A Simulation-based Comparison between Industrial Autoscaling Solutions and *COCOS* for Cloud Applications*

Luciano Baresi, Giovanni Quattrocchi
Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milan, Italy
{*name.surname*}*@polimi.it*

***Abstract*—Dynamic resource allocation is the mechanism that allows one to change the resources associated with applications at runtime and match their actual needs. The autoscaling solutions offered by cloud infrastructures are probably the most widely-used incarnation of this concepts. Originally conceived to manage virtual machines according to user-defined rules, they are now much more sophisticated and can also allocate containers (lighter than virtual machines).**

This paper surveys the autoscaling solutions provided by the major cloud vendors and analyzes the services they provide. It also compares them against the solution we developed, called *COCOS autoscaling*. We simulated the different proposals and fed them with diverse workloads. Obtained results show that *COCOS autoscaling* outperforms its competitors in most of the cases: it optimizes resource allocation and keeps applications' response times under set thresholds.

***Keywords*-autoscaling, elastic computing, cloud computing, containers, control theory**

I. Introduction

Software systems are becoming more and more complex, are often required to handle very diverse workloads, and must keep agreed qualities of service. These requirements call for componentized architectures to both ease the replication of system parts and support their distribution [1]. Scalability is thus a prominent feature and computing resources should be allocated on demand. Ideally, provisioned resources should match the intensity of to-be-served workloads. Fewer allocated resources (under-provisioning) imply a degraded quality of service since they are not enough to cope with the actual needs. More resources than those required (over-provisioning) would not produce any visible effect on the quality of service with a clear waste of money [2].

Nowadays, cloud computing probably offers the best support to the idea of dynamic resource allocation and provides means to easily change allocated resources up to a theoretically infinite amount [3]. These means are called *autoscaling* capabilities.

Traditionally, autoscaling solutions were only able to resize a cluster of virtual machines (VM) by using simple user-written rules. VMs are heavyweight, slow to boot (since they contain a full-fledged operating system), and their scalability is limited to the addition/removal of entire machines (horizontal scalability). This impacts significantly the effectiveness of autoscaling since it imposes high latency and thus can only accommodate sub-optimal resource allocations.

The success of containers [4], a lightweight virtualization solution, offered new opportunities. Compared to VMs, containers are faster to boot and scale since they operate on a shared operating system. They can then be replicated quickly (horizontal scalability), but one can also change provisioned resources (i.e., they can be reconfigured[1]) at runtime (vertical scalability) in hundreds of milliseconds. Even if containers can run on physical machines directly, they are usually executed on top of VMs to better exploit their internal resources. Traditionally VMs hosted single components; containers allow them to run multiple isolated components at the same time without significant overhead [5]. Containers then allow for a faster and finer-grained resource allocation than VMs, they can better handle fast changing and fluctuating workloads, but they impose more sophisticated autoscaling solutions.

Public cloud providers offer several VM- and container-related services that implement sophisticated autoscaling solutions. Users can rent VMs and manually run containers on them, or they can run containers directly without accessing the underlying VM-based infrastructure (Containers-as-a-Service). Cloud providers also offer means to ease the deployment, management, and autoscaling of containers through dedicated orchestrators (such as Kubernetes[2]).

Given the many, diverse existing solutions, this paper surveys the autoscaling mechanisms of three of the most widely-used cloud providers: Google Cloud Platform, Amazon Web Services, and Microsoft Azure. We describe the infrastructural services along with their autoscaling mechanisms: scaling operations, time needed, and costs.

The paper also introduces *COCOS autoscaling*, our autoscaling solution [6] and compares it against the

*This work has been partially supported by the European project SODALITE (grant agreement 825480) and by the national research project SISMA (MIUR, PRIN 2017, Contract $201752ENYB$)

[1]Note that VMs cannot be reconfigured at runtime easily. To the best of our knowledge, Google and Amazon Web Services do not offer the feature. Microsoft Azure supports it, but it requires that given a set of VMs, changes must be applied to all of them and at the same time.

[2]https://kubernetes.io/

978-1-7281-8786-0/20/$31.00 ©2020 IEEE
DOI 10.1109/ICWS49710.2020.00020

above-mentioned mechanisms. *COCOS autoscaling* exploits control-theory to scale containerized applications at runtime. It aims to optimize allocated resources and keep the response time under a given threshold for different application types: microservices, big-data batch applications, and GPU-empowered machine learning applications. In addition, *COCOS autoscaling* provides a control loop that is at least one order of magnitude faster then its industry competitors (i.e., control period equals to 1 second).

The paper proposes a comparison of *COCOS autoscaling* against the relevant autoscaling mechanisms by means of *RAS* (Resource Allocation Simulator), a new lightweight simulation environment. Obtained results show that in most of the cases *COCOS autoscaling* outperforms its competitors both in terms of saved resources and number of violated execution times.

The rest of the paper is organized as follows. Section II discusses what the three aforementioned industrial frameworks offer. Section III introduces *COCOS autoscaling* and its main differences with respect to what proposed by the main cloud providers. Section IV describes the simulator we developed and reports on the experimentation we conducted to assess the benefits of our solution. Section V surveys related work, and Section VI concludes the paper.

