Efficiency Analysis of Provisioning Microservices

Hamzeh Khazaei, Cornel Barna, Nasim Beigi-Mohammadi, Marin Litoiu
School of Computer Science, York University
Toronto, Ontario, Canada
Email: {hkh,cornel.barna,nbm,mlitoiu}@yorku.ca

***Abstract*—Microservice architecture has started a new trend for application development/deployment in cloud due to its flexibility, scalability, manageability and performance. Various microservice platforms have emerged to facilitate the whole software engineering cycle for cloud applications from design, development, test, deployment to maintenance. In this paper, we propose a performance analytical model and validate it by experiments to study the provisioning performance of microservice platforms. We design and develop a microservice platform on Amazon EC2 cloud using Docker technology family to identify important elements contributing to the performance of microservice platforms. We leverage the results and insights from experiments to build a tractable analytical performance model that can be used to perform what-if analysis and capacity planning in a systematic manner for large scale microservices with minimum amount of time and cost.**

I. Introduction

Infrastructure-as-a-Service (IaaS) cloud providers, such as Amazon EC2 and IBM Cloud deliver on-demand operating system (OS) instances in the form of virtual machines (VM). A virtual machine manager (VMM), or hypervisor, is usually used to manage all virtual machines on a physical machine. This virtualization technology is quite mature now and can provide good performance and security isolation among VM instances. An individual VM has no awareness of other VMs running on the same physical machine (PM). However, for applications that require higher flexibility at runtime and less isolation, hypervisor based virtualization might not satisfy the entire set of quality of service (QoS) requirements.

A *container* runs directly on a Linux kernel with similar performance isolation and allocation characteristics as VMs but without the expensive VM runtime management overhead [1], [2]. Containerization of applications, that is deployment of application or its components in containers, has become popular in cloud service industry. For example, Google is providing many of its popular products through a container-based cloud. A *Docker container* [3] comes with all dependent software packages for an application, providing a fast and simple way to develop and deploy different versions of the applications [3]. Container based services are popularly known as *Microservices* and are being leveraged by many service providers for a number of reasons: (1) to reduce complexity when using tiny services; (2) to scale, remove and deploy parts of the system or application easily; (3) to improve flexibility by using different frameworks and tools; (4) to increase the overall scalability; and (5) to improve the resilience of the system. Containers have empowered the usage of microservices architectures by being lightweight, providing fast start-up times, and having a low overhead [4].

9781-5090-1445-3/16$31.00 © 2016 IEEE (CloudCom'16)

A flexible computing model combines IaaS based clouds with container based PaaS (Platform-as-a-Service) cloud. Platforms such as Nirmata [5], Docker Cloud [6], previously known as Tutum, and Giant Swarm [7] offer platforms for managing virtual environments made of containers while relying on IaaS public/private clouds as the backend resource providers.

Service availability and service response time are two important quality measures from cloud's users perspective [8]. Quantifying and characterizing such performance measures requires accurate modeling and coverage of large parameter space while using tractable and solvable models in timely manner to assist with runtime decisions. This paper consider container based PaaS operating on top of VM based IaaS and introduces a performance model for microservice provisioning. The model supports microservice management use cases and incorporate the following contributions:

- Supports virtualization at both VM and container layers
- Captures different delays imposed by the microservice platform on users' requests;
- Characterizes the service availability and elasticity;
- Provides capacity planning and what-if analysis for microservice platform providers;
- Provides insights in performance vs cost trade offs.

The rest of the paper is organized as follows. Section II describes the new trend of emerging microservice platforms. Section III describes the details of the platform that we are going to model in this work. Section IV elaborate the performance analytical model. Section V and VI present the details of our experiments and the numerical results obtained from the analytical model. In section VII, we survey related work in cloud performance analysis, and finally, section VIII summarizes our findings and concludes the paper.

II. Microservice Platforms

Recently, a pattern has been adopted by many software-as-a-service providers in which both VMs and containers

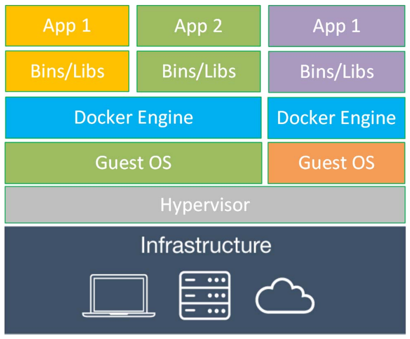


Fig. 1: Leveraging both virtualization techniques, ie, VMs and containers, to offer microservices on IaaS cloud.

