2018 IEEE Symposium on Service-Oriented System Engineering

Comparing Container-based Microservices and Workspace as a Service: Which One to Choose?

Junming Ma, Bo An, Donggang Cao, Xiangqun Chen

Key Lab of High-Confidence Software Technology (Peking University), Ministry of Education, Beijing, China
{mjm520,anbo,caodg,cherry}@pku.edu.cn

***Abstract*—The concept of microservices has gained increasing popularity since 2014. Almost during the same period, container technology keeps developing and is considered as an excellent way to build microservices-based applications. Mainstream public cloud vendors such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform all provide users with container-based solutions to implementing microservices. Workspace as a Service (WaaS) proposed by An et al. is another approach which uses containers to serve users. Both container-based microservices and WaaS are used to effectively utilize cluster resources via maintaining a number of containers. In this paper, we compare the designing ideas and supporting platforms of these two approaches, which provides a perspective for cluster administrators and users to understand the scenarios where to use them and how to make an appropriate choice to meet their needs. We find that container-based microservices are more suitable for professional IT companies while WaaS fits education and research institutions better.**

***Keywords*-cloud computing; container-based virtualization; microservices; Workspace as a Service;**

I. Introduction

The term of microservices has received widespread attention in recent years. Martin Fowler[1] gave an explanation of the definition and features of microservices in his blog in the year of 2014. This new architectural style aims at decomposing a single application into a suite of small services which are so-called microservices. In this paper, the term of container-based microservices refers to an approach to building this style through container technology. Each microservice is a process that can be deployed independently and different microservices interact with each other through lightweight communication mechanisms. For developers, using the microservices architecture has the following advantages compared with developing traditional monolithic applications: loosely coupled microservices architecture is more flexible; each microservice can be developed independently by different teams without affecting each other; microservices architecture makes it possible to release a new service while maintaining the availability and stability of the whole system, which is helpful for implementing DevOps.

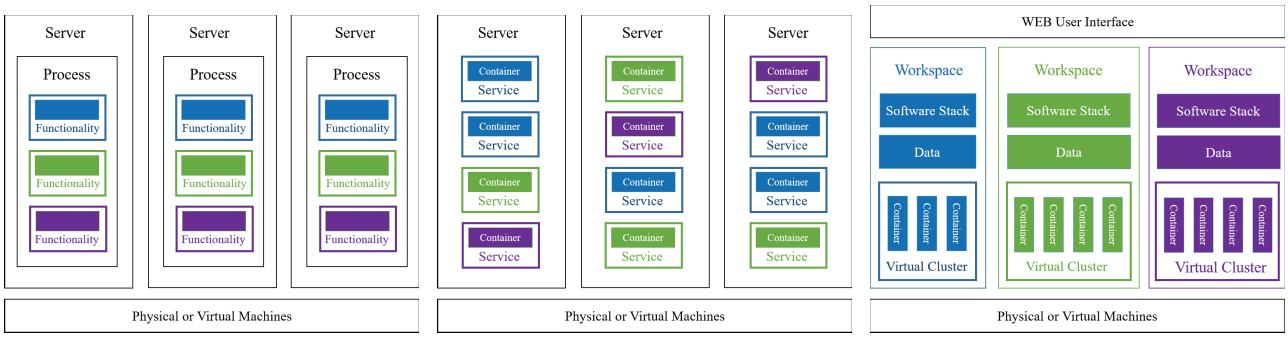
It is very appropriate to implement microservices-based applications through popular container technology. The key points of a container are the isolation and image mechanisms. In Linux, resource and access isolations are achieved through OS kernel features such as cgroups and namespace. An image consists of files which make up the applications and their dependencies. Due to these features, a container creates a perfect runtime encapsulation as a microservice needed. Currently, major public cloud vendors such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform all provide users with container services for developing microservices-based applications. More and more companies choose to run workloads via a container-based microservices way on their clusters, no matter bare-metal machines or cloud virtual machines.

Workspace as a Service (WaaS) is another approach proposed by An et al.[2] that uses container clusters to provide users with services. WaaS aims at building customized online workspaces for multi-users in a shared cloud environment. A workspace consists of three aspects including a virtual cluster, computing services, and data. WaaS makes containers connected via virtual networks to form virtual clusters on the underlying resource and provides customized software stacks for meeting the needs of different users. At the top lier, a web browser works as an interface for users to access their own workspaces and process tasks. For WaaS users, constructing a workspace simplify the procedures to utilize the underlying resources. For WaaS operators, WaaS effectively utilizes whole cluster resources by running different workspaces onto a shared environment.

