2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)

SmartVM: A Multi-Layer Microservice-Based Platform for Deploying SaaS

Xi Zheng*‡, Jiaojiao Jiang†, Yuqun Zhang*, Yao Deng‡, Min Fu§¶, Tianlei Zheng‖, and Xiao Liu‡
*Shenzhen Key Lab of Computational Intelligence, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
†School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia
‡School of Information Technology, Deakin University, Melbourne, Australia
§Department of Computing, Macquarie University, Sydney Australia
¶Data61, CSIRO, Sydney Australia
‖School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia

***Abstract*—With the advent of Software-as-a-Service (SaaS), SaaS developers are facing many challenges associated with the multi-tenancy and the dramatically increased number of users. In order to achieve resource-optimized, on-demand dynamic scaling across multiple tenants, and reduce costs, in this paper, a new platform, named SmartVM, is created to enable SaaS developer to create, customize, and deploy SaaS solutions in a multi-tier microservice-based manner. We develop an e-commerce SaaS prototype to evaluate effectiveness and efficiency of SmartVM. The results show that the SmartVM deployments outperforms the conventional monolithic and microservice deployments in smart monitoring, cost reduction, and resource optimization.**

I. Introduction

In recent years, SaaS (Software as a Service) has become popular because it can significantly bolster software applications such that they do not have to be developed through a long lifecycle as on-premise development. Nowadays, SaaS developments are increasingly conducted on top of microservices and deployed on Docker containers, such as Jenkins [1]. Many challenges for SaaS developments arise for the applications featuring being real-time, automated, or batched where a minor change might risk breaking the critical business processes. These challenges include security and privacy, scalability, resource optimization, availability and fault tolerance, and cost [2].

Extensive research has focused on those challenges and some well-known practices have also been concluded to deal with them. However, they are not always suitable for configuration managements in SaaS environments for the following reasons.

- SaaS deployment is often highly costly due to the unscalable usage of containers [3].
- Many SaaS applications do take timing constraints as non-functional requirements that can be easily violated under insufficient system resources.
- Conventional Docker monitoring tends to be expensive by not taking business constrains (user specification, etc.) and container utilization into consideration and leads to resource inefficiency.

The work was supported by Shenzhen Science and Technology Innovation Commission grant ZDSYS 201703031748284.

In this paper, a new framework, namely SmartVM, is proposed to bridge the gap between the best practices and the real-world adoptions. SmartVM aims at resolving the issues of scalability, monitoring, and cost in multiple-tenant SaaS developments and deployments. In particular, SmartVM enables a multi-layer structure that categorizes microservices to be business and API microservices, where each of them is designed with a separate load balancer. The higher-tier business microservices is designed for implementing business logics, and the lower-tier API microservices is designed for implementing resource-aware APIs (*e.g.*, to access database, to send emails, to generate PDFs), which are further recognized to be CPU-, Memory-, and/or IO- intensive.

To evaluate the efficacy of SmartVM on SaaS deployment, we conduct a set of experiments for comparing SmartVM deployment against two benchmarks : (1) the monolithic deployment; (2) the uniform microservice deployment. More specifically, the evaluation is conducted in terms of the following metrics:

- **Resource metric** measures the number of the invoked Docker containers.
- **Performance metrics** measure the CPU, Memory and I/O usage.
- **Business Constraint** measures the occurrence of business violations or request timeouts.

The experiment results show that SmartVM outperforms the conventional monolithic and uniform microservice deployments. In particular, SmartVM presents higher CPU, Memory and I/O resource usage, fewer invoked Docker containers, and fewer business violations. The experimental results also suggest that Docker containers can achieve desirable performance in terms of CPU workload and file I/O, which validates the advantages of SmartVM.

The rest of this paper is organized as follows. Section II introduces some related work of the proposed prototype. Section III presents the proposed solutions of SmartVM. Section IV evaluates the performances of SmartVM and compares it

0-7695-6329-5/17/31.00 ©2017 IEEE
DOI 10.1109/ISPA/IUCC.2017.00077

IEEE computer society

Authorized licensed use limited to: Macquarie University. Downloaded on December 10,2020 at 16:41:20 UTC from IEEE Xplore. Restrictions apply.

with the conventional monolithic and uniform microservice deployments. Section V concludes the remarks in this paper.

