Quantifying Cloud Elasticity on Smart Devices with Container-based Autoscaling

Xuxin Tang[1, #], Fan Zhang[2, #],
Xiu Li[2, *], Samee U. Khan[3], Zhijiang Li[1, *]

1.Division of Information Science, School of Printing and Packaging, Wuhan University, Wuhan, China, lizhijiang@whu.edu.cn
2.Division of Information Sci. & Tech., Graduate School at Shenzhen, Tsinghua University, Shenzhen, China, li.xiu@sz.tsinghua.edu.cn
3. Department of Electrical and Computer Engineering, North Dakota State University, Fargo, USA
# These authors contributed equally to this work with last names in alphabetical order * Co-corresponding Authors

***Abstract*— Containers have been a pervasive approach to help rapidly develop, test and update the Internet of Things applications. The autoscaling of containers can adaptively allocate computing resources for various data volumes over time. Therefore, elasticity, a critical feature of a cloud platform, is significant to measure the performance of lightweight containers on smart devices. In this paper, we propose a framework with container auto-scaler. It monitors containers resource usage and accordingly scales in or scales out containers in need. Further, we define elasticity mathematically in order to quantify the cloud elasticity using the proposed framework. Extensive experiments are carried out with different workload modes, workload durations, and scaling cool-down period of times. Experiment results show that the framework captures the workload variation firmly with a very short delay. We also find out that the cloud platform shows the best elasticity in repeat workload mode due to its recurring and predictable feature. Finally, we discover the length of the cool-down period should be properly set up in order to balance system stability and good elasticity.**

Keywords—autoscaling; container; elasticity; pervasive computing

I. INTRODUCTION

The Internet of Things (IoT), is constituted by the highly connected smart devices, such as environmental sensors, medical trackers, home appliances, and industrial devices. Due to the varying data volumes of smart devices, enterprises and researchers look for methods that enable them to manage those devices and compute the real-time streams of data [23]. Therefore, cloud companies offer relational services, such as Amazon AWS [1] and Microsoft Azure [3] for example. IBM [2] also provides an open Bluemix platform for container-based virtualization on Linux.

However, in reality, there are some services on smart devices, which need to consume a wide range of workloads having different resource demands. Those demands sometimes can be predicted but other times can not. Therefore, an elasticity model needs to be introduced to model the supply-demand compute resources relations for those devices in the cloud environment.

For the purpose of saving runtime cost, elasticity is the fundamental build stone of cloud computing. While under-provisioning leads to performance degradation, over-provisioning causes the waste of resources. Elasticity can measure the capability of a cloud platform in terms of its adaption of workload changes by scaling out or scaling in resources automatically [4]. It addresses two major aspects [4] [5]: (1) cost efficiency, allocating only the compute resources in need, (2) time efficiency, allocating idle resources for the present request immediately.

Containerization is the state-of-art virtualization of the cloud platform [11]. Containers only need seconds to bootstrap, initiate, versus minutes for a regular VM. Docker uses libcontainer, allowing containers to utilize more host operating system resources than VMs. It also adopts a layered file system AuFS [6], which remains the base layer as read-only, so that users' change will be saved as the top layer. Consequently, it only saves the changes across two consecutive file system versions. In this way, users are able to restore the OS as a shared image and its deployment in another server environment easily.

TABLE I. COMPARISON BETWEEN CONTAINER AND VIRTUAL MACHINE

Performances	Kinds of Virtualizations	
	Container	*Virtual Machine*
Size	Megabytes	Hundreds Megabytes
Start time	Seconds	Minutes
Management Overhead	Low	High
Portability	High	Low

In terms of its lightweight feature, containers are fit to serve the high elasticity needs of a cloud platform. As containers need minimal sharing of the computing resource, e.g. cgroups are leveraged to allocate resources on servers in a far more granular fashion; more concurrent containers can be launched on a single machine. Moreover, containers demand lower compute resource overhead and have better performance as compared to VMs. Also, containers have higher portability than VMs. These small instances are isolated from each other and pack applications so that can be conveniently moved or copied from one cloud deployment to another.

Due to their speed and agility, containers are expected to widely applied on the smart devices. Meanwhile, the proliferation of using containers paves the way for our interest in utilizing the containerization in an elastic and autoscaling cloud. Our contributions based on this are summarized below:

1. We implemented an autoscaling service, called Auto-scaler, on a containerized orchestration platform, consisting of four components, to control the scale-in/scale-out of containers when workloads fluctuate in the small devices.

2. We defined a measurable cloud elasticity by providing executable analysis on the elastic performance of container-based autoscaling on individual smart devices.

3. We provided intensive elasticity measurement upon the container autoscaling. The proposed container-based auto-scaler is tested using the *Stress* [14] workload on the Mesosphere Data Center Operating System (DC/OS) cloud infrastructure and the tests lend us lots of valuable experience in automatic container autoscaling.

The remainder of the paper is organized as follows: Section II reviews the previous researches of autoscaling approaches; Section III presents our algorithm by separating it into several parts; Section IV provides an elasticity evaluation of containers cluster; Section V presents an experimental result of our work and analysis the data it provides; Section VI draws a conclusion with discussion of our technique and future work.