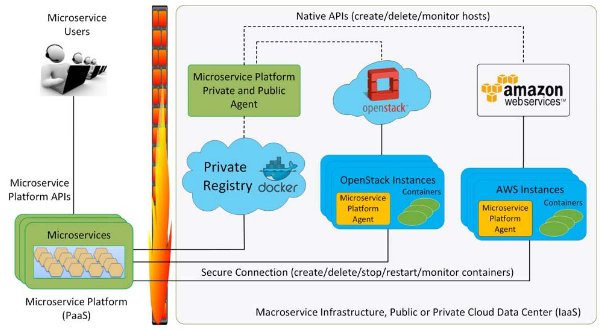

Microservice Users
Native APIs (create/delete/monitor hosts)
Microservice Platform Private and Public Agent
openstack
amazon webservices
Private Registry docker
OpenStack Instances
Microservice Platform Agent
Containers
AWS Instances
Microservice Platform Agent
Containers
Microservice Platform APIs
Microservices
Secure Connection (create/delete/stop/restart/monitor containers)
Microservice Platform (PaaS)
Macroservice Infrastructure, Public or Private Cloud Data Center (IaaS)

Fig. 2: Conceptual Model: including both microservice platform and the backend public/private cloud.

are leveraged to provide so called *microservices*. Microservices is an approach that allows more complex applications to be configured from basic building blocks, where each building block is deployed in a container and the constituent containers are linked together to form the cohesive application. The application's functionality can then be scaled by deploying more containers of the appropriate building blocks rather than entire new iterations of the full application. *Microservice platforms* (MSP) such as Nirmata [5], Docker Cloud [6] and Giant Swarm [7] facilitate the management of such service paradigm. MSPs are automating deployment, scaling, and operations of application containers across clusters of physical machines in cloud. MSPs enable software-as-service providers to quickly and efficiently respond to customer demand by scaling the applications on demand, seamlessly rolling out new features and optimizing hardware usage by using only the resources that are needed. Fig. 1 shows the layered architecture in which both virtualization techniques are leveraged to deliver microservices on the cloud.

Fig. 2 depicts the high-level architecture of MSPs and the way they leverage the backend public or private cloud (ie, infrastructure-as-a-service clouds). Various microservice platform providers such as Nirmata, Docker Tutum and Giant Swarm implemented their platform based on this conceptual model.

III. System Description

In this section we describe the system under modeling with respect to Fig. 3 that shows the servicing steps of a request in microservice platforms. In microservice platforms (MSP), a request may originate form two sources: first, direct requests from users (ie, microservice users in Fig. 2) that want to deploy a new instance of an application or service; second type would be runtime requests from applications (eg, consider adaptive applications) by which applications adapt to the runtime conditions; for example, scaling up the number of containers to cope with a traffic peak. We model these two types of requests altogether as a Poisson process.

The steps incurred in servicing a provisioning request in MSP are shown in the Fig. 3. User requests for containers are submitted to a global finite queue and then processed on a first-come, first-serve basis (FCFS). A request that finds the queue full, will be rejected immediately. Once the request is admitted to the queue, it must wait until the VM Assigning Module (VMAM) processes it. VMAM finds the most appropriate VM in the user's cluster (based on the policy) and then send the request to that VM's so that the Container Provisioning Module (CPM) initiates and deploys the container. when a request is processed in CPM, a pre-build or customized container image is used to create a container instance. These images can be loaded from a public repository like Docker Hub [3] or private repositories.

If there are not enough resources (ie, VM) in the MSP to accommodate the new request, scheduler asks for a VM from the backend IaaS (see Fig. 3). The request will be rejected if the application (or the user) has already reached its capacity. Otherwise, a VM will be provisioned and the last request for container will be deployed on this new VM. In the IaaS cloud when a request is processed, a pre-built or customized disk image is used to create a VM instance [8]. In this work we assume that pre-built images fulfill all user requests.

To model the behavior of this system, we design a provisioning performance model that includes all the servicing steps in fulfilling requests for containers. Then, we solve this model to compute the provisioning performance metrics: request rejection probability, mean response delay and cluster utilization as functions of variations in workload (request arrival rate), container lifetime and cluster size. We describe our analysis in detail in Section IV, using the symbols and acronyms listed in Table I.