II. Related Work

Deploying SaaS has been a long-term research topic [4]. Switching to microservice-based deployments has been used to solve numerous issues with traditional SaaS deployment. However, there are still many issues with deploying SaaS, such as scalability, monitoring, and cost. In this section, we present the related work of those issues along with two conventional deployments: the monolithic deployment and the uniform microservice deployment.

Monolithic Deployment Dragoni et al. [5] defines a monolithic system as a system which cannot be expressed into independently executable modules or services. In monolithic deployments, in order for a component to scale, a host has to scale out/up and leads to decreased efficiency. It can be observed that it ignores business requirement monitoring. Moreover, it generates new Docker container(s) whenever CPU, memory or I/O usage violation occur (*e.g.*, the occurrence exceeds the preset threshold) . Hence, cost inefficiency can be caused.

Uniform Microservice Deployment Thnes et al. [6] defines microservices as small applications which can be deployed independently, scaled independently and has only a single task or function to accomplish. In this way, a distributed system consists of numerous independent microservices . In a uniform microservice deployment where each component is attributed into an independent service using containers, hosts can shuffle the containers amongst them and scale out/up. Dragoni et al. [5] demonstrates that scaling a microservice architecture does not imply duplicating its components. Instead, instances of services can be deployed and disposed according to load requirement. However, similar to the monolithic deployment, the uniform microservice deployment does not consider the business requirement violation or cost issues.

Cost Espadas et al. [3] introduces a tenant-based model to tackle over- and under- utilization of resources. The model essentially isolates the execution of each tenant by assigning it a tenant-based load balancer which distributes requests based on tenant information. In addition, a tenant-based VM instance allocator determines the number of instances required for a certain workload or task. Experimental results suggest a general reduction in over and underutilization; however, statistical testing using t-test indicated that averages only for underutilization were statistically improved. Khazaei et al. [7] provides an efficiency analysis of the provisioning of microservices on the cloud using Amazon EC2 and Docker containers. Major gains can be obtained in efficiency and resource optimization compared to a single system deployment. A common challenge is developing a cost model, which calculates cost based on resource usage instead of resource allocation.

Scalability Traditional deployments lack the ability to scale in a non-uniform manner [5], [8]. According to [9], the effectiveness of provisioning of services on the cloud is dependent on the number and location of service facilities deployed at various hosts. For example, when monolithic systems are exposed to increasing load, it is difficult to isolate out which component is stressed since the whole system runs as a single process. Thus, although only a single component may be experiencing an increased load, the whole system needs to scale out. Smaragdakis et al. [9] propose a solution to migrate, add or remove servers within a limited network scope, by using only the local network and topology information. The current methods to determine such information rely on data centralization, gathering, and transmission of entire network information, before deciding. This is impractical in large networks and result in degraded performance.

Monitoring Stubbs et al. [10] claim that monitoring is easier in a uniform and consolidated system compared to a system, which is comprised and built by interconnecting numerous functional units. Consequently, not only do we have to deal with increased overhead when dealing with a distributed system, but also take into consideration the messaging queues and the work flow systems in place. Stubbs et al. [10] proposed a Serfnode solution aiming to deal with the problem of service discovery while monitoring micro services running in containers such as Docker. Later, Fazio et al. [11] highlight the difficulty in monitoring and scheduling a microservice based SaaS deployment, and emphasize on the need for custom monitoring tools developed using cluster wide frameworks such as Apache Spark, which could be implemented along with general orchestration frameworks such as Kubernetes. However, they fail to provide a demonstrable proposition.

Based on the analysis above, a smart platform is commandingly required to reduce cost, automate scaling, and finally improve the SaaS deployment.

III. SmartVM Platform

In this section, we first introduce all the components involved in SmartVM. Then, we explore the detailed solutions of the SmartVM. Finally, a comparison between the SmartVM deployment and the conventional monolithic and uniform microservice deployment is presented. GlobalMonitor

The proposed SmartVM platform targets the mainstream SaaS deployment practice, which is associated with Docker containers that gains wide-spread popularity from many companies including Amazon and Google [13]. SmartVM is designed to enable multi-tier microservices and the associated load balancers (*i.e.*, two tiers in this paper), global monitors, and distributed monitors. Fig. 1 demonstrates the working and process flows of the SmartVM platform when deployed with a microservice-based application using Docker containers. All the components of SmartVM can be deployed with different granularity (*e.g.*, VM level or Docker container level). The following lists all the components involved in the deployment architecture.