II. RELATED WORK

A. Autoscaling techniques in clouds

Autoscaling is mainly applied in cloud computing, whereby a number of computational resources can be scaled in/out automatically according to the real-time workloads. Before cloud computing, the provided resources have to exceed the normal demand a lot to maintain stability away from losing data or potential business, also contributing to the redundant waste of computing resources and energy. In fact, Autoscaling can solve these two problems with no upper limit of resources theoretically. Therefore, the improvements and applications of autoscaling techniques have kept growing these years since 2009, when the earliest autoscaling research [7] on cloud arose.

The various autoscaling techniques are divided into two kinds by [13], predictive and reactive systems. The former systems aim to develop prediction models, such as reinforcement learning [8] and machine learning [10], in order to allocate resource in advance. Also, those mathematical model firstly should be trained by real workloads ([12], [15], [16]). However, the training model for a sport website might not be appropriate for an education website and consequently, every website should train their own predictive models in advance which needs much human intervention to adjust periodically. Thus, a simpler and common solution is needed by the most companies or institutions. The threshold-based strategy is a kind of reactive systems in which the virtual machines are effectively shut on or off with little manual intervention. Autoscaling in AWS allows customers to create self-specified Auto Scaling groups and scaling policies by which the load for applications is available automatically handled. While the groups maintain the minimum and the maximum number of instances in each group, the scaling policies enable instances launched or terminated as demand. RightScale [9] and Azure Automation [3] similarly supports manual scaling as well as threshold-based automatic scaling. Thus, we adopt the threshold-based strategy in our algorithm to simulate the environment of big cloud companies.

B. Elasticity in Cloud Computing

Elasticity is a significant reference in evaluating a cloud platform. High elasticity corresponds to short respond time, high processing frequency and small granularity in scaling actions. The widely accepted definition of elasticity was given in [4]. The elasticity is described as the degree to which a system provisions or de-provisions computational resources dynamically along with workload changes in an automatic manner. Elastic system restrains the timely provided resources matching the current demand as closely as possible.

The performance metrics vary from distinctive applications, which are assorted into hardware, general OS process, load balancer, application server, database and message queue [13]. Metrics in measuring the elasticity of hardware selectively and always involve CPU utilization per VM, disk access, network interface access and memory usage. To estimate the elasticity of container, we select CPU utilization per container, CPU utilization per PM, response time and scaling speed in the experiment.

C. Linux Container

The virtualization technologies can be split up into three categories: full-virtualization, para-virtualization and container-based virtualization [17]. Linux Container is a container-based technique, classified as operating-system-level virtualization method, in which the kernel of an operating system runs multiple isolated user-space instances, instead of just one. Para-virtualization, such as Xen, shows the same ability of portability, isolation, and optimization of the hardware resource utilization, but LXC (Linux container) performs better in the execution of small isolated processes by introducing less overhead [17]. In addition, one container can be scaled out/in within a minute, and consequently can react immediately when encountering possible unforeseen crash. Therefore, the Linux container is capable of tolerating fluctuating stress and reducing overhead, the features which autoscaling coincidently needs.

As referred in the introduction of our paper, Docker is utilized extensively in cloud computing at present. It is produced as an open-source project [11] that automates the deployment of Linux applications inside software containers. The utilization of Docker in autoscaling has been researched these years. Researchers found Docker can facilitate troublesome difficulties than before, which is proved by [19], that Hsi-En Yu et al built a Virtual HPC Cluster of Docker with autoscaling to solve the software dependency issue under the HPC environment. What is more, containers can lessen the system overhead much. Experiments were carried by Hoenisch P. et al [20] on the four-fold autoscaling, involving the horizontally and vertically scaling of VMs and containers. By evaluating their approach with realistic apps, their results showed that the average cost per request can be reduced by about 20-28%. Therefore, we choose Docker container in our research on autoscaling.

III. Container-based Autoscaling

A. Introduction of the Scheduling Platform

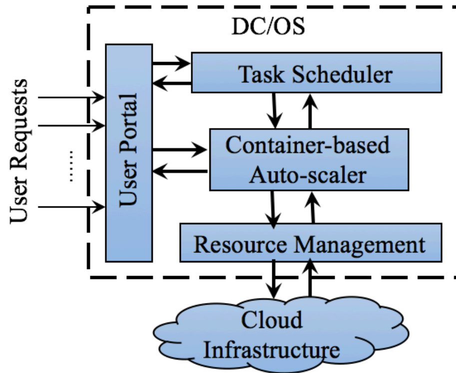


Fig. 1. The architecture of the DC/OS platform.

We design an automatic tool for scaling the containers in a cloud environment. The scheme is implemented on a platform, called DC/OS (known also as Datacenter Operating System) [21], to support containerized resource management. As shown in Figure 1, the DC/OS works as a middle-tier between user applications and IaaS cloud infrastructure. User Portal is an interactive website where one user request can be transferred to the DC/OS system in the form of a JSON file and the scaling up/down actions also reflect on it.