IV. Analytical Model

The performance model of the microservice platform is shown in Fig. 4. The MSP performance model has been realized by a 3-dimensional Continues Time Markov Chain (CTMC) with states labeled as (i, j, k); state i indicates

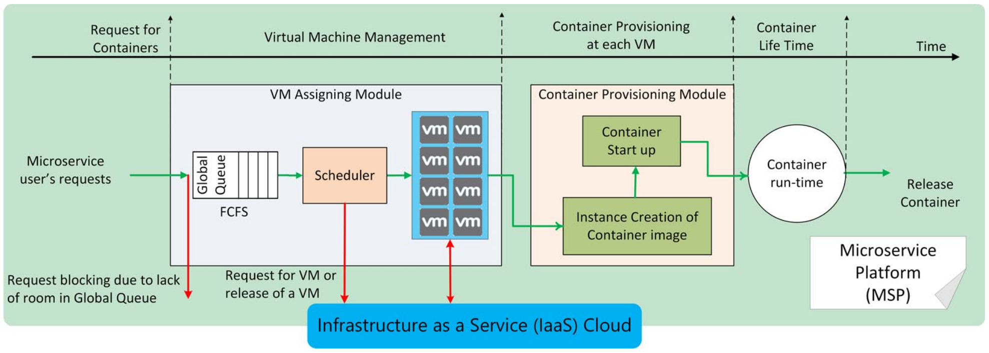

Request for Containers
Virtual Machine Management
Container Provisioning at each VM
Container Life Time
Time
VM Assigning Module
Container Provisioning Module
Microservice user's requests
Global Queue
FCFS
Scheduler
vm
Container Start up
Instance Creation of Container image
Container run-time
Release Container
Microservice Platform (MSP)
Request blocking due to lack of room in Global Queue
Request for VM or release of a VM
Infrastructure as a Service (IaaS) Cloud

Fig. 3: Servicing steps of a provisioning request in microservice platforms; derived form conceptual model in Fig. 2.

TABLE I: Symbols and corresponding descriptions.

Symbol	Description
λ	Mean arrival rate of requests for containers
$1/\alpha$	Mean time to provision a VM
$1/\beta$	Mean time that takes to decomission a VM
s	Min size of the cluster (VMs)
S	Max size of the cluster (VMs)
M	Max number of containers that can be deployed on a VM
$1/\mu$	Mean container lifetime
u	Utilization in the user's cluster
L_q	Microservice global queue size, $L_q = S \times M$
bp_q	Blocking probability in the microservice global queue
P_{req}	Probability of request for a VM
P_{rel}	Probability by which a VM will be released
λ_c	Arrival rate of requests for VMs
$1/\eta_c$	Mean VM lifetime
$1/\phi$	Mean time to provision a container
$\overline{wt_q}$	Mean waiting time in microservice global queue
$\overline{rt}$	Mean response time of a requests
$\overline{\text{vm}_{\text{no}}}$	Mean no of VMs in the cluster
$\overline{\text{cont}_{\text{no}}}$	Mean no of containers in the cluster
$\overline{\text{util}}$	Mean cluster utilization

the number of requests in Microservice Global Queue, j denotes the number of running containers in the platform and finally k shows the number of active VMs in the user's cluster. Each VM can accommodate up to M containers that is set by the user. The request arrival can be adequately modelled as a Poisson process [9] with the arrival rate of λ. Also let ϕ be the rate at which containers can be deployed on a VM and μ be the service rate of each container (note that $1/\mu$ would be the container lifetime). Therefor, the total service rate for each VM is the product of number of running containers by μ. Assume α and β are the rates at which the MSP can obtain and release a VM from IaaS cloud respectively. The scheduler asks for a new VM from backend IaaS cloud when explicitly ordered by the MSP user (or application) or when the utilization of the host group is equal or greater than a predefined value. For state (i, j, k), utilization u is defined as follows,

$$u = \frac{i + j}{k \times M} \tag{1}$$

in which M is the maximum number of containers that can be run on a single VM. On the other hand, if utilization drops lower than a predefined value, the MSP will release one VM to optimize the cost. A VM can be released if there is no running containers on it so that the VM should be fully *decommissioned* in advance. Also the MSP holds a minimum number of VMs (ie., s) in the cluster regardless of utilization, in order to maintain the availability of the microservices. The user may also set another value for its application(s) (ie, S) indicates the MSP can not request more than S VMs from IaaS cloud on behalf of the user. Thus the application scale up at most to S VMs in case of high traffic and scale down to s VMs in times of low utilization. We set the global queue size (ie, L_q) to the total number of containers that it can accommodate at its full capacity (ie, $L_q = S \times M$). Note that requests will be blocked if the user reached its capacity, regardless of Global Queue state.

State $(0, 0, s)$ indicates that there is no request in queue, no running container and the cluster consists of s VMs that is the minimum number of VMs that user maintain in its host group. Consider an arbitrary state such as (i, j, k), in which five transitions might happen:

1) Upon arrival of a new request the system with rate of λ moves to state $(i + 1, j, k)$ if the user still has capacity (ie, $i + j < S \times M$), otherwise the request will be blocked and the system stays in the current state.
2) A container will be instantiated with rate ϕ for the request in the head of Global Queue and moves to $(i - 1, j + 1, k)$.
3) The service time (ie, lifetime) of a containers finishes with rate of $k\mu$ and the system moves to $(i, j - 1, k)$.