- **Business Microservice (BMS)** The Business Microservice (BMS) implements the business logics. For each BMS, SmartVM provides a BMS service, which is responsible for messaging and communicating between

Authorized licensed use limited to: Macquarie University. Downloaded on December 10,2020 at 16:41:20 UTC from IEEE Xplore. Restrictions apply.

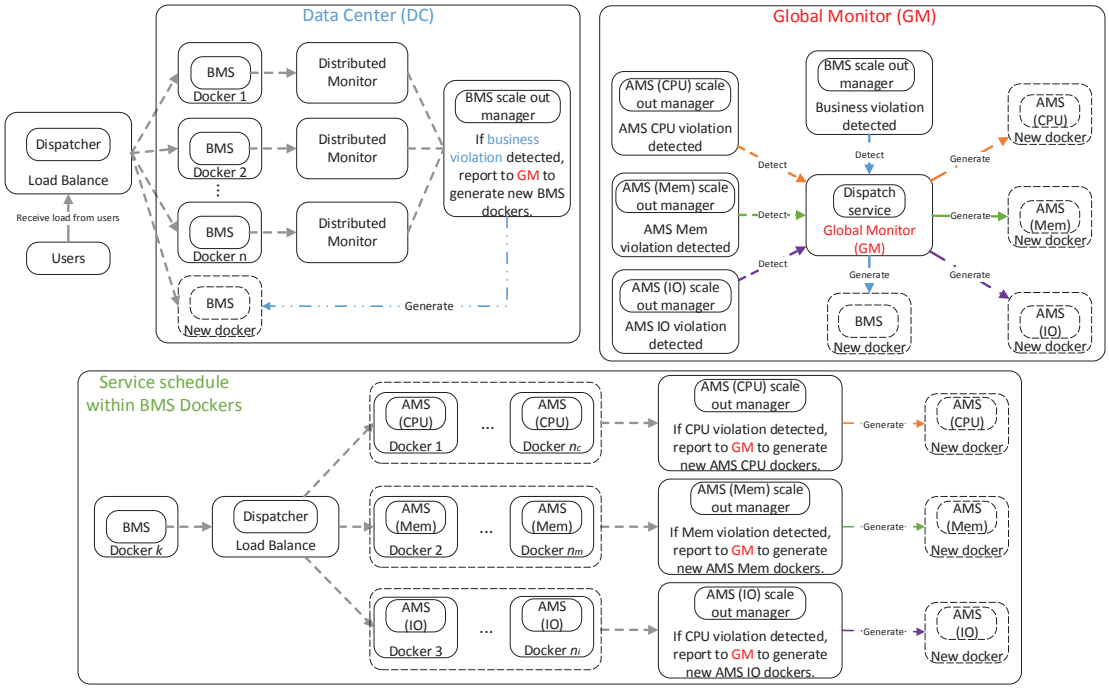


Fig. 1: The proposed SmartVM Platform.

BMSs and the underlying API Microservices (AMS), and between BMSs and distributed monitors (DMs). For each BMS, SmartVM also provides a Docker container to host it and its associated composed service.