Both container-based microservices and WaaS use containers encapsulate underlying resources to create isolated software runtime environments on clusters. In order to manage these containers, low-level platforms are needed to support high-level container-based microservices and WaaS to work stably. The supporting platforms essentially can be regarded as container-based cluster management systems. Both platforms have an open source implementation so those who are interested in have easy access to employ the two approaches. For new users who try to manage clusters using these approaches and open source systems, new concepts and usages make them become confused and lost easily.

There is no prior work systematically comparing these two approaches to help users tell differences and make choice. In this paper, we make a qualitative comparison

0-7695-6394-5/18/$31.00 ©2018 IEEE
DOI 10.1109/SOSE.2018.00040



(a) Monoliths (b) Container-based Microservices (c) Workspace as a Service

Figure 1: The Models of Monoliths, Container-based Microservices, and WaaS

of them from the aspects of basic designing ideas and supporting platforms. Our comparison provides a perspective for cluster users and administrators to understand these two container-based cluster management technologies and coarsely explores their suitable scenarios.

The rest of the paper is organized as follows. Section II introduces the designing ideas of container-based microservices and WaaS. Section III describes two representative supporting platforms of the two approaches. Related work is discussed in section IV. Section V concludes this paper.

II. Designing Idea

A. From Monoliths to Microservices

In order to understand container-based microservices, we need to understand the monolith application first. Figure 1a illustrates the model of a monolith application. A monolith application usually wraps all functionalities into a single process. Scaling a monolith application is achieved by running multiple replicas of the single process on multiple servers. Maintaining a large monolith application involves many teams, which makes developing, testing, releasing, and deploying difficult. The difficulty keeps increasing when the monolith application becomes increasingly complicated with a large number of functionalities. The step from monoliths to microservices is a solution to this problem. For the microservices-based approach, a monolith application is decomposed into many fine-grained services, each of that focusing on a single functionality. The approach of container-based microservices uses a container to encapsulate a single functionality, which usually could be a single process. Different services are combined to form a whole application. Application scaling can be achieved by running replicas at the granularity of service. Different services can be developed and maintained by different teams. The development, test, release, and deployment of a service will not affect the rest of the application. Furthermore, resources are isolated through container technology. Fine-grained services of different applications can be mixedly deployed on a whole cluster to improve resource utilization. Figure 1b illustrates the model of container-based microservices approach. A specific implementation may be different compared with this model. For example, a service may consist of multiple containers to support a functionality or some implementations may have their own further abstractions. However, different implementations follow the same general idea.

B. The Model of WaaS

Organizations usually build private clouds to reduce their IT expense through resource sharing upon existing infrastructure. Most private cloud platforms focus on IaaS level, which usually only provide users with low-level resources such as virtual machines. System environments have to be manually set up in virtual machines for users' jobs. What users need indeed is a high-level workspace with customized software stacks, configurations, and computing resources readily available. Jobs including online programming, debugging, and result visualizing can be done directly in this workspace. Currently, some PaaS cloud platforms such as Google App Engine provide users with built-in services for simplifying software development and deployment. However, PaaS is mainly designed for web applications and not powerful enough to meet users' extensive needs.

WaaS aims at providing users with such workspaces. Figure 1c illustrates the model of WaaS. Each user has a virtual cluster in his personal workspace, which consists of a set of connected containers. According to their needs, users can choose different container images which wrap customized software stacks. WaaS offers a web-based, user-friendly interface at the top tier. Each user does his jobs directly in a browser. Different users' workspaces are isolated from each other through containers, making the entire cluster resources shared by multiple users.

The designing idea of WaaS is different from container-based microservices. WaaS is not designed for making coordination among different teams nor a practice of DevOps to speed up time-to-market of software. WaaS focuses on fine-

grained resource sharing and ease of use, enabling users to obtain online working environments instantly on a shared cluster.

C. Application Container vs System Container

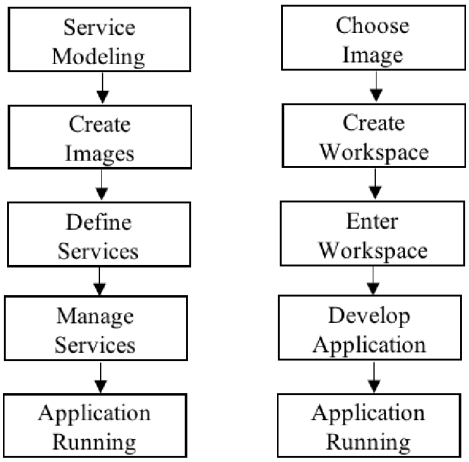
Although both container-based microservices and WaaS are built on top of containers, the types of used containers are different. Container-based microservices use application containers like Docker containers while WaaS uses system containers such as LXC. Using an application container is to better manage the lifecycle of a single-process service. An application container wraps a single process and its dependencies, making it possible for the service to be quickly started, shipped and deployed. Application container technology fits the concept of microservices, which leads to that an application container becoming the perfect carrier for a microservice. In contrast, a system container is considered as an OS-level isolation mechanism. It creates a clean OS environment in which to run multiple processes. A system container is more like a kind of lightweight virtual machine.