4) If the utilization gets higher than the threshold, the scheduler requests a new VM, and the system moves to state $(i, j, k+1)$ with rate α.
5) Or, the utilization dropped bellow a certain value, MSP decommission a VM, and the system releases the idle VM so that moves to $(i, j, k-1)$ with rate β.

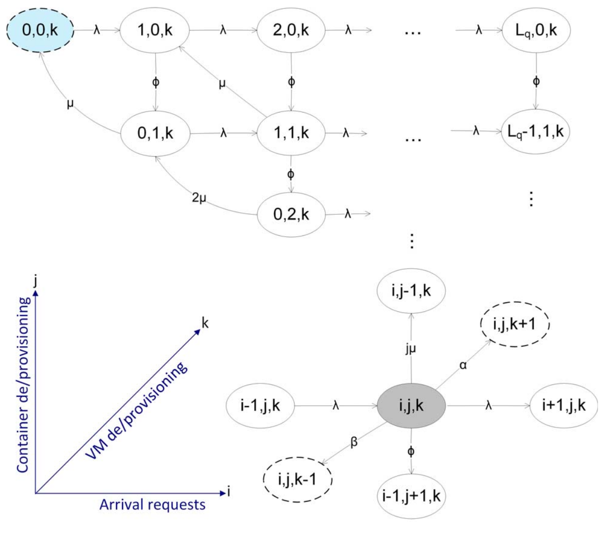


Fig. 4: Microservice platform model.

Suppose that $\pi_{(i,j,k)}$ is the steady-state probability for the model (Fig. 4) to be in the state (i, j, k). So the blocking probability in MSP can be calculated as follow,

$$bp_q = \sum_{(i,j,k)\in S} \pi_{(i,j,k)} \quad \text{if i + j} = L_q \tag{2}$$

We are also interested in two probabilities by which the MSP requests (p_{req}) or releases (p_{rel}) a VM.

$$P_{\text{req}} = \sum_{(i,j,k)\in S} \pi_{(i,j,k)} \quad \text{if} \quad u \geq \text{high-util} \;\&\; k < S \tag{3}$$

$$P_{\text{rel}} = \sum_{(i,j,k)\in S} \pi_{(i,j,k)} \quad \text{if} \quad u \leq \text{low-util} \;\&\; k > s \tag{4}$$

Using these probabilities, the rate by which microservice platform requests (ie, λ_c) or releases (ie, η_c) a VM can be calculated.

$$\lambda_c = \lambda \times p_{\text{req}} \tag{5}$$

$$\eta_c = \mu \times p_{\text{rel}} \tag{6}$$

In order to calculate the mean waiting time in queue, we first establish the probability generating function (PGF) for the number of requests in the queue [10], as

$$Q(z) = \sum_{i=0}^{L_q} (\pi_{(i,j,k)}) z^i \tag{7}$$

The mean number of requests in queue is

$$\overline{q} = Q'(1) \tag{8}$$

Applying Little's law [10], the mean waiting time in the global queue is given by:

$$\overline{wt_q} = \frac{\overline{q}}{\lambda(1 - bp_q)} \tag{9}$$

The total response time of a request would be summation of waiting time in queue, VM provisioning time from IaaS in case of need and container provisioning time at MSP. Thus, the response time can be calculated as:

$$\overline{rt} = \overline{wt_q} + (p_{req} \times \frac{1}{\alpha}) + (1/\phi) \tag{10}$$

We set α and ϕ based on our experiment on Amazon EC2 and Docker ecosystem which will be described in section V. The mean number of running VMs, mean number of running containers and mean utilization in the system can be calculated as follow:

$$\overline{\text{vm}_{\text{no}}} = \sum_{(i,j,k)\in S} k \times (\pi_{(i,j,k)}) \tag{11}$$

$$\overline{\text{cont}_{\text{no}}} = \sum_{(i,j,k)\in S} j \times (\pi_{(i,j,k)}) \tag{12}$$

$$\overline{\text{util}} = \sum_{(i,j,k)\in S} (\frac{i+j}{k \times M}) \times (\pi_{(i,j,k)}) \tag{13}$$

V. Experimental Setup and Results

In this section, we present our microservice platform and discuss experiments that we have performed on this platform. For experiments we couldn't use available third party platforms such as Docker Cloud or Nirmata as we needed full control of the platform for monitoring, parameter setting, and performance measurement. As a result, we have created a microservice platform from scratch based on the conceptual architecture presented in Fig. 2. We employed Docker Swarm as the cluster management system, Docker as the container engine and Amazon EC2 as the backend public cloud. We developed a front-end in Java for the microservice platform that interacts with the cluster management system (ie, Swarm) through REST APIs. The microservice platform leverages three initial VMs, two configured in *worker* mode and another one in *master* mode to manage the Docker Swarm cluster. All VMs are of type `m3.medium` (1 virtual CPU with 3.5 GB memory). In our deployment, we have used *Consul* as the discovery service, that has been installed on the Swarm Manager VM.