- **API Microservices (AMS)** The API Microservice (AMS) implements the resource-aware library functions. It can be further separated into CPU-, I/O- or memory- intensive AMS. For each AMS, SmartVM allocates an AMS service, which is responsible for messaging and communicating between AMSs and BMSs, and between AMSs and distributed monitors (DMs). Each AMS runs its independent process using a Docker container, i.e., each AMS runs in a separate Docker container.
- **Load Balancers** In SmartVM, we design load balancers of two tiers. The first-tier load balancer adopts the concepts of the traditional load balancing which serves as a gateway between users and the underlying applications. The first-tier load balancer employs a dispatcher that is connected with Global Monitor and BMS. The dispatcher keeps track of how many BMS containers (*i.e.*, the Docker containers hosting BMSs) are available and implements a round-robin fashion load balancing mechanism for simplicity and efficiency. It acts as an API gateway to balance loads between the user requests and microservices. The dispatcher receives the tasks from the users (in this paper, the users request are simulated as loads by using a load generator) and pushes them into a queue. The requests in the queue are then distributed to BMS containers for processing. The second-tier load balancer is used between BMS and AMS. Each BMS can invoke multiple resource-aware AMSs, the requests from BMSs to AMSs are handled similarly with the first-tier load balancer by a dispatcher. However, the dispatcher is designed in a different manner where it can be customized and optimized for the specific deployment environment between BMS containers and AMS containers (*i.e.*, the Docker containers hosting AMSs). In this paper, the dispatcher for the second-tier load balancer is designed in a round-robin fashion such that it can be fully aware of how many AMS containers are available and connected to the global monitor (GM).
- **Global Monitor** The global monitor (GM) primarily aims at connecting each distributed monitor to keep track of the status of AMS and BMS containers. Additionally, a GM is responsible for scaling AMS or BMS containers to new containers in the existing cluster of VMs or a new cluster of VMs when detecting resource usage violations for AMSs (*e.g.*, CPU/Memory/IO usage exceeds preset thresholds) or BMSs (*e.g.*, violating preset timing constraint). Specifically, the GM collects status and resource usage data of all containers on each VM. Depending on these data, GM controls when to scale container, what type of container to scale and what VM the container will scale in. Then it send orders to DM to implement scaling. Ideally, we envision the availability of a cluster of GMs, with one or multiple GMs dedicated to a cluster of docker containers reserved for AMS and BMS. In this paper, a GM is created as a single instance for simplicity and it is deployed in the VM (*i.e.*, control VM) that contains first-tier load balancer and application webserver.

Authorized licensed use limited to: Macquarie University. Downloaded on December 10,2020 at 16:41:20 UTC from IEEE Xplore. Restrictions apply.

- **Distributed Monitor** The distributed monitor (DM) can be deployed to either AMS/BMS containers or individual VMs(*i.e.* work VMs) according to the factors, such as complexity of AMS and BMS, physical environment settings (*e.g.*, whether the target SaaS application needs to be deployed to different physical data centers, the available bandwidth). In this paper, we deploy one DM for every single container. For each BMS, a DM keeps track of the occurrence of business violations. When it exceeds a preset threshold (*i.e.,* 5 in this paper), the DM would send requests to the GM for scaling the associated BMS. For each AMS, a DM keeps track of system resource usage (*e.g.*, CPU, Memory), and when the occurrence of resource violations exceeds a preset threshold(*i.e.,* 3 in this paper), the DM would send requests to the GM to scale out for the associated AMS.
- **Scale Out Manager** In SmartVM, a BMS scale out manager is designed and deployed to every single VM. A BMS scale out manager is connected to a GM to generate requested BMS containers (in this paper, each time the scale out command generates one Docker container) and reports to the GM the unique identifier of the newly generated BMS containers. Similarly, an AMS scale out manager is designed and deployed to every single VM in a similar manner where it is connected to GM to generate AMS containers as requested. Note that in this paper we only focus on the scenarios that need scaling out. The scale out manager efficiently executes scale-out script to generate new containers and report their registry information to GM. The DM monitors business violation of BMS and usage violation of AMS in a host VM. It reports violation data to GM when violation is detected. The GM recieves violation information from DM and scale out manager. In the future work we will make these information visualized. After analyzing these information, the GM generates scale-out order and sends it to specific host VM.

IV. Evaluations

We conduct a set of experiments that evaluates the efficacy of the monolithic deployment, the uniform microservice deployment, and the SmartVM deployment. The evaluation simulates a typical retail SaaS application that receives and processes user requests. The evaluation application can simulate various retail business functions such as browsing through shopping items and checking out. We generate the user requests through a real-time load generator based on user number and the deployment type. The real-time load generator can simultaneously simulate a specific number of active users to send requests to the evaluation application over a period of time. For each user request, the monolithic deployment platform has exactly one request. The application, upon receiving per-user request, executes 30 distinct business functions. For each user request, the microservice deployment issues 30 distinct business function calls. For each user request, the SmartVM deployment also issues 30 distinct business service calls. For each business function/service, at least one of CPU-, memory-, or I/O-intensive API calls is required. .