Container-based microservices use application containers to encapsulate, build, migrate, and deploy microservices while WaaS uses system containers to create and maintain workspaces. The differences between these two types of container technologies determine different usages. In WaaS, users can manipulate workspaces by accessing system containers via ssh terminals just as they would do with virtual machines, but it is not recommended to do this with single-process application containers. Therefore, the supporting platforms of container-based microservices provide users with a set of new mechanisms for maintaining microservices within application containers.

D. How to Use

Figure 2 illustrates the workflows of using container-based microservices and WaaS. If a user wants to build a microservice-based application through container-based microservices approach, he needs to model the application at first. An application is required to be decomposed into a series of microservices. For each service, a corresponding container image should be created or found in an image warehouse. Next, the user needs to define and organize these services with some mechanisms provided by the supporting platforms, which helps users automatically deploy and maintain these services. Finally, the application begins to work on the supporting platforms.

For WaaS, all work is done in browsers. Users firstly customize their own workspaces by provisioning container resources and choosing container image templates according to their needs. These steps are finished by clicking several buttons in browsers. After a workspace is launched, a Jupyter Notebook[3] service which acts as the user interface will be started by default. Through Jupyter Notebook, users could do all jobs including programming, debugging and result visualizing in workspaces via browsers. Creating a needed workspace with a browser is easy and convenient, which frees users from installing, configuring required software dependencies and lowers the threshold for using applications, especially complex distributed applications in cluster environments.



(a) Container-based Microservices (b) WaaS

Figure 2: The Workflows of Container-based Microservices and WaaS

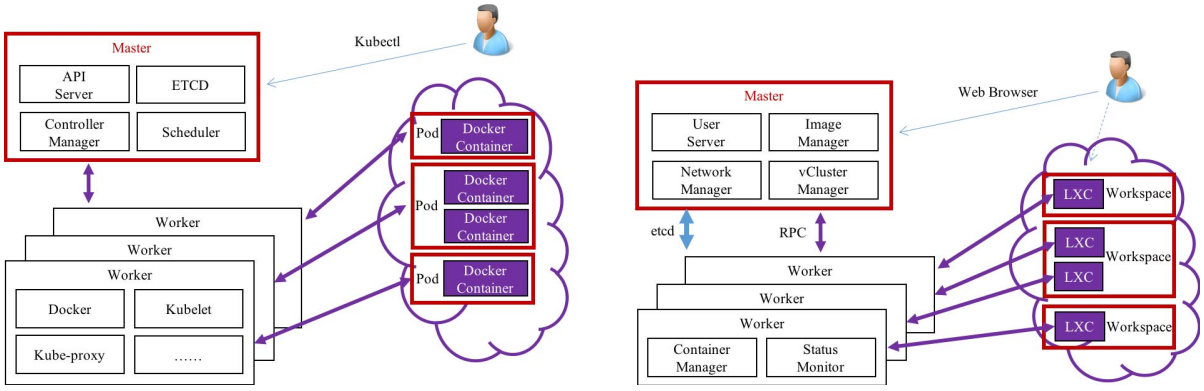
III. Supporting Platform

In this section, we introduce and compare the supporting platforms of container-based microservices and WaaS. Kubernetes[4] is the most well-known container management system used to build microservices, which is an open source[1] project launched by Google and originated from Google's internal Borg[5] system. Mainstream cloud vendors including Google Cloud Platform, Azure, and Bluemix use Kubernetes as the underlying engine to provide users with container services. Therefore, Kubernetes is representative as the supporting platform for container-based microservices. WaaS has also been implemented as an open source project Docklet[2]. In this paper, we choose Kubernetes and Docklet as the supporting platforms of the two approaches.

A. Architecture

The simplified system architectures of Kubernetes and Docklet are illustrated in Figure 3. Both Kubernetes and Docklet employ a typical distributed system structure of Master/Worker. The components on the master node are responsible for managing an entire cluster as well as cluster's interaction with external clients while the components on

[1]https://github.com/kubernetes/kubernetes
[2]https://github.com/unias/docklet



(a) Kubernetes (b) Docklet

Figure 3: Simplified System Architectures

each worker node are responsible for processing tasks that take place on each node.

Kubernetes makes a further abstraction than the model in Figure 1b. A Pod in Kubernetes is a basic unit for scheduling. Each Pod consists of one or multiple containers which share a set of Linux namespaces, cgroups, IP address, port space, volumes and can use inter-process communications talk with each other. A set of Pods with rules that are used for load balancing form a service in Kubernetes.