The Task Scheduler is built on Apache Marathon [22], a framework for container orchestration. When an application is needed to deploy on the cloud, the Task Scheduler receives the RESTful request from User Portal firstly and then launches an instance using corresponding Docker image. If there already exist containers running the same task, it will transfer the messages to the Container-based Auto-scaler.

At the end point, Zookeeper [19], applied in Resource Management, is used for coordination of master nodes and slave nodes in Cloud Infrastructure. It keeps track of all of the key information for multiple purposes, such as sharing of capacity information and disaster recovery.

The Container-based Auto-scaler considers two aspects of information: the requests received and the available compute resource exposed by Resource Management, e.g. CPU and memory. Taking all into consideration, the auto-scaler allocates corresponding compute resource across nodes from Cloud Infrastructure to provision or de-provision the containers. In short, the Autoscaling section, running as a standalone and third-party service, monitors the real-time computing resource usage of each container, adopts user-defined scaling policies, making scaling decisions and performs scaling actions.

B. Overview of the Container Auto-scaler

To ease understandability, the major parameters used in the auto-scaler algorithm are listed in TABLE II.

The strategy-based scaling considers values of five parameters. First of all, the lower and upper thresholds of the computing resources, in the form of the percentage, are important. Once the thresholds are exceeded, the auto-scaler begins to record the timestamp. The autoscaling action will be carried until the resource utilization has been keeping beyond the thresholds for t^D. If the over-provision or under-provision remain after last autoscaling action, it takes time t^I to wait and then scale automatically again. The largest amount of containers is related with the size of VMs and host PM, so make it based on the practical situation. The range of containers in our experiment is [0, 5]. Online measurements will be recorded constantly and make decisions through it.

TABLE II. Parameters of the Container-based Autoscaling Platform

Parameter	Description	Source
r	Two resource dimensions of a PM: r ∈ {CPU}.	
L(r)	The lower threshold set up for resource r.	Service users
U(r)	The upper threshold set up for resource r.	
t^I	The interval time, called cool-down time, between two autoscaling.	
t^D	The threshold of utilization has been exceeded for time t^D, called duration time.	
n	The increased or reduced amount of containers in a strategy when autoscaling.	
$[n^{min}, n^{max}]$	The minimum and maximum numbers of containers.	
N	The present number of containers.	Online Measurements
S(r)	The current utilization of the resources.	

The structure and principle of the auto-scaler are shown in Figure 2. It consists of four parts, monitoring mechanism, history recorder, decision mechanism and execution mechanism, respectively. They are further elaborated in detail in the following sections.

1) Monitoring Mechanism

The container-based auto-scaler sends out heartbeat message to the scheduler in DC/OS first and retrieves the information of all the applications. For each application, it checks its tag field and figures out if a scaling policy is defined. If so, it will parse the scaling policy definition and begin to poll the corresponding metrics on the Mesos master node. After doing this, a list of executors/containers under this same application, with the CPU usages, are retrieved.

After the first poll, it waits for a short period of time T_Δ for the second poll. Suppose the CPU *time cycle* of the two sampling time points are CPU_1 and CPU_2 respectively, the actual CPU usage within the two periods are $(CPU_2 + CPU_1)/T_\Delta$.

The monitoring mechanism considers all the containers with one same application that are deployed across multiple slave nodes, considering the metric in need and perform real-time performance report.

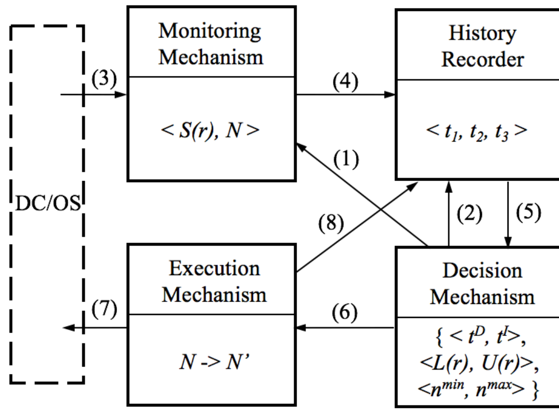


Fig. 2. .The framework of the container-based auto-scaler built as the custom development integrated with DC/OS. We separate the process of autoscaling into seven steps: (1) Decision section sends thresholds of metrics to Monitoring Mechanism firstly; (2) then Decision section also sends thresholds of metrics to History Recorder; (3) Monitoring Mechanism retrieves information of all the applications from Task Scheduler; (4) Monitoring section filters timestamps when thresholds are exceeded; (5) Decision Mechanism receives the signals from History part and then decides whether scaling or remaining; (6) Execution Mechanism is instructed by Decision part to send scaling requests; (7) Execution Mechanism transfers specific scaling requests to the Resource Management of DC/OS system; (8) History recorder memorizes the timestamp and actions of scaling. t1, t_2, t_3 are key timestamp information recorded. N' is the updated number of containers after scaling.

Algorithm 1 Monitoring Mechanism
Input: $S(r)$, $U(r)$, $L(r)$, N

1: **If** ($S(r)<L(r)$) || ($S(r)>U(r)$))
2: Transfer the signals to History Recorder.
3: **Else**
4: Remain unchanged.
5: **End If**
6: Send the timely performance report.