For the containers, we have used Ubuntu 16.04 image available on Docker Hub. Each running container was restricted to use only 512 MB memory, thus making the capacity of a VM to be 7 containers. The Swarm manager strategy for distributing containers on worker nodes was *binpack*. The advantage of this strategy is that fewer VMs can be used, since Swarm will attempt to put as many containers as possible on the current VMs before using another VM. Table II presents the input values for two *scenarios* in our experiments.

TABLE II: Range of parameters for experiments.

Parameter	Value(s)		Unit
	Scenario 1	Scenario 2	
Arrival rate	20	20 . . 40	req/min
VM capacity	7	7	container
Container lifetime	1	2	minute
Desired utilization	50% . . 80%	70% . . 90%	N/A
Initial cluster size	2	2	VM
Max cluster size	10	10	VM

To trigger VM elasticity, we have defined two cluster utilization (based on Eq. 1) thresholds: an upper threshold that signifies cluster is overloaded, and a lower threshold to show that the cluster is underloaded. In order to avoid oscillation or ping-pong effect in provisioning/releasing VMs, we do not add/remove a VM immediately after crossing thresholds rather we employ Algorithm 1 as the elasticity mechanism.

Algorithm 1: The decision making algorithm for adding and removing VMs.

```
if $utilization \geq upper\_threshold$ then
    // cluster overload
    if $heat < 0$ then
        // reset any buildup for VM removal
        $heat \leftarrow 0$;
    $heat \leftarrow heat + 1$;
else if $utilization \leq lower\_threshold$ then
    // cluster underload
    if $heat > 0$ then
        // reset any buildup for adding VM
        $heat \leftarrow 0$;
    $heat \leftarrow heat - 1$;
else
    // the utilization is within range
    // move heat one unit towards 0
    if $heat > 0$ then
        $heat \leftarrow heat - 1$;
    else if $heat < 0$ then
        $heat \leftarrow heat + 1$;
if $heat = 6$ then
    Add a new VM ;
    $heat \leftarrow 0$;
else if $heat = -6$ then
    Remove a VM ;
    $heat \leftarrow 0$;
```

In order to control experiment's costs, we have limited the cluster size to maximum of 10 running VMs for the application, which gives us a maximum capacity of 70 running containers. For the same reason, we set the container lifetime as 1 and 2 minutes for two scenarios. Under this configuration, our experiments take up to 1000 minutes combined (300 for the first scenario and 700 minutes for the second). The results of our experiments have been presented in Fig. 5. Note that, the X axis in Fig. 5 is experiment time in which we report the average values of performance indicators in every single minute; hereafter we call each minute of the experiments as *iteration*.

In the first experiment scenario (Fig. 5(a)), we have set the average lifetime of a container to one minute; the lower and upper utilization thresholds are set to 50% and 80% respectively (shaded area in the third plot of Fig. 5(a) shows the areas where the cluster is underloaded or overloaded). The arrival rate has a Poisson distribution with mean value of 20 requests per minute shown in the second plot of Fig. 5(a) with blue line. In the first plot, red line shows the number of running VMs and the blue line enumerates the number of running containers.