II. Public Cloud Providers

This section focuses on Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure. For each platform, we concentrate on how they support autoscaling and fluctuating workloads. Table I shows the execution times of the different actions that manage VMs and containers; measurements are reported as averages among five repetitions. Note that we used "similar" VMs on the different platforms, and when we say *All*, obtained values were very similar and thus a single, average, value is shown.

A. Google Cloud Platform

Google provides two main cloud infrastructural services: *Google Compute* (GC) and *Google Kubernetes Engine* (GKE). GC provides means to rent and manage virtual machines; GKE focuses on containers.

We measured the time GC takes to start and terminate a Linux n1-standard-2 VM . The machine is equipped with 2 CPU cores and 7.5GB of RAM. It was started (terminated) in 11 (16) seconds. Among the three considered providers, this result is by far the best one. With respect to a past survey [7], the start-up time obtained by GC is 62% faster than all reported solutions. After the first minute, paid anyway, GC bills resources per second according to the type and amount of resources provisioned. Even if predefined machine types exist, GC allows users to create customized VMs with different configurations. This configuration cannot be changed at runtime (vertical scalability).

Provider	Service	Action	T [s]
GCP	Compute	Start VM	16
GCP	Compute	Terminate VM	11
GCP	GKE	Start Cluster+Nodes	234
GCP	GKE	Scale-out Cluster	79
GCP	GKE	Scale-in Cluster	243
AWS	EC2	Start VM	69
AWS	EC2	Terminate VM	65
AWS	Fargate/ECS	Start Service	101
AWS	Fargate/ECS	Scale-out Service	53
AWS	Fargate/EC2	Scale-in Service	355
AWS	ECS	Start Cluster	79
AWS	ECS	Terminate Cluster	123
AWS	ECS	Scale-out Cluster	101
AWS	ECS	Scale-in Cluster	128
AWS	EKS	Start Cluster	698
AWS	EKS	Start Node Group	145
AWS	EKS	Scale-out Cluster	138
AWS	EKS	Scale-in Cluster	108
Azure	Compute	Start VM	112
Azure	Compute	Terminate VM	87
Azure	AKS	Start Cluster	212
Azure	AKS	Terminate Cluster	320
Azure	AKS	Scale-out Cluster	134
Azure	AKS	Scale-in Cluster	251
All	Kubernetes	Start Pod/Service	2
All	Kubernetes	Terminate Pod /Service	1
All	Kubernetes	HPA Scale-out Service	4
All	Kubernetes	HPA Scale-in Service	3
All	Container on VM	Run container	1
All	Container on VM	Terminate container	0.5
All	Container on VM	Vertically Scale container	0.2

Table I: Execution times of VM/container operations.

GC offers a single autoscaling mechanism (GC Autoscaler) whose algorithm is not fully disclosed. To use it, one must define a metric (i.e., CPU utilization or custom metrics) and a desired/target value. The autoscaler automatically adjusts the number of VMs to meet the target value without human intervention. After adding a VM (scale-out action), a cool down period must elapse before executing another scale-out action (default is 60 seconds). Moreover, before scaling in, that is, before removing a VM, one must wait for 10 minutes (stabilization period) to be sure the system is ready to safely release resources.

GCP also offers a hosted version of Kubernetes, its popular, open-source, container orchestrator that exploits a master-slave architecture. Kubernetes manages *pods*, groups of containers that are deployed and managed all together. Table I shows that starting and terminating a pod running an instance of NGINX[3], a popular application platform for microservices, is very fast: 2 and 1 seconds respectively.

Kubernetes runs on a cluster of VMs and offers a cluster-level autoscaling system that scales up or down the number of cluster slaves with respect to the resources pods need. We measured that GKE needed 234 seconds to create a cluster[4] of three VMs (one master, two slaves), and 79 and 243 seconds, respectively, to add or remove a VM. This shows a significant overhead compared to what we obtained with

[3]https://www.nginx.com

[4]We wanted to have a small-enough system, but it also had to be meaningful: one master and two slave VMs meet this requirement.

GC and plain VMs.

In Kubernetes, a *service* is an abstraction that connects instances of the same pod type and adds load balancing and, optionally, autoscaling. A *Controller* is a component that allows for different types of control actions. For example, *Replica Set* guarantees that the number of pod instances is kept constant during service lifecycle, while *Horizontal Pod Autoscaler* (HPA) can change the desired amount of replicas of a *Replica Set*. Given a metric and a user-defined target value (tm), the goal of HPA is to compute the new amount of pod replicas (tr) to meet the target value. To do that, HPA scales the current number of pod replicas (cr) proportionally to the ratio between measured (cm) and target metric values: $tr = ceil(cr * cm/tm)$. The new value (allocation) is only enacted if the ratio cm/tm is greater than set tolerance (default value is 0.1). By default, the computation is carried out every 30 seconds (control period) and a 5-minute *stabilization period* is used when multiple scale-in actions are planned. Table I reports that the scale-out/scale-in of a Kubernetes service requires 4 and 3 seconds, respectively.