2) *History Recorder*

Managing the scaling history and its start/finish timestamp points are considered in this section. At any point, if we find an application has a metric running above its upper scaling threshold, this event together with the timestamp is recorded in a history recorder.

Users are allowed to define an induration period. If there are sufficiently many such above-scaling-threshold events during the induration period (t^D in Figure 3), it suggests the compute resource that user concern is highly utilized. At this point, the container auto-scaler will immediately send out a performance alert signal and will consequently consider a scaling action.

In Figure 3, the first scaling starts at time t^D and finishes at time ($t^D + T$). However, it is during this period of scaling time that the application shows its worst stability of performance. Therefore, we consider a relatively long time period t^I, that allows the application to be cooled down before considering the next cycle of the scaling period.

Users can define the induration period and the cool-down period, but pay attention to the cool-down period, if shorter than the scaling period T, will lead to very unpredictable scaling behavior that the users will lose control of the Autoscaler and the strategies set couldn't be correctly obeyed.

Algorithm 2 presents the process of History Recorder. The meanings of t_1, t_2, t_3 are: during t1, no scaling is expected since there are not enough counts of alerts that compute resource has exceeded the threshold; during t2, scaling action is triggered since all the needed criteria are met; during t3, no scaling is triggered since it is still within the cool-down period of time of the last scaling cycle. N' is the number of containers after scaling.

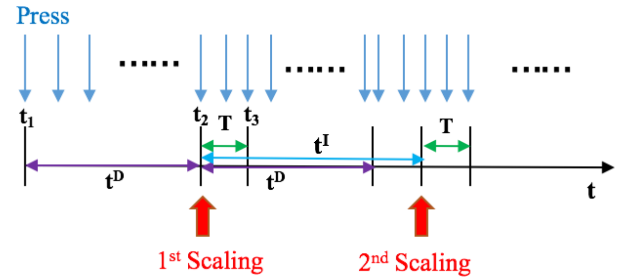


Fig. 3. Theory of the History Record. t^D is the time being pressed, which is calculated again after every scaling. t^I is the scheduled time between two scalings. The starting up of new containers needs T time.

Algorithm 2 History Recorder
Input: t, t^I, t^D
Output: t_1-The timestamp recording the first time Auto-scaler receive alerts, t_2-The timestamp when scaling action is triggered, t_3-The timestamp when the cool-down period ends.

1: **If** there is the first time receiving signals,
2: $t_1=t$;
3: **Elif** ($t-t_1>=t^I$)
4: $t_2=t$ && send signal to Decision Mechanism;
5: **Elif** ($t-t_2>=t^D$)
6: $t_3=t$ && send signal to Decision Mechanism;
7: **End If**

3) *Decision Mechanism*

The autoscaling strategy is pre-defined by each user per the need of their service deployed inside the containers. The containers' CPU usages are periodically polled by the auto-scaler. There are three conditions need to be met before an autoscaling action is triggered:

a) As introduced in Monitoring Mechanism that the containers CPU's usage hits a predefined scaling policy. If it hits the resource usage upper bound then a scaling out action is expected to add more containers to the application, otherwise, that will be a scaling in action to reduce the existing container count.

b) As introduced in History Recorder that the application is not within its cool-down period that during which time no further scaling action is allowed.

c) The application itself is ready to be scaled. There are many causes that an application is not ready to be scaled: either the application hasn't finished its previous round of scaling, or the application is under deployment or other non-active modes.

If all the three criterions have been met, and a scaling action from N containers to N' containers is expected. Otherwise, no scaling action should be performed. The whole process in detail can be found in Algorithm 3.

Algorithm 3 Decision Mechanism

Input: $\{ < t^D, t^I>, <L(r), U(r)>, <n^{min}, n^{max}>, \Delta n \}$

Output: N'

1: Sends $<L(r), U(r)>$ to Monitoring Mechanism;
2: Sends $<t^D, t^I>$ to History Recorder;
3: **If** History Recorder sends signals to Decision Mechanism,
4: $N'=N\pm\Delta n$;
5: **Elif** $((N'>=n^{min})\&\&(N'<= n^{max}))$,
6: Change the number of containers to N';
7: Send the timestamp and action to History Recorder;
8: **Else**
9: Remain Unchanged;
10: **End If**

4) Execution Mechanism

If the container auto-scaler finally decides to scale, it will consequently send a request to Marathon, denoting the new number of containers that the service expects to use Marathon, after having received the request, will consequently send out an update request to the master nodes to resize the cluster. Three steps are listed in the following paragraphs to demonstrate the process.

a) Scaling Out and Scaling in

Scaling out a container group entails the need that the leading master node should check if all the slave nodes have enough computing resource to accommodate the new cluster. Scaling in is a little tricky since it is likely all the containers are being used to serve workload. The easiest solution would be giving the containers to be recycled a grace period of time until they can be killed. This can also be resolved by disabling the containers from the service discovery system and all the requests for these containers are re-routed to the live containers.

b) Publish Containers' IP and Ports

After a scaling out action, the new containers are deployed. Each one will publish its hosting machine's IP address and the mapping port, then after the framework has passed the health-check of the new containers, upcoming requests could be load balanced to the new containers.

c) Health-check Containers

There are two levels of health-checking mechanisms. The first level starts from the service perspective, that each service owner designs a health-checking endpoint and checks if the service inside the container is launched properly. If this health check fails, the service owner can implement multiple ways to deal with this issue, such as re-starting the service inside the container, etc. The second level health checking happens at the container level. If a container fails to function for a certain period of time, then the framework will kill the container and redeploy the service in another host.