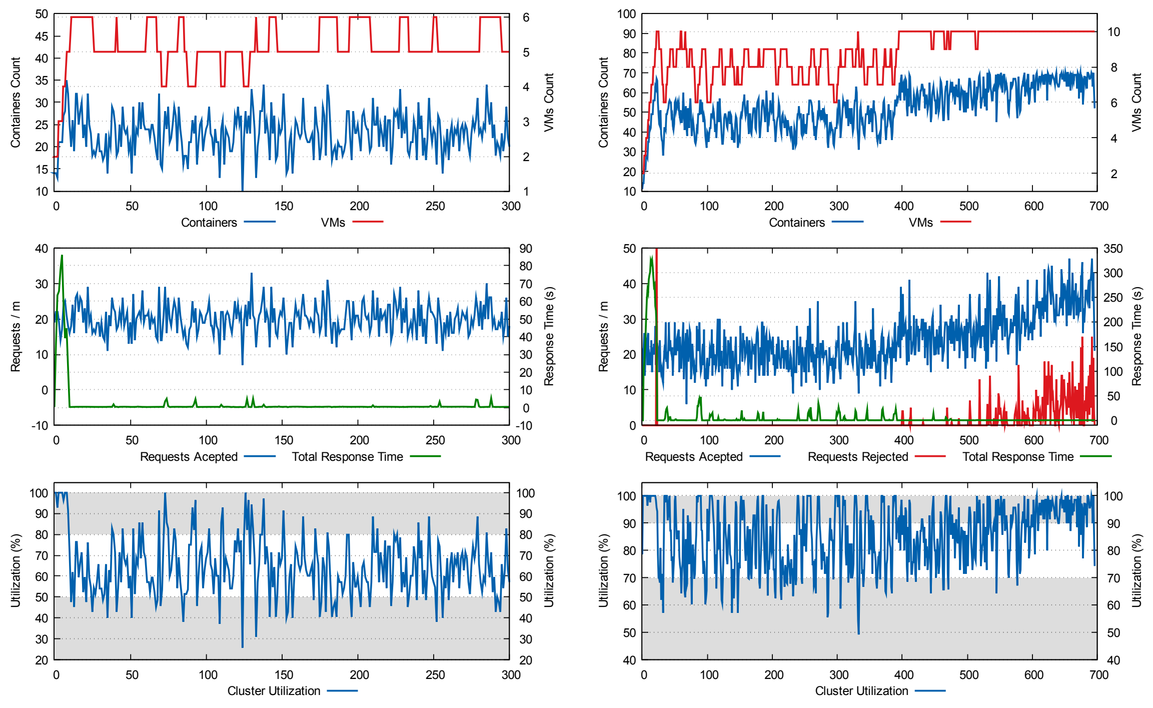
An interesting observation here is the behavior of the system at the beginning of the experiment. Because the workload is very high and the capacity of the cluster is low, the response becomes very long (ie, up to approximately 85 s) that is attributed to long waiting time in the queue for capacity to become available. Once enough VMs have been provisioned, the queue empties and the response time drops. Occasional spikes appear in the response time when there is no available capacity (and a new VM has to be added to the cluster), but they are less severe. All in all, in scenario 1 the system is operating well with desired utilization and response time with no rejected request.

In the second experiment scenario, presented in Fig. 5(b), we have increased the lifetime of a container to two minutes and changed the cluster utilization thresholds to 70% and 90%; we also increased the arrival rate from 20 req/min to 40 req/min around iteration 400. The other parameters of the experiments remained the same. We noticed the same high response time at the beginning of the experiment; this time the cluster scaled up to 10 VMs (maximum allowed number) while still the queue did not get cleared; this is attributed to longer lifetime of containers that makes resource releasing slower compared to the first scenario. At this moment, because maximum capacity of the cluster has been reached, we have witnessed a large number of rejected requests (around iteration 20, the red line in the second plot). After 50 iterations the behaviour of the cluster is similar to that of scenario 1. Around iteration 400, we started to increase the workload (blue line in the second plot, Fig. 5(b)). This resulted in eventually utilization of all allowed VMs and rejection of requests as there was no capacity available.

VI. Numerical Validation and Analysis

In this section, We first validate the analytical model with results of experiments presented in section V. Second, thanks to minimal cost and runtime associated with analytical performance model, we leverage it to study interesting scenarios at large scale with different configuration and parameter settings to shed some light on MSP provisioning performance.

We use the same parameters, outlined in Table II, for both experiments and numerical analysis. The analytical model has been implemented and solved in Python using NumPy,



(a) Scenario 1.

(b) Scenario 2.

Fig. 5: Experimental results; see Table II for parameter settings.

SciPy, Sympy and Matplotlib libraries [11]. Table III shows the comparison between the results from analytical model and experiment for both scenarios. As can be seen, both analytical models and experimental results are well in tune with differences less than 10%. Note that in analytical model for the sake of equilibrium conditions (ie, steady state) we put a limit on queue size while we do not have such a limitation in the experiment. Also, in the experiments we employ a more sophisticated elasticity policy (ie, Algorithm 1) compared to simple threshold approach in analytical model. These two differences (ie, queue limit and elasticity policies) are the source of narrow divergence between analytical model and the experiments.

TABLE III: Corresponding results from analytical model (AM) and experiment (Exp) for both scenarios.

Parameter	Scenario 1		Scenario 2	
	AM	Exp	AM	Exp
Response Time	2.115	2.233	2.98	3.168
Utilization	63.3%	64.7%	79.5%	82.4%
Mean No of VMs	4.6	5.15	7.33	7.85
Mean No of Containers	20	22.1	39.99	44.2

Now that the analytical model captures the microservice platforms accurately enough, we leverage it to study interesting scenarios at large scale. Note that analytical model take less than a minute to conclude while in experiment we needed around 1000 minutes to compute desired performance indicators; more importantly, analytical model cost is negligible compared to the cost of experiments. Table IV presents the range of individual parameter values that we either set or calculate from our experiments for numerical analysis of the analytical model.