Kubernetes also provides *Vertical Pod Autoscaler* (VPA): it works on a Replica Set and changes the configuration (CPU and memory) of the respective pod instances. At the time of writing, VPA is still limited and does not provide real vertical scalability because it requires that pods be restarted to change their configurations and the scaling actions are carried out on the whole Replica Set and not on single pods independently. In addition, the integration with HPA and JVM-based containers is not fully supported.

Kubernetes Engine bills a fixed amount per hour per cluster (set of VMs) it manages, in addition to the cost of the VMs themselves.

B. Amazon Web Services

Amazon provides diverse services for running applications on the cloud. The most basic one is *Elastic Compute Cloud* (EC2), which allows one to create different types of VMs by starting from predefined or custom VM images. We measured the time EC2 takes to start/terminate a Linux VM called *t3.medium*: it provides 2 CPU cores and 4GB of RAM. Table I shows EC2 takes on average 69 (65) seconds to create (terminate) the VM. This service is billed per second (after the first minute, paid anyway), and the cost depends on the VM type. Containers can be run atop these VMs by installing a container runtime such as Docker[5].

EC2 offers different automated scaling types: *Dynamic Scaling*, *Scheduled Scaling*, and *Predictive Scaling*. All of them exploit *CloudWatch*, the AWS' monitoring system. CloudWatch lets users observe variations of predefined infrastructural metrics, such as CPU utilization or custom ones provided by the applications themselves. Metrics are aggregated into a single data point over a monitoring period (usually 60 or 120 seconds). CloudWatch allows for creating alerts when set thresholds of observed metrics are reached.

Dynamic Scaling is a reactive scaling system that automatically changes the number of VMs according to monitoring data. EC2 offers three sub-types of dynamic scaling: simple, step, and target scaling. *Simple* scaling permits users to create actions (or *policies*) that are executed when a certain CloudWatch alert is triggered. For example, the number of VMs can be increased/decreased if CPU utilization is greater/less than 70%/30%. Each time an action is executed, the system waits for a so-called *cool down period* (default is 180 seconds) before applying any further action. Since actions take time to execute, this period guarantees that different operations do not overlap.

Step scaling requires that users exploit adjustment tables to define scaling strategies. Given a CloudWatch alarm, users must specify for each percentage difference on observed metrics the percentage of instances to add/remove. For example, one can say that when the CPU utilization is greater than 70%, the number of VMs should be increased by 10% for values between $70\% - 80\%$, by 15% between $80\% - 90\%$, and by 20% for values greater than 90%. The number of added VMs is always rounded up to the nearest integer. Step scaling does not use cool down periods but users must specify a *warm-up time* for VMs. The system can execute multiple, subsequent actions, but added/removed VMs are only considered (not considered anymore) after the warm-up time.

Target scaling, which is the most advanced dynamic scaling functionality, is similar to what offered by GC. It allows users to set a target value for a metric and it automatically manages the VMs to keep that metric at the set point. AWS does not disclose the algorithm behind this technology, but it seems to scale instances proportionally to the change in the metric (as, for example, the algorithm behind HPA). In fact, AWS suggests to only use metrics that change values proportionally to the number of VMs. Scaling actions are activated automatically as soon as CloudWatch generates an alert. Users must set a minimum and a maximum number of instances to allocate and a warm up period similarly to step scaling.

Scheduled Scaling allows users to define time triggers for scaling actions. For example, users can decide to add a certain number of VMs at the beginning of the weekend and release them at the end. An evolution of this mechanism is *Predictive Scaling*, a proactive scaling system that uses machine learning to analyze the workload patterns of the last 14 days and foresee the resource demand for the next two days. Forecast data are updated each day with a one-hour granularity. *Predictive Scaling* works in conjunction with *Scheduled Scaling*: it automatically generates scheduled scaling actions according to the forecasts. Being proactive, *Scheduled Scaling* and *Predictive Scaling* are complemen-

[5]https://www.docker.com

tary to *Dynamic Scaling* and they are seen as mid/long-term actions on the rented infrastructure to prepare for the future.

AWS also offers different solutions to help manage containers. *Elastic Container Service* (ECS) provides a managed container orchestrator that allows one to deploy, manage, and scale containers by means of easy-to-use, and partially automated, interfaces. Containers are organized in *tasks*. A task, similarly to a Kubernetes pod, is a group of containers that are deployed and executed all together. Tasks are associated with a user-defined static resource allocation (CPU and memory). Tasks are run on a cluster of dedicated EC2 VMs and deployed onto them using one of the different available placement strategies (e.g., bin packing).

Table I shows that starting and terminating an ECS cluster of three *t3.medium* instances takes respectively 79 and 123 seconds. A *service* is an abstraction that groups different instances of the same tasks. Services help users set the desired amount of task instances to execute and manage. When an instance fails, the service immediately schedules a new one for execution. To start a service with a container that embeds an instance of NGINX, ECS takes 182 seconds while to add and remove a task instance to a service it takes 53 and 355 seconds, respectively.

Services provide autoscaling and load balancing facilities to distribute the workload among the different task instances. In particular, ECS supports *Dynamic Scaling* (step and target) and *Scheduled Scaling* for container services. AWS also offers another container-centric service called *Fargate*. It hides VMs to users who only manage containers, tasks, and services. Users are billed proportionally to the resources allocated for running their tasks, which cannot be changed at runtime.