IV. PERFORMANCE EVALUATION

In this section, we describe the experimental settings and introduce the performance of the container auto-scaler. The evaluation is performed as a reactive autoscaling.

A. Experimental Settings

The experiment is carried out on a physical machine of an 8G memory with an Intel processor of Core™ i5-46707 CPU @ 2.3GHz *4. There are three virtual machines, a boot node, an agent node and a master node respectively, on a DC/OS 1.7 platform. DC/OS has the entire Mesos/Marathon/Zookeeper framework that eases the deployment significantly. Containers are deployed on the slave node controlled by our system in an automatic manner. The application used to test the system is *Stress* workload [14], which can stress the CPU to each container as needed.

The experiment is conducted strictly in following the three steps: Configure the container auto-scaler's parameters at first, including thresholds of compute resource utilization, cool-down period, induration period, scaling factor. After that, we obtain and save the metrics of the measurement data. Finally, we evaluate the performance of container auto-scaler and study the overall elasticity of the platform.

B. Elasticity measurement

We modify the elasticity measurement in [4] and design elasticity evaluation, including objective and subjective evaluations, to work in the container-based environments.

1) The objective evaluation

Figure 4(a) presents one of the experiments and three metrics are revealed. It shows the measured CPU utilization of the containers and executor respectively. The right vertical axis presents the real-time amount of containers. Even though there is a slight latency between the actual amount of running containers and resource usage, we can still find a strict positive relationship between them when scaling in and out. With all these figures considered, evaluation of the container auto-scaler performance can be inferred from the five metrics.

2) The subjective evaluation

There are several important metrics used in our method, shown in the following TABLE III. The container auto-scaler is one of the reactive autoscaling systems and the unknown workload is the prerequisite. Therefore, we adapt the measurement method in [4] to our need.

TABLE III. THE METRICS OBTAINED FOR ELASTICITY EVALUATION

Metrics	**Description**
T	The timespan of the experiment.
T_O	Aggregated timespan for scaling out the containers.
T_I	Aggregated timespan for scaling in the containers.
N	The actual number of containers during each scaling phase.

We propose an evaluation plan composed of obtaining four metrics: (1) The timespan of the whole experiment, (2) The aggregated timespan of either scaling out or scaling in, (3) The instances added or reduced during each scaling phase and (4) the average of these major metric values.

We define the deployment speed of containers as $S = \Delta N/\Delta T$, which calculates the average number of containers that can be deployed within a scaling period of time. Take scaling out as an example. $\overline{S_O}$ represents the average scaling-out speed across the entire experiment. $\overline{T_O}$ is used to measure the average scaling-out time length for each scaling-out period.

Since cloud elasticity reflects how fast a cloud platform could react to variations of workload changes in terms of the running containers, apparently it is positively proportional to the deployment speed $\overline{S_O}$ and inversely proportional to the scaling-out time length $\overline{T_O}$. In terms of the scaling-out period, we define the measure cloud elasticity as $E_O = \frac{1}{\overline{T_O}} * \overline{S_O}$. Therefore, the scaling-in periods can be similarly defined as $E_I = \frac{1}{\overline{T_I}} * \overline{S_I}$. And combining them together, the elasticity of the whole experiment can be calculated as $E = \frac{1}{\bar{T}} * \bar{S}$.

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. Impact of Different Workload Modes

The *Stress* workload has four different modes, single mode, repeat mode, shuffle mode and repeat with shuffle mode. In a single mode, we run the stress test for one single round. In a repeat mode, we repeat the single run a lot of times. In a shuffle mode, we set up multiple degrees of workload within a spectrum of heavy workload to light workload. They are executed in a defined but mixed sequence. In a shuffle-repeat mode, the shuffle mode is executed in a totally undersigned order and it is also repeated.

The container auto-scaler is leveraged to resolve the resource under-provisioning and/or overprovisioning across the different modes in the experiments. The results are shown in Figure 4. The upper threshold we set for CPU usage is 60%, while the lower threshold is 30%.

Figure 4(a) presents a 45-minute period single mode experiment. First, there is a trend of steep increase in CPU utilization at the beginning and it continues until around the nine minutes. Then a relatively small fluctuation shows up and then it is reduced to 70% after six minutes. According to the green line, auto-scaler decides to scale out after 2-minute waiting and the first instance is started in half a minute. We see that the number of containers increases with CPU workload intensity.