Microservice applications may span across large number of VMs and, as a result, incorporate a very large number of containers. Also, container may live longer depending on the applications; for example consider an email application, which has been deployed as microservices, that spin a container for each active user; this container serves the user while he/she is checking or composing emails and will be released when the user logs out of the system [12]. Motivated by these facts we employed analytical model to investigate the microservice provisioning performance under such an assumptions. In general, as the analytical model scale linearly[1], with increasing input parameters such as number of VMs and the number of containers per VMs, it can be leveraged to study microservice platforms with large scales.

We let the mean container lifetime to be in the range of [8 . . 20] minutes; also we set various cluster size to include [28 . . 44] VMs. Under this parameter setting we

[1]We described a formal approach to prove linear scalability for performance models in [13]. It can be used to prove the linear scalability of the proposed performance model in this paper as well.

TABLE IV: Range of parameters for the analytical model.

Parameter	**Value(s)**	**Unit**
Arrival rate	16	request/min
VM capacity	7	container
Mean container provisioning time	0.435*	second
Mean VM provisioning time	104.6*	second
Container lifetime	8 . . 20	minute
Desired utilization	70% . . 90%	N/A
Initial cluster size	2	VM
Max cluster size	28 . . 44	VM
*These values have been obtained from experiments.		

obtained the interested performance indicators. Fig. 6(a) shows the trend of total response time when changing above control variables. As can be seen, in order to fulfill a request under 5 seconds, the container lifetime should not exceed 12 minutes and the cluster should include at least 40 VMs. Also, it can be noticed that none of the clusters can maintain response time lower than 25 seconds when the mean lifetime of containers is 20 minutes on average.

Fig. 6(b) shows the probability by which a request may get rejected due to either lack of room in the global queue or lack of capacity in the microservice platform. A linear relationship can be noticed between the container lifetime and the rejection probability. Also, for keeping the rejection below 5%, the application should employ at least 40 VMs.

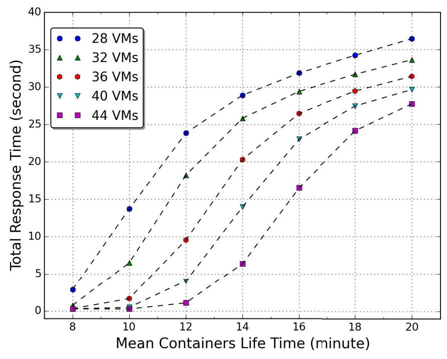
We also characterize the utilization of the cluster under various cluster size and container lifetime. As can be seen in Fig. 6(c), we set the desired utilization between 70% and 90%. If the mean container lifetime is 8 minute, regardless of the cluster size, the utilization would be less than 70% which economically is not desirable. On the other hand, for containers with average lifetime of 20 minutes, the utilization of the cluster would be more than 90% which is not desirable for the sake of performance and reliability.

In addition to results presented here, we also characterized the response time, rejection probability, utilization, number of VMs and number of containers for different arrival rate of requests which have been omitted due to page limit.

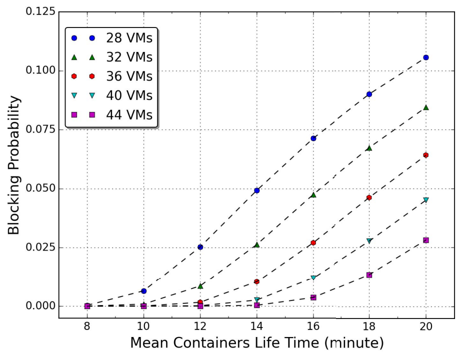
VII. Related Work

Performance analysis of cloud computing services has attracted considerable research attention although most of the works considered hypervisor based virtualization in which VMs are the sole way of providing isolated environment for the users. However, recently, container based virtualization is getting momentum due to its advantages over VMs for providing microservices.

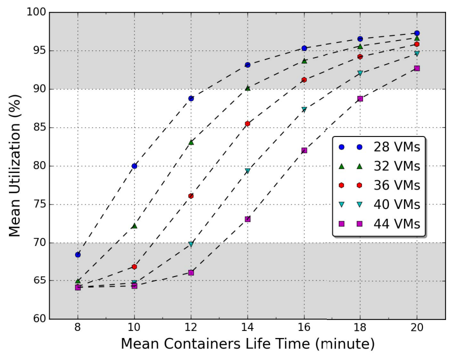
Performance analysis of cloud services considering containers as a virtualization option is in its infancy. Much of the works have been focused on comparison between implementation of various applications deployed either as VMs or containers. In [14], authors showed that containers have outperformed VMs in terms of performance and scalability.



(a) Total response time for requests.



(b) Blocking probability of requests.



(c) Mean utilization of clusters.

