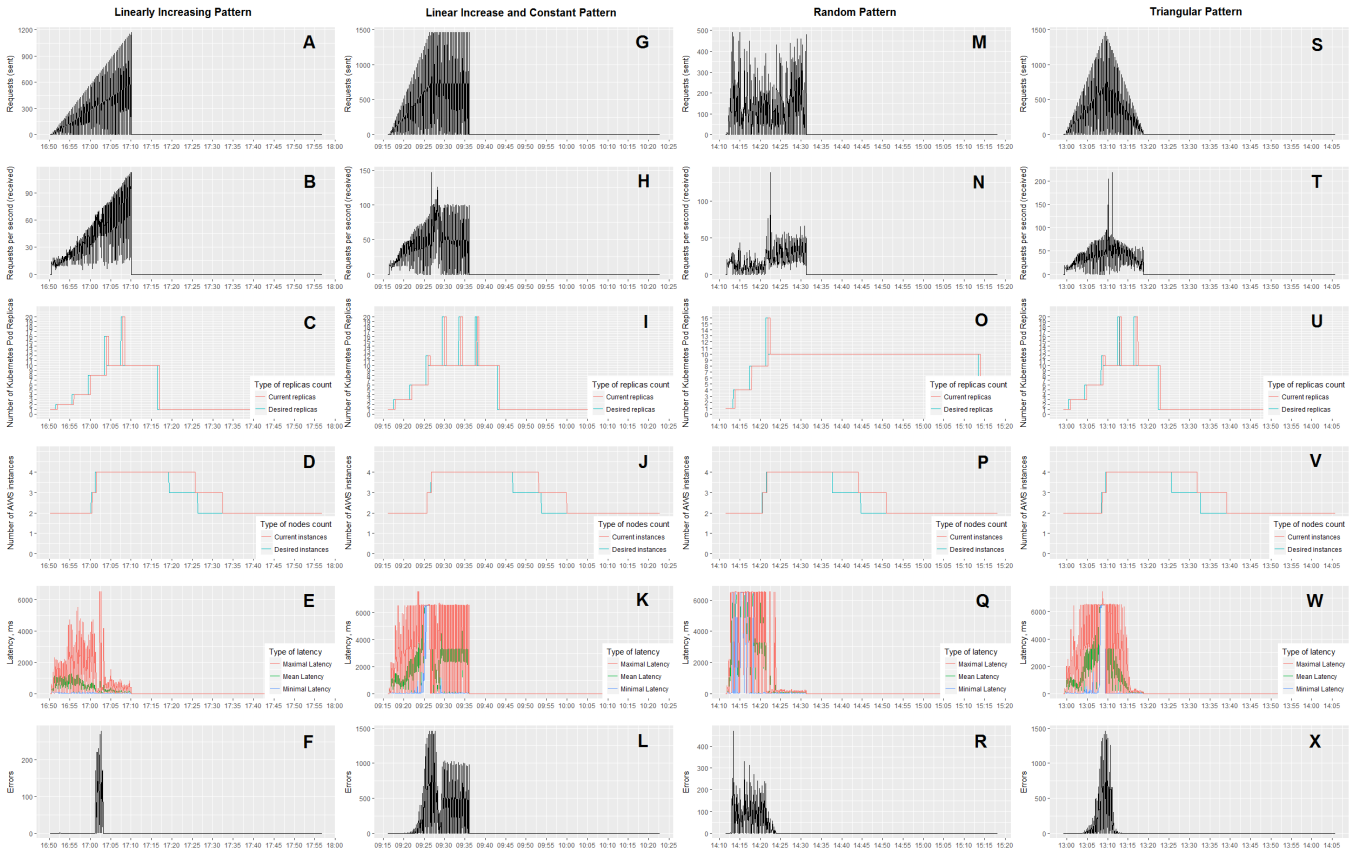

Fig. 7. Example of autoscaling time and performance graphs for combination of AWS native autoscaling with Kubernetes pods clusters resizing. Each graphs column corresponds to one of the foar load patterns: linearly increasing, linearly increasing with constant level, random, triangle.

layers in order to derive the best configuration of autoscaling rules. This could be done automatically using APMT on a first step to tune the scaling rules subject to optimization goals.

In the scope of the further research we plan to conduct a comprehensive estimation of existing autoscaling solutions and their combinations using the developed approach and APMT. As a result, we aim to acquire highly performant recipes to use autoscaling solutions and policies and combinations thereof for multilayered cloud applications.

REFERENCES

[1] M. Tighe and M. Bauer, "Integrating cloud application autoscaling with dynamic VM allocation," 2014 IEEE Network Operations and Management Symposium (NOMS), Krakow, 2014, pp. 1-9.

[2] N. Roy, A. Dubey and A. Gokhale, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," 2011 IEEE 4th International Conference on Cloud Computing, Washington, DC, 2011, pp. 500-507.

[3] T. Thepparat, A. Harnprasarnkit, D. Thippayawong, V. Boonjing and P. Chanvarasuth, "A Virtualization Approach to Auto-Scaling Problem," 2011 Eighth International Conference on Information Technology: New Generations, Las Vegas, NV, 2011, pp. 169-173.

[4] Yazhou Hu, Bo Deng and Fuyang Peng, "Autoscaling prediction models for cloud resource provisioning," 2016 2nd IEEE International Conference on Computer and Communications (ICCC), Chengdu, 2016, pp. 1364-1369.

[5] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde and A. Srivastava, "A Pluggable Autoscaling Service for Open Cloud PaaS Systems," 2012 IEEE Fifth International Conference on Utility and Cloud Computing, Chicago, IL, 2012, pp. 191-194.

[6] V. Medel, O. Rana, J. . Baares and U. Arronategui, "Modelling Performance & Resource Management in Kubernetes," 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), Shanghai, 2016, pp. 257-262.

[7] C. Pahl, "Containerization and the PaaS Cloud," in IEEE Cloud Computing, vol. 2, no. 3, pp. 24-31, May-June 2015.

[8] P. Heidari, Y. Lemieux and A. Shami, "QoS Assurance with Light Virtualization - A Survey," 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg City, 2016, pp. 558-563.

[9] A. M. Joy, "Performance comparison between Linux containers and virtual machines," 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, 2015, pp. 342-346.

[10] N. Bila, P. Dettori, A. Kanso, Y. Watanabe and A. Youssef, "Leveraging the Serverless Architecture for Securing Linux Containers," 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), Atlanta, GA, USA, 2017, pp. 401-404.

[11] M. N. A. H. Khan, Y. Liu, H. Alipour and S. Singh, "Modeling the Autoscaling Operations in Cloud with Time Series Data," 2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW), Montreal, QC, 2015, pp. 7-12.

[12] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," in IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, Sept. 2014.

[13] S. R. Seelam, P. Dettori, P. Westerink and B. B. Yang, "Polyglot Application Auto Scaling Service for Platform as a Service Cloud," 2015 IEEE International Conference on Cloud Engineering, Tempe, AZ, 2015, pp. 84-91