Experiments with longer testing periods are reported in Figure 4(b), presenting the repeated workload mode. This kind of workloads stresses the system in a recurring manner. A clear repeated pattern can be observed from both the computing resource utilization and the number of containers launched.

Shuffle workload with random strength shown in Figure 4(c), presents one single round of shuffle workload mode. The sequence of stresses is quite different from the single mode. Though the first fluctuate of the blue line is not as high as that in Figure 4(a), the auto-scaler still scales out as soon as possible. The CPU utilization of containers is more sensitive to the heavy workloads.

Figure 4(d) shows the results under the workload that combines the repeat mode and the shuffle mode. The red and blue lines are mostly overlapped. As the workload suddenly rises, the container auto-scaler kicks in after a short but reasonable response period. This fast response is mainly due to the fast deployment of containers. The benefit can be better seen in workload periods that are highly fluctuated.

Concluded above, we summarize the experience of testing under different workload modes in below as four different aspects:

- The container auto-scaler has a very fast response time when workload varies. Autoscaling, launching and deploying new containers show very short lag due to the light-weight feature of containerization.
- The variation of the added containers at the scaling-out phase closely reflects the variation of the computing resource utilization. Therefore, we can capture the resource utilization well during the experiments.
- Since we use *Stress* as the benchmark application, and it is mostly CPU intensive, the scaling patterns can be better seen in Figure 4, where the relationship of CPU usage and the container number variation are reported.

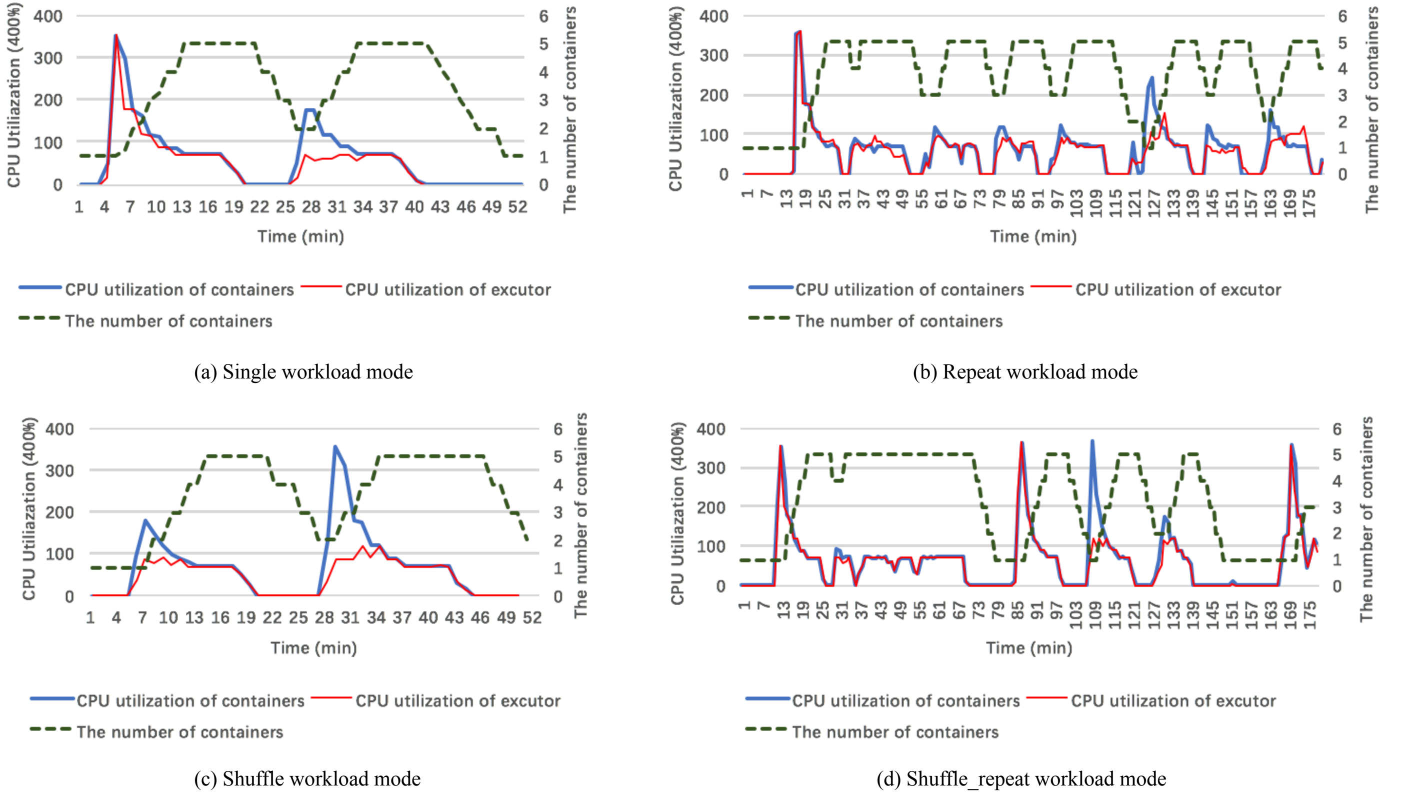
There is no noticeable distinction regarding the resource utilization between the executors and the containers, even though they behave slightly different when they are overloaded. Therefore, either of them can be used to reflect the real-time workload of the cloud platform.