Finally, AWS provides *Elastic Kubernetes Service* (EKS), a hosted and managed version of Kubernetes. AWS bills this service a fixed quota per hour per cluster in addition to the VMs rented for running the cluster. EKS can also be used together with Fargate to remove the need for managing the cluster. Table I shows that the initial configuration of EKS takes longer than the GCP's counterpart; the same applies to scaling the cluster out, while scaling it in is faster on AWS.

C. Azure

Azure offers two main infrastructural services: *Azure Compute* and *Azure Kubernetes Service* (AKS).

Compute provides the main functionality of a standard IaaS. VMs of different types can be rented on demand and billed on a per-second basis. We measured the time it takes to start and terminate a Linux Fsv2 VM equipped with 2 CPUs and 4GB of RAM. Table I shows that the two actions were slower compared to the times measured when using GCP and AWS.

Compute provides different types of scaling mechanisms. Applications can be monitored using *Azure Monitor*, which supports both predefined (e.g., CPU and memory utilization) and custom metrics. Users must specify a maximum and a minimum number of VMs and then define rules to let a monitoring event trigger a scaling action (add/remove a constant number of VMs). Moreover, users can associate scaling operations to specific time intervals (e.g., for the next two days) or repeat the same operations on certain dates (e.g., on weekends).

Finally, AKS provides a hosted version of Kubernetes that offers the same functionality as the one provided by GCP and AWS. Kubernetes is run on Azure Compute VMs and there is no additional charge for the service. According to our measurements, AKS is the fastest provider for configuring Kubernetes clusters (212 seconds vs 234 with GCP and more than 700 with AWS).

III. *COCOS autoscaling*

COCOS autoscaling [6] is our custom autoscaling solution. Our work started from control theory, as theoretical enabler for fast control loops, and containers as means for the fast enaction of computed resource allocations. As already explained, containers are faster than VMs and easier to manage. Note that the last three rows of Table I show the time (in seconds) needed to start (1), terminate (0.5), and scale vertically (0.2) an NGINX container. *COCOS autoscaling* only manages container-based applications and differs from the aforementioned industrial solutions for the following four main reasons:

Fast and fine-grained vertical scalability: COCOS autoscaling is fast because it exploits vertical scalability applied to deployed containers that run on a given set of VMs[6]. Resources are continuously changed (*each second*), and can thus closely follow the workload, without restarting containers. Scalability is based on a feature of the Linux kernel, called *cgroup*, which provides means for configuring memory and CPU cores. Computing resources can be changed by means of CPU *shares*, *reservation*, and *quotas*. Shares set a soft limit to the CPU cores used by each container. This limit is only enforced in the case of resource contention. Reservation allows for pinning a container to a set of CPU cores, but it does not provide means for allocating fractions of cores. *COCOS autoscaling* exploits quotas since they set an hard limit on resource usage and cores can be allocated with decimal precision. Memory can be allocated to containers in a hard or soft way. The former sets a strict upper bound to usable memory; the latter gives more freedom.

Distributed model-based control: COCOS autoscaling associates a lightweight control-theoretical planner with each container. Used controllers depend on application types. So far, we have developed controllers for microservices, big-data batch applications (i.e., Spark applications [8]),

[6]Readers interested in how VMs and containers are deployed and scaled horizontally can refer to [6] for more details.

and GPU-empowered machine learning applications (i.e., TensorFlow). These controllers aim to guarantee envisioned execution times while optimizing consumed resources. The fact that controllers need not be synchronized and resource allocation is computed in constant time allows for fast, one-second control periods.

Compatibility with other scaling solutions: COCOS autoscaling focuses on vertical scalability and it can be integrated with existing or custom [9] horizontal scaling systems for containers and virtual machines. *COCOS autoscaling* produces significant indicators on required resources at both application and system level. Other horizontal autoscaling solutions can exploit these data and change the number of container replicas (e.g., Kubernetes HPA) and/or the number of VMs (e.g., AWS target scaling).

Control guarantees: Control-theory provides four types of formal guarantees on the control (allocation) carried out [10], [11]: *stability*, that is, the ability of the controlled system to reach a fixed point and remain in its neighborhood, *settling time*, that is, how fast the system converges to a stable point, *maximum overshooting*, that is, the maximum gap between the set-point and the measured controlled variable during settling time, and *steady-state* error time, that is, the difference between the point reached and the set-point. To the best of our knowledge, none of the three approaches of Section II provides such guarantees, even if not all the algorithms are fully disclosed in the documentation.

A. Control Architecture

Figure 1 shows the control architecture *COCOS autoscaling* exploits at container and machine (VM) levels. Each container is equipped with a dedicated controller that is deployed automatically given the application type. Similarly to AWS' target dynamic scaling, users set a target response time (τ°). During container execution, disturbances (D) affect the actual response time (τ), which is continuously monitored along with a set of other relevant metrics (M). The controller employs a feedback loop to compute, at each control step (1 second), the control error ϵ as the difference between τ° and τ. The controller uses ϵ together with M to compute the current resource demand u_C. Ideally, u_C is the exact quantity of CPU cores needed to achieve $\tau = \tau^\circ$.