The experiment aims at collecting statistics of the following metrics

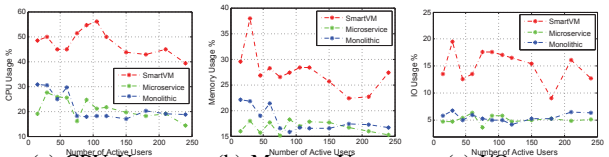
- Resource Utilization (CPU, Memory, I/O)
- Number of active microservices (reflected by containers) at a given time.
- Occurrence of business violations (request timeouts)

A. Experiment Settings

The deployment environment consists of 15 Virtual Machines or hosts running CentOS7 Linux. Each VM has the following configuration: a. Hypervisor VMware; b. CPU Intel Xeon E5-2690v4 @ 2.60Ghz, 2 Cores; c. RAM2vGB; d. HDD64 GB. The 15 VMs are deployed on 4 physical nodes. One physical node supports three to 4 VMs. Each VM runs on the same VLAN with four 10GbE nertwork interfaces.

B. Evaluation Applications

To simulate a typical scenario, the deployment is tested as three evaluation applications: 1) Evaluation application 1 (E1) simulates a *monolithic* deployment; 2) Evaluation application 2 (E2) is the E1 rebuilt in the uniform *microservice* deployment; 3) Evaluation application 3 (E3) simulates the proposed *SmartVM* deployment. The user requests (loads) are simulated by a load generator.



(a) CPU Usage (b) Memory Usage (c) I/O usage

Fig. 2: Resource Usage

C. Resource Usage

Figs. 2a, 2b, and 2c show the statistics of average CPU usage, average memory usage and average I/O usage for the three evaluation applications. In general, the proposed SmartVM architecture outperforms the conventional monolithic and microservice-based architectures.

More specifically, from Fig. 2a, we can see that with the increase of the number of active users, the average CPU usage of E1 is similar to the average CPU usage of E2. Their CPU usage lies between 10% and 20%. In contrast, the usage of CPU is much higher in E3, which is generally above 30%. In particular, the fewer active users, the higher usage of CPU. When the number of users is fewer than 50, the average CPU usage is around 50%. The reason of the situation is that, in E1 there is only one CPU container at first. During the increase of active users and requests, CPU usage in the container increases and reaches the scale limit. Then new CPU containers will be scaled out, which leads to the decrease of the average CPU usage of all CPU containers. The average usage of CPU in E1 is much higher than E2 and E3, which means that E1 can make use of CPU resource better and has higher efficiency than E2

Authorized licensed use limited to: Macquarie University. Downloaded on December 10,2020 at 16:41:20 UTC from IEEE Xplore. Restrictions apply.

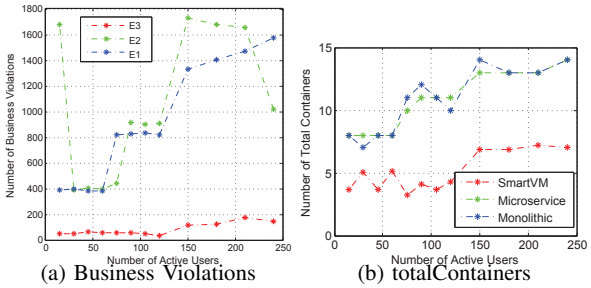
and E3. Therefore, this justifies that the proposed SmartVM deployment optimizes the CPU usage.

From Fig. 2b, it can be observed that E1 and E2 present similar memory usage with the increasing number of active users. Their memory usage lies between 14% and 18%. However, the memory usage of E3 lies between 20% and 30% and is much higher than E1 and E2 when the number of active users is fewer than 100. The resaon for rise and fall of memory usage is similar to the above discussion about CPU usage. With the increasing number of active users, the memory usage of E3 shows much better results than E1 and E2. Therefore, this justifies that the proposed SmartVM deployment optimizes the memory usage.