TABLE IV. CALCULATION RESULTS

Metrics	Single	Repeat	Shuffle	Repeat & shuffle
$\overline{T_O}$ (min)	8.00	6.75	8.00	9.00
$\overline{T_I}$ (min)	9.50	6.14	9.00	7.40
$\overline{T}$ (min)	8.75	6.47	8.50	8.27
$\overline{S_O}$ (/min)	0.437	0.331	0.382	0.346
$\overline{S_I}$(/min)	0.369	0.356	0.325	0.361
$\bar{S}$(/min)	0.400	0.330	0.353	0.363
E_O(/min^2)	0.0546	0.0490	0.0478	0.0384
E_I(/min^2)	0.0388	0.0580	0.0361	0.0488
E(/min^2)	0.0457	0.0510	0.0415	0.0439

We also evaluate the elasticity in the subjective way introduced before and display experimental results in TABLE IV. As shown in Table IV, the average values of scaling time, scaling speed and elasticity are calculated. Firstly, the repeat mode has an obviously better elasticity measurement compared with other experiment modes no matter in over- or under-provisioned conditions, or in an aggregated elasticity. This is due to the fact that the repeat mode shows a known workload pattern, that allows the container auto-scaler to make easy recurring scaling decisions.

We can also see that the best scaling-out elasticity is seen in the single mode, while its scaling-in elasticity is relatively worse. The repeat&shuffle workload mode, due to its totally being unpredictable, has the worst scaling-in elasticity. This happens since the scaling-in action involves adding more containers, which usually takes longer time compared with the scaling-out actions, where the unwanted containers are simply killed.



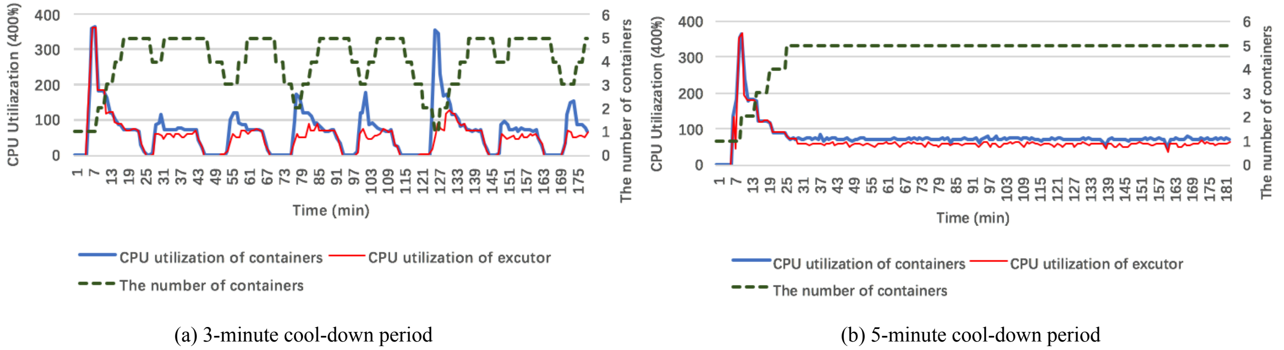
(a) Single workload mode

(b) Repeat workload mode

(c) Shuffle workload mode

(d) Shuffle_repeat workload mode

Fig. 4. The real-time CPU utilization and the number of containers in four different modes. The resource utilization of the Mesos slave and the varied number of the concurrent running containers managed by the container auto-scaler. The solid blue line shows the average resource utilization of the containers, and the red line is that of the executor that launches the containers on each slave node. We can see the repeat and cyclic pattern of the resource utilization, in accordance with the variation of the container numbers.



(a) 3-minute cool-down period

(b) 5-minute cool-down period

Fig. 5. Experimental results with varied cool-down periods by fixing the workload strength to five-minute duration under the repeat mode.

In conclusion, under the same configuration, the container auto-scaler does a better job under the repeat mode, in which workload shows a recurring pattern. For random workload patterns such as shuffle and repeat shuffle modes, elasticity is noticeably worse than the other two modes.

B. Impact of the Cool-down period

In this section, we study the relationship between the cool-down period and its impact on elasticity. We evaluated two different cool-down periods, three minutes (Figure 5(a)) and five minutes (Figure 5(b)). To make the experiment results fair, we fix the workload duration as five minutes.

In Figure 5(b), the CPU utilization line shows a sudden rise at the beginning then decreases gradually and finally is stabilized. The running containers start from one and finally step up to five and keep that size until the end of the experiment. The scaling-in action lasts for 17 minutes. Evaluated elasticity here is 0.007561437, 5 times lower than that in Figure 5(a), in which elasticity is 0.036572341. This is mainly caused by the shorter cool-down period leads to shorter scaling timespan (T_O and T_I), that inevitably leads to good elasticity.

However, the cool-down period shouldn't be too short in practice. That will cause the application in a consistent scaling mode, leading to instability and unpredictability of the real performance.