Multiple containers/controllers are deployed on a single VM and are independent of one another, that is, they do not interact during execution. This means that the sum of computed resource demands can be greater than the actual number of CPU cores provided by the machine (resource contention). *COCOS autoscaling* deploys a *supervisor* on each VM and resource demands are not directly transferred to containers but sent to this component. At each control period, the supervisor aggregates all the u_C and, if needed, computes a feasible resource allocation u'_C by downscaling the demands according to a specified policy (e.g., proportional, priority-based, requirement-based [8]). If the sum of

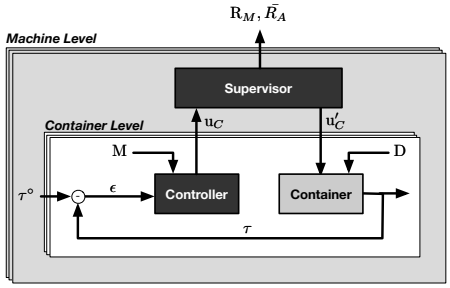


Figure 1: *COCOS autoscaling*'s control architecture.

resource demands is less than available ones, the supervisor can be configured to scale up demanded resources to boost applications' performance at the expense of a sub-optimal allocation (over-provisioning).

Finally, before provisioning each container with u'_C CPU cores, the supervisor calculates two indicators: R_M and $\bar{R}_A$, that is, the saturation level of the VM and the needs of each application running on it. We then aggregate the data retrieved from each VM into system-level indicators. This information eases the allocation (horizontal scalability) of VMs and containers (e.g., as done in [6]).

IV. Evaluation

To enable and ease the comparison of different autoscaling solutions, we developed a simulator called *RAS* (Resource Allocation Simulator). *RAS* is a lightweight simulation environment[7], written in Python, that allows one to mock different autoscaling solutions. It comes with a library of existing solutions and workloads, but developers can easily customize them and conceive new ones. *RAS* is organized around the following components:

Applications: They mimic the behavior of cloud applications. They define a function that computes the average response time given the amount of allocated resources and the number of requests to process. To mimic concurrent requests, in this paper we wanted a function with a hyperbolic relationship (monotonically decreasing) between response time and the ratio between requests to serve and allocated cores. We also wanted the function to have a horizontal lower asymptote since once available cores are enough to serve all requests, the addition of new cores would not further decrease the response time. A practically acceptable function, taken from [9] can be the following:

$$RT = (1 + rand()) * \frac{(c_1 + c_2) * req + c_1 * c_3 * cores}{req + c_3 * cores} \quad (1)$$

where req is the number of requests to serve, $cores$ the amount of CPU cores allocated to the application, and c_1, c_2, c_3 were obtained through profiling in previous works.

[7]Source code and the complete experiments we carried out are available at https://github.com/deib-polimi/RAS

These three parameters define how the behavior (response time) is affected by changes in core allocation and workload. We also added a noise function $rand()$ that returns a number between -0.1 and 0.1 ($\pm 10\%$ disturbance). This model also implicitely assumes that memory be scaled proportionally to cores and, since applications are CPU-bound, always sufficient. Applications also come with a *less-than* requirement on their average response time (SLA).

Workload Generators: They provide a function to bind each time instant to a number of requests to serve. *RAS* currently offers three workload types: *step*, *ramp*, and *sin*. *Step* generates square-wave shaped workloads that associate a number of requests with different-length time intervals. The transitions between intervals are immediate (impulse like). *Ramp* produces workloads that are constantly ascending or descending with a given slope up to a certain point in time when the workload becomes constant. *Sin* produces periodic workloads that follow a sin function.

Controllers: They implement autoscaling mechanisms, have control periods (frequency of control), and can have cool down/stabilization/warm-up periods (as defined in Section II). They exploit a *Monitoring* component to store the current number of requests to serve (generated by a *Workload Generator*) and the *Application*'s response time (computed using the proper model) at each point in time. These data allow a *Controller* to compute and actuate the resource allocation for the next control period.

RAS provides four built-in controller types. *Simple* controller mimics the behavior of AWS' and Azure's rule-based autoscaling systems, *Step* controller implements the step autoscaling solution provided by AWS, *Target* controller simulates Kubernetes' HPA as well as Google's Autoscaler and AWS' target autoscaling system (according to provided details), and the solution embedded in *COCOS autoscaling* [9]. We only selected this last type, among those we developed, for fairness: the other autoscaling systems are not meant to control big-data batch applications or GPU-empowered ones.

A. Assessment

The experiments presented here reused $c_1 = 0.007$, $c_2 = 1.8$ and $c_3 = 565.8$ as reported in [9]. We also set RT (response time) to always be less than 0.6 seconds (SLA), tha is, a reasonable value for an interactive application [12]. We ran experiments (simulations) for 1000 seconds (longer experiments produced similar results). The experiments exploited the six workloads described in Table II, where t is measured in seconds.