From Fig. 2c, we see that E1 and E2 present similar usage in the I/O when the number of active users increases. Their average I/O usage is between 4% and 7%. In contrast, the usage of I/O is much higher in E3. When the number of active users increases, E3 still shows better I/O usage compared with E1 and E2. Therefore, this justifies that the proposed SmartVM deployment optimizes the I/O usage.

D. Business Violation

Each user request contains a parameter 'timeout' that means user wants to get response in an expected period of time. If BMS cannot generate response within the 'timeout', it results in a business violation. Fig. 3a shows the number of business violations against the number of active users. We can see that, E1 and E2 cause 400 to more than 1600 business violations with the generally increasing number of active users. In contrast, the number of business violations in E3 is much fewer than 200. Therefore, this justifies that the proposed SmartVM deployment can reduce business violations.



(a) Business Violations (b) totalContainers

Fig. 3: Business Violations & Total number of Containers

E. Number of Containers Used

Fig. 3b shows the total number of docker containers used against the number of active users in E1, E2, and E3. From the figure, we can see that, for E1 and E2, the number of business containers increases with the number of active users. Moreover, the increasing curves for E1 and E2 are almost the same. The number of docker containers used for E3 is 4 when the number of users increased to 50 while the number of docker containers used for E1 and E2 is 7, it shows 42% improvement. When the number of users increased to more than 200, the number of docker container in E3 is 6 while the total number for E1 and E2 climbs up to 14, the improvement is 57%. This clearly shows the great cost saving of SmartVM as compared with monolithic and uniform microservice deployment.

V. Conclusion and Future Work

The proposed SmartVM SaaS deployment platform provides a solid although barebones prototype to effectively deploy SaaS applications. Based on the complexity of the target SaaS application and deployment environment, SmartVM can be customized to create a highly scalable, flexible, cost effective deployment environment based on Docker containers. The experiment results justify the outperformance of the SmartVM architecture in load balancing and auto scaling. In the future, we will develop a cost mechanism which bills based on the amount of resources currently in use, it would achieve optimal resource utilization, reduce precious resource wastage such as unused processing, memory or storage and at the same time reduce costs−for the customers as well as the cloud provider.

References

[1] J. Turnbull, The docker book. Lulu. com, 2014.

[2] T. Dillon, C. Wu, and E. Chang, "Cloud computing: issues and challenges," in Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on. IEEE, 2010, pp. 27–33.

[3] J. Espadas, A. Molina, G. Jiménez, M. Molina, R. Ramírez, and D. Concha, "A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures," Future Generation Computer Systems, vol. 29, no. 1, pp. 273–286, 2013.

[4] D. Guo, W. Wang, G. Zeng, and Z. Wei, "Microservices architecture based cloudware deployment platform for service computing," in Service-Oriented System Engineering (SOSE), 2016 IEEE Symposium on. IEEE, 2016, pp. 358–363.

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," arXiv preprint arXiv:1606.04036, 2016.

[6] J. Thönes, "Microservices," IEEE Software, vol. 32, no. 1, pp. 116–116, 2015.

[7] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on. IEEE, 2016, pp. 261–268.

[8] Z. L. UU, M. Korpershoek, and A. O. VU, "Towards a microservices architecture for clouds."

[9] G. Smaragdakis, N. Laoutaris, K. Oikonomou, I. Stavrakakis, and A. Bestavros, "Distributed server migration for scalable internet service deployment," IEEE/ACM Transactions on Networking, vol. 22, no. 3, pp. 917–930, 2014.

[10] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode," in Science Gateways (IWSG), 2015 7th International Workshop on. IEEE, 2015, pp. 34–39.

[11] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," IEEE Cloud Computing, vol. 3, no. 5, pp. 81–88, 2016.

[12] H. Mary Beth, "The cloud trifecta: Scalability, cost and efficiency, https://www.eci.com/blog/238-the-cloud-trifecta-scalability-cost-and-efficiency.html," 2012.

[13] M. Janakiram and C. McCrory, "Is docker a threat to the cloud ecosystem?" https://gigaom.com/2014/08/14/is-docker-a-threat-to-the-cloud-ecosystem/, 2014.

Authorized licensed use limited to: Macquarie University. Downloaded on December 10,2020 at 16:41:20 UTC from IEEE Xplore. Restrictions apply.