C. Discussions and Experience

We have gained lots of experience during the phase of testing the container auto-scaler. First, container auto-scaler captures the CPU computing resource changes and varies the container size accordingly. As a scaling decision maker and scaling request sender, it performs exactly the way it is expected.

Then, the container auto-scaler varies the container size in a very timely manner. From all the features above, we can see that very small delay is seen after the CPU utilization changes until it kicks in and the scaling phase starts. This is mainly due to the lightweight feature of containerization.

Finally, we learned that the cool-down period could be properly managed in order to balance a good system stability and platform elasticity.

VI. Conclusion

In this paper, we propose a lightweight container auto-scaler for elastic autoscaling on individual smart devices. The system architecture, together with the autoscaling details, including the monitoring mechanism, history recorder, decision mechanism and execution mechanism, are introduced. We have defined the concept of cloud elasticity under this scenario, and carried out a significant amount of experiments to study the advantages of the container auto-scaler, its use in measuring cloud elasticity under various experiment scenarios, and gained significant insight as to the critical factors to improve the cloud elasticity.

In future, we plan to extend the work in the following three aspects. First, we plan to use more experiment benchmarks on public cloud for more extensive evaluation. Second, we want to classify the workloads as being CPU-bound, memory-bound, disk-bound and a mixing of all. This helps us to characterize each benchmark application in more details and gain more insight in the scaling feature. Last but not least, we plan to present a proactive auto-scaler, which combines the reactive auto-scaler with a few mature prediction methods, to better scale the containers in terms of the count as well as their t-shirt sizes.

Acknowledgment

This study is funded by National Natural Science Foundation of China (41671441).

Samee U. Khan's work is supported by (while serving at) the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amazon AWS Services: https://aws.amazon.com/ (8.26.2017).

[2] IBM: https://www.ibm.com/ (8.26.2017).

[3] Microsoft Azure: https://azure.microsoft.com/en-us/?b=16.48 (8.26.2017).

[4] NR. Herbst, S. Kounev and R Reussner, “Elasticity in Cloud Computing: What it is, and What it is Not”, International Conference on Autonomic Computing, 2013, pp. 23-27.

[5] S. Islam, K. Lee, A. Fekete and A. Liu, "How a consumer can measure elasticity for cloud platforms," in ICPE, 2012.

[6] AuFS Wikipedia: https://en.wikipedia.org/wiki/Aufs (8.26.2017).

[7] T. C. Chieu, A. Mohindra, A. A. Karve, et al, “Dynamic scaling of web applications in a virtualized cloud computing environment”, IEEE International Conference on e-Business Engineering. IEEE, 2009, pp. 281-286.

[8] R. S. Sutton, A. G. Barto, Introduction to reinforcement learning, Cambridge: MIT Press, 1998.

[9] RightScale: http://www.rightscale.com/ (8.26.2017).

[10] P. Bodık, R. Griffith, C. Sutton, et al, “Statistical machine learning makes automatic control practical for internet datacenters”, Proceedings of the 2009 conference on Hot topics in cloud computing, 2009, pp. 12-12.

[11] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment”. Linux Journal, vol.239, March 2014.

[12] B. Simmons, H. Ghanbari, M. Litoiu, et al, “Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree”, Proceedings of the 7th International Conference on Network and Services Management. International Federation for Information Processing, 2011, pp. 343-347.

[13] Lorido-Botran, T., J. Miguel-Alonso, and J.A. Lozano, “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments.” Journal of Grid Computing, 2014. **12**(4): p. 559-592.

[14] Stress website: http://people.seas.harvard.edu/~apw/stress/ (8.26.2017).

[15] Mi, H., Wang, H., Yin, G., Zhou, Y., Shi, D., Yuan, L., “Online self-reconfiguration with performance guar- antee for energy-efficient large-scale cloud computing data centers.” In: 2010 IEEE International Conference on Services Computing (SCC), 2010, pp. 514–521.

[16] Roy, N., Dubey, A., Gokhale, A., “Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting.” In: 2011 IEEE 4th International Conference on Cloud Computing, IEEE, 2010, pp. 500–507. doi:10.1109/CLOUD.2011.42.

[17] Scheepers M J. “Virtualization and containerization of application infrastructure: A comparison.” 21st Twente Student Conference on IT. 2014, pp. 1-7.

[18] Amazon EC2 User Guide: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. (8.26.2017).

[19] Yu, H.E. and W. Huang, “Building a Virtual HPC Cluster with Auto Scaling by the Docker.” Computer Science, 2015.

[20] Hoenisch P, Weber I, Schulte S, et al. “Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers.”, International Conference on Service-Oriented Computing. Springer Berlin Heidelberg, 2015, pp. 316-323.

[21] DCOS: https://docs.mesosphere.com/ (8.26.2017).

[22] Marathon: https://github.com/mesosphere/marathon (8.26.2017).

[23] Gubbi J, Buyya R, Marusic S, et al. “Internet of Things (IoT): A vision, architectural elements, and future directions”. Future generation computer systems, 2013, 29(7), pp: 1645-1660.

.