Fig. 6: Analytical model results; see Table IV for input parameters.

Container deployment process 5x more requests compared to VM deployment and also containers outperformed VMs by 22x in terms of scalability. This work shows promising performance when using containers instead of VMs for service delivery to end users.

Another study has been carried out in [15] to compare the performance of three implementations of an application using native, Docker and KVM implementation. In general, Docker equals or exceeds KVM performance in every case. Results showed that both KVM and Docker introduce negligible overhead for CPU and memory performance.

The authors in [16] performed a more comprehensive study on performance evaluation of containers under different deployments. They used various benchmarks to study the performance of native deployment, VM deployment, native Docker and VM Docker. All in all, they showed that in addition to the well-known security, isolation, and manageability advantages of virtualization, running an application in a Docker container in a vSphere VM adds very little performance overhead compared to running the application in a Docker container on a native OS. Furthermore, they found that a container in a VM delivers near native performance for Redis and most of the micro-benchmark tests.

Amaral et al. [4] evaluated the performance impact of choosing between the two models for implementing related processes in containers: in the first approach, master-slave, all child containers are peers of each other and a parent container which serves to manage them; in the second approach, nested-container, involves the parent container being a privileged container and the child containers being in its namespace. Their results showed that the nested-containers approach is a suitable model, thanks to improved resource sharing and guaranteeing fate sharing among the containers in the same nested-container.

These studies reveal a promising future for using both virtualization techniques in order to deliver secure, scalable and high performant services to the end user [17], [18]. The recent popularity of microservice platforms such as Docker Cloud, Nirmata and Giant Swarm are attributed to such advantages mentioned above. However, to the best of our knowledge, there is no comprehensive performance model that incorporates the details of provisioning performance of microservice platforms. In this work, we studied the performance of PaaS and IaaS collaborating with each other to leverage both virtualization techniques for providing fine-grained, secure, scalable and performant microservices.

VIII. Conclusions

In this paper, we presented our work on provisioning performance analysis of microservice platforms. We first designed and developed a microservice platform on Amazon cloud using Docker family technologies. We performed experiments to study the important performance indicators such as total request response time, request rejection probability, cluster size and utilization while capturing the details of provisioning at both container and VM layers. Then we developed a tractable analytical performance model that showed a high fidelity to experiments. Thanks to linear scalability of the analytical model and realistic parameters from experiments, we could study the provisioning performance of microservice platforms at large scale. As a result, by leveraging proposed model and experiments in this paper, what-if analysis and capacity planning for microservice platform can be carried out systematically with minimum amount of time and cost.

Acknowledgments

We would like to thank Dr. Murray Woodside for his valuable technical comments and inputs. This research was supported by Fuseforward Solutions Group Ltd., the Natural Sciences and Engineering Council of Canada (NSERC), and the Ontario Research Fund for Research Excellence under the Connected Vehicles and Smart Transportation (CVST) project.

References

[1] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 275–287.

[2] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.

[3] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[4] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*, Sept 2015, pp. 27–34.

[5] Nirmata, Inc. (2016, 6) Microservices Operations and Management. [Online]. Available: http://nirmata.com

[6] Docker Cloud. (2016, 6) The Docker Platform for Dev and Ops. [Online]. Available: https://cloud.docker.com

[7] O. Thylmann. (2016, 6) Giant Swarm Microservices Framework. [Online]. Available: https://giantswarm.io

[8] H. Khazaei, J. Mišić, and V. B. Mišić, "A fine-grained performance model of cloud computing centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2138–2147, November 2013.

[9] G. Grimmett and D. Stirzaker, *Probability and Random Processes*, 3rd ed. Oxford University Press, Jul 2010.

[10] L. Kleinrock, *Queueing Systems, Volume 1, Theory*. Wiley-Interscience, 1975.

[11] SciPy. (2016, 6) A python-based ecosystem of open-source software for mathematics, science, and engineering. [Online]. Available: http://scipy.org

[12] J. Beda. (2015, 05) Containers at scale: the Google Cloud Platform and Beyond. [Online]. Available: https://speakerdeck.com/jbeda/containers-at-scale

[13] H. Khazaei, J. Mišić, V. B. Mišić, and S. Rashwand, "Analysis of a pool management scheme for cloud computing centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 849–861, 2013.

[14] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. IEEE, 2015, pp. 342–346.

[15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[16] VMware, Inc. (2016, 6) Docker containers performance in vmware vsphere. [Online]. Available: https://blogs.vmware.com/performance/2014/10/docker-containers-performance-vmware-vsphere.html

[17] U. Gupta, "Comparison between security majors in virtual machine and linux containers," *arXiv preprint arXiv:1507.07816*, 2015.

[18] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference*. IEEE, 2015, pp. 583–590.