Since surveyed industrial solutions allow one to use simple, step and target controllers on both VMs and containers (CR in the next tables and figures), we simulated their behavior when controlling both resource types: six controllers in addition to *COCOS autoscaling*, which we only used with containers. This means that, for example, *SimpleVM* refers a simple controller used on VMs and *SimpleCR* on containers. With VMs we set a control period of 180 seconds to accommodate all the delays reported in Table I to start and terminate VMs, and of 30 seconds for containers (as in the HPA). Moreover, we set cool down, stabilization, and warm-up periods to 0. We used smaller values than the default ones to speed up the decision process. For *COCOS autoscaling*, we set a control period of 1 second.

Name	Type	Description
SN1	Sin	$req(t) = 500 * sin(t\frac{\pi}{100}) + 700$
SN2	Sin	$req(t) = 1000 * sin(t\frac{\pi}{50}) + 1000$
SP1	Step	$req(t) = 1000 * (1 + floor(t/100))$
SP2	Step	$req(t) = 30000$ if $50 \leq t < 800$ else $req(t) = 50$
RP2	Ramp	$req(t) = 10t$ if $t < 800$ else $req(t) = 8000$
RP2	Ramp	$req(t) = 20t$ if $t < 800$ else $req(t) = 16000$

Table II: Used workloads ($req(t)$ means requests at time t).

We configured simple controllers with the following rule: *add 1 core if* $RT \geq 0.9 * SLA$, *remove 1 core if* $RT \leq 0.5 * SLA$, and step controllers with the following policy: *increase core allocation by* 10% *if* $0.9 * SLA \leq RT < SLA$, *by* 20% *if* $SLA \leq RT < 1.1 * SLA$, *and by* 30% *if* $RT \geq 1.1 * SLA$, *decrease core allocation by* 10% *if* $RT < 0.8 * SLA$. Target and *COCOS autoscaling* controllers were set to keep $RT = 0.8 * SLA$.

Table III shows obtained results. Columns $Approach$ and W report used controller and workload, respectively. The minimum (m), maximum (M), and mean (μ) response times, along with standard deviation (σ) are shown in dedicated columns. Columns V shows the number of SLA violations and column A_μ gives the average number of allocated cores.

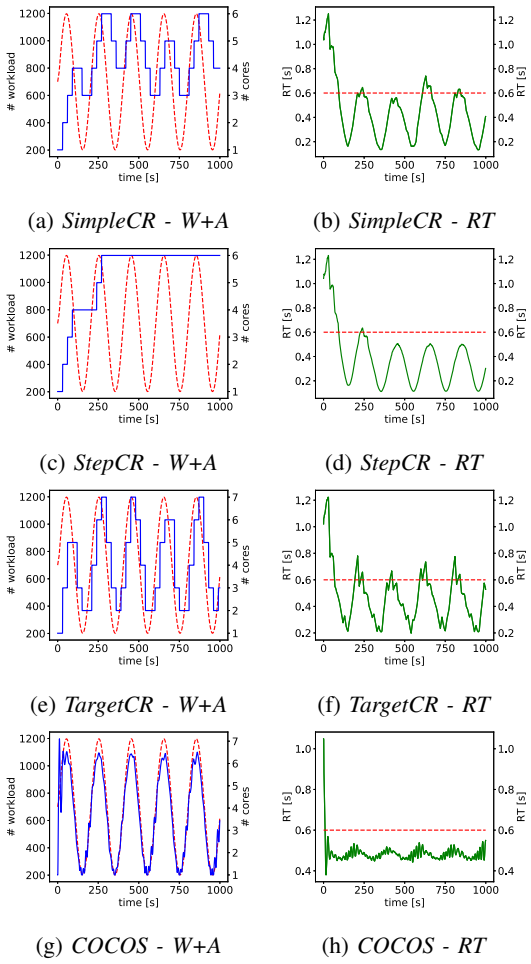
These data show that the intrinsic speed enabled by containers allows container-based approaches to outperform (i.e., lower SLA violations) VM ones in all cases but with the exception of *TargetVM* with workload $SN2$: apparently target controllers have problems with sin workloads. *StepVM* and *SimpleVM* show similar performance in all cases (similar V), while *StepCR* obtained better results than *SimpleCR*. Target controllers obtained fewer violations and smaller RTs than step and simple controllers in all cases with the exception of sin workloads.

COCOS autoscaling outperformed all competitor approaches except for the experiment with workload $SP2$. Compared to simple, step, and target controllers, *COCOS autoscaling* reduced the number of violations from 1 to 3 orders of magnitude. *COCOS autoscaling* also outperformed *TargetCR* in all the experiments except for workload $SN2$ where results are similar (shorter response time for *COCOS autoscaling*, fewer violations for *TargetCR*, similar allocations). Note that the standard deviation of *COCOS autoscaling* is always lower than that of the other approaches, thus witnessing a more stable autoscaling system.

Since Table III shows that target controllers are the best competitor of *COCOS autoscaling*, we tried to set its control period to 1 second. Obtained results, not presented here for

		RT					
Approach	**W**	μ	σ	m	M	V	A_μ
SimpleVM	*SN1*	0.69	0.30	0.16	1.30	619	2
SimpleCR	*SN1*	0.45	0.24	0.13	1.25	186	4
StepVM	*SN1*	0.69	0.30	0.16	1.30	617	2
StepCR	*SN1*	0.39	0.24	0.11	1.24	115	5
TargetVM	*SN1*	0.69	0.30	0.16	1.29	623	2
TargetCR	*SN1*	0.47	0.20	0.20	1.24	210	3
COCOS	*SN1*	0.48	0.04	0.38	1.05	10	3
SimpleVM	*SN2*	0.77	0.46	0.02	1.50	627	2
SimpleCR	*SN2*	0.50	0.35	0.02	1.49	414	4
StepVM	*SN2*	0.77	0.46	0.02	1.50	626	2
StepCR	*SN2*	0.38	0.31	0.01	1.48	141	7
TargetVM	*SN2*	0.65	0.48	0.01	1.50	458	4
TargetCR	*SN2*	0.59	0.40	0.02	1.49	504	4
COCOS	*SN2*	0.43	0.15	0.06	1.21	82	5
SimpleVM	*SP1*	1.40	0.08	1.20	1.51	1000	3
SimpleCR	*SP1*	0.72	0.11	0.61	1.26	1000	17
StepVM	*SP1*	1.37	0.08	1.20	1.50	1000	3
StepCR	*SP1*	0.59	0.15	0.50	1.23	219	24
TargetVM	*SP1*	0.87	0.30	0.50	1.51	816	17
TargetCR	*SP1*	0.53	0.14	0.45	1.24	92	27
COCOS	*SP1*	0.50	0.06	0.37	1.26	18	27
SimpleVM	*SP2*	1.37	0.75	0.04	1.90	754	3
SimpleCR	*SP2*	1.16	0.65	0.01	1.87	753	14
StepVM	*SP2*	1.37	0.75	0.03	1.91	754	3
StepCR	*SP2*	0.77	0.61	0.01	1.90	434	79
TargetVM	*SP2*	1.08	0.68	0.01	1.90	748	30
TargetCR	*SP2*	0.50	0.36	0.01	1.87	133	105
COCOS	*SP2*	0.47	0.25	0.01	1.72	146	104
SimpleVM	*RP1*	1.31	0.22	0.01	1.47	970	3
SimpleCR	*RP1*	0.62	0.07	0.01	0.67	828	17
StepVM	*RP1*	1.28	0.22	0.01	1.47	969	3
StepCR	*RP1*	0.54	0.06	0.01	0.65	100	21
TargetVM	*RP1*	0.84	0.28	0.01	1.43	744	15
TargetCR	*RP1*	0.50	0.06	0.01	0.65	19	24
COCOS	*RP1*	0.48	0.05	0.01	0.54	0	24
SimpleVM	*RP2*	1.53	0.22	0.01	1.67	982	3
SimpleCR	*RP2*	0.93	0.10	0.01	1.00	982	17
StepVM	*RP2*	1.52	0.22	0.01	1.66	982	3
StepCR	*RP2*	0.59	0.13	0.01	0.95	243	43
TargetVM	*RP2*	0.97	0.34	0.01	1.64	887	26
TargetCR	*RP2*	0.53	0.10	0.01	0.94	115	48
COCOS	*RP2*	0.50	0.04	0.01	0.56	0	48

Table III: Obtained results.



(a) *SimpleCR - W+A* (b) *SimpleCR - RT*

(c) *StepCR - W+A* (d) *StepCR - RT*

(e) *TargetCR - W+A* (f) *TargetCR - RT*

(g) *COCOS - W+A* (h) *COCOS - RT*

Figure 2: CR-based autoscaling systems on $SN1$ workload.

lack of space, showed these controllers significantly oscillate as for resource allocation without even reaching a stable point with most of the workloads (*TargetCR* performed better than this fast version).

Figure 2 helps better visualize obtained results. It presents eight charts that show the results we obtained with controllers *SimpleCR*, *StepCR*, *TargetCR*, and *COCOS autoscaling*, respectively, on workload $SN1$. For each controller, the first chart (called *W+A*) shows the workload (dashed line, left axis) and allocated cores (solid line, right axis); the second chart presents the obtained response time (solid line) and SLA (dashed horizontal line). The charts show how *COCOS autoscaling* is able to better follow the evolution of the workload with a faster and more precise allocation; it also keeps the response time almost constant during the experiments.

Finally, we highlight some threats that may affect the validity of our results [13]. As for internal threats, we used a simulator and synthetic workloads for testing the autoscaling systems. The use of real systems would have been much more expensive and time-consuming. We are confident that the actual measures summarized in Table I are enough to guide our experiments and feed our simulator properly. *RAS* also aims to offer a simple and cheap tool to test and compare diverse autoscaling solutions. Proposed workloads are designed to be a suite of *stress tests* for the different approaches and highlight major drawbacks or positive features.

For external threats, we cannot generalize the results to applications that are not interactive and/or not CPU-bound. This is a first assessment and we will extend the evaluation with more application types in the future.

V. Related Work

The problem of allocating resources to cloud applications at runtime has been widely studied [14], [15]. Existing solutions cover both horizontal [16], [17] and vertical scaling systems [12], [18]. In general these works either: i) provide slower controllers than *COCOS autoscaling*, or ii) only

support a specific application type or iii) are not designed to control multiple applications cooperatively as *COCOS autoscaling* does.

As an example, Yu et al. [19] recently present Microscaler, a tool devoted to the runtime management of microservices constrained by SLAs. As *COCOS autoscaling*, it focuses on response times and its goal is to produce close-to-optimal allocations when multiple inter-dependent microservices are controlled. Microscaler is based on Machine Learning and heuristics and provides a horizontal autoscaling solution on top of Kubernetes. Its decision process lasts 2 minutes, and thus it 2 orders of magnitude slower than *COCOS autoscaling*. It does not provide formal guarantees on the control and it does not support vertical scalability as *COCOS autoscaling*.

Besides this work, other surveys and comparisons of autoscaling techniques for cloud applications exist. Podolskiy et al. [20] propose a comparison among different autoscaling approaches. They deployed Kubernetes on AWS, GCP, and Azure and tested their rule-based (simple, in our experiments) autoscaling of VMs in combination with an old version of HPA that was also rule-based. They show that scaling actions took between 6 to 155 seconds, and thus confirm both our detailed evaluation of the delays (Table I) and that *COCOS autoscaling* is able to scale faster then the others. They also only compare rule-based approaches, while we tested different solutions with both VMs and containers.

Netto et al. [21] introduce Auto-scaling Demand Index (ADI), an indicator to evaluate autoscaling actions. They evaluate different types of autoscaling strategies using ADI. Their results show that predictive allocations do not perform well with highly dynamic workloads and that policies that use fixed addition/removal of resources are generally worse than adaptive ones (as *COCOS autoscaling*).

VI. Conclusions

This paper surveys the main industrial autoscaling solutions, for VMs and containers, and uses *RAS*, a dedicated simulator, to compare them against our solution *COCOS autoscaling*. Obtain results show that *COCOS autoscaling* outperforms all the other approaches.

References

[1] I. Gorton and J. Klein, "Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems," *IEEE Software*, vol. 32, no. 3, pp. 78–85, 2015.

[2] N. Sfondrini, G. Motta, and L. You, "Service level agreement (sla) in public cloud environments: A survey on the current enterprises adoption," in *2015 5th Int. Conf. on Information Science and Technology*. IEEE, 2015, pp. 181–185.

[3] S. Dustdar, Y. Guo, B. Satzger, and H. L. Truong, "Principles of Elastic Processes," *IEEE Internet Computing*, vol. 15, no. 5, pp. 66–71, 2011.

[4] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[5] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, 2014.

[6] L. Baresi and G. Quattrocchi, "Cocos: A scalable architecture for containerized heterogeneous systems," in *2020 IEEE Int. Conf. on Software Architecture (ICSA)*, 2020, pp. 103–113.

[7] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *Proc.of the IEEE Fifth Int. Conf. on Cloud Computing*. IEEE, 2012, pp. 423–430.

[8] L. Baresi, A. Leva, and G. Quattrocchi, "Fine-Grained Dynamic Resource Allocation for Big-Data Applications," *IEEE Transactions on Software Engineering*, Early Access 2019.

[9] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A Discrete-Time Feedback Controller for Containerized Cloud Applications," in *Proc. of the 24th Int. Sym. on Foundations of Software Engineering (FSE)*. ACM, 2016, pp. 217–228.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems*. John Wiley & Sons, 2004.

[11] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 784–810, 2017.

[12] E. Lakew, A. Papadopoulos, M. Maggio, C. Klein, and E. Elmroth, "Kpi-agnostic control for fine-grained vertical elasticity," in *Proc.of the 17th IEEE/ACM Int. Sym. on Cluster, Cloud and Grid Computing*. IEEE, 2017, pp. 589–598.

[13] C. Wohlin *et al.*, "Empirical research methods in web and software engineering," *Web Engineering*, 2006.

[14] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[15] W. Delnat, E. Truyen, A. Rafique, D. Van Landuyt, and W. Joosen, "K8-scalar: A workbench to compare autoscalers for container-orchestrated database clusters," in *Proc.of the 13th Int. Conf. on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2018, p. 33–39.

[16] C. Guerrero, I. Lera, and C. Juiz, "Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture," *Journal of Grid Computing*, vol. 16, no. 1, pp. 113–135, 2018.

[17] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive Control of Virtualized Resources in Utility Computing Environments," in *Proc. of the 2nd EuroSys Conf.* ACM, 2007, pp. 289–302.

[18] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith, "Runtime vertical scaling of virtualized applications via online model estimation," vol. 2014, 09 2014.

[19] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE Int. Conf. on Web Services*, 2019, pp. 68–75.

[20] V. Podolskiy, A. Jindal, and M. Gerndt, "Iaas reactive autoscaling performance challenges," in *2018 IEEE 11th Int. Conf. on Cloud Computing (CLOUD)*, 2018, pp. 954–957.

[21] M. A. S. Netto, C. Cardonha, R. L. F. Cunha, and M. D. Assuncao, "Evaluating auto-scaling strategies for cloud computing environments," in *2014 IEEE 22nd Int. Sym. on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, 2014, pp. 187–196.