There are four major components on the master node of Kubernetes: API Server provides APIs for cluster and application management and acts as a bus for internal components. Besides, external clients access a cluster through API Server, which requires API Server the capabilities of authentication, authorization and access control. Controller Manager is responsible for maintaining applications. ETCD saves information about a cluster's status. Scheduler puts Pods on specific hosts according to certain policies.

On each worker node, there are also some important components to support the whole system: Kubelet acts as an agent of a worker node, managing Pods and containers. The duty of Container Runtime is to download images and run containers. Docker is the most used Container Runtime in Kubernetes. Network forwarding from a node port to a Kubernetes service and further to backend Pods is achieved by Kube-proxy.

Docklet runs a single master process on its master node to manage an entire cluster. The important components in a master process are as follows. User Server is responsible for user authentication and authorization. Network Manager is used for managing network configurations and connection among containers. VCluster has the duty of maintaining the lifecycles of virtual clusters. The software stack required for a customized workspace is configured, saved and loaded by Image Manager. In addition, an ETCD cluster saves information about the status of Docklet cluster as it does in Kubernetes.

Similarly, there is a worker process running on each Docklet's worker node, acting as an agent of a worker node. There are two components in a worker process, namely Container Manager and Status Monitor. System container LXC is used in Docklet. Container Manager manages the lifecycle of each LXC and configures its CPU, memory, network, and disk according to a user's need. Status Monitor collects each worker node's status information and reports it back to the master node. The master process communicates with worker processes through remote procedure call.

Kubernetes decomposes responsibilities of master and worker nodes into different components. Each component has a corresponding process to perform its responsible tasks. Different components work together through a specific component API Server. Docklet also implements a decomposition on capabilities, but it is located at the granularity of method and function. Different components coordinated by RPC form a single master or worker process. This architectural difference makes Kubernetes more loosely coupled than Docklet.

Kubernetes focuses on microservices and intends to provide a well-equipped platform to build, deploy and run microservices-based applications. In order to achieve this, Kubernetes abstracts a series of concepts such as Pod, Service and etc. Furthermore, to better serve users, the architecture is designed to support load balancing, service discovery, rapid deployment and auto-scaling for microservices. However, Docklet focuses on workspaces. Workspaces are supposed to be resource sharing, easy to use and convenient. Therefore, Docklet needs to design fine-grained resource managers to take over the setups of network, storage and other underlying resources for users so that a working environment can be provided for directly using. The differences between the two architectures reflect the different designing ideas between container-based microservices and WaaS.

B. How to Use Kubernetes and Docklet

The use of Kubernetes and Docklet follows the workflows illustrated in Figure 2. However, a new set of mechanisms need to be learned in Kubernetes. In order to manage and maintain services, Kubernetes designs a lot of objects to implement the capabilities of load balancing, service discovery and etc. The definitions of these objects are often achieved by writing configuration files. Users have to learn how to write these definitions and call Kubernetes API to create these objects. Kubernetes then will automatically deploy and maintain these services.

Additionally, troubleshooting is also important in these platforms. In Kubernetes, both cluster errors and application exceptions could cause an application not work successfully. For example, insufficient resource or node failures may lead to Pods failing scheduled. In this situation, users need to view the status information to troubleshoot problems. Logging information can also be useful. Due to the use of application containers, logs are recommended to be acquired through a container's stdout, stderr streams or via third-party plug-ins like fluentd. For an application's internal faults, users can debug by entering the containers or launching another specific debugging service.

Troubleshooting in Docklet is more ordinary. Because system containers are just like virtual machines, Docklet users could debug applications within their workspaces directly. Users can output application logs to any valid directory or use any suitable debugging tools. In addition, popular debugging tools can also be included in the software stack of an image to speed up debugging procedures.

C. User Support

Kubernetes and Docklet both have multi-user support. Kubernetes employs a Role-Based Access Control (RBAC) mechanism to enable fine-grained resource access control for different users. In Kubernetes, objects *Role* and *ClusterRole* contain permissions for resources and APIs which can be accessed in a specific namespace or the entire cluster. Objects *RoleBinding* and *ClusterRoleBinding* map *Role* or *ClusterRole* to an individual user or a user group, so different users obtain different permissions.

Docklet does not employ a fine-grained access control mechanism like Kubernetes. In Docklet, there are two types of roles, i.e. admin and normal users. Normal users are further divided into different levels. For admin user, he has permissions to monitor the cluster's status, modify resource quotas for normal users, manage image libraries and issue notifications to users. For normal users, they are root users in their own unique workspaces but are isolated from each other. Normal users with different levels have different resources quota in their workspaces. In short, using Kubernetes to manage a cluster is similar to the scenario that multiple users share a computer through a multi-user operating system and using Docklet is just like that a public cloud vendor serving multiple tenants with IaaS.

Cluster management systems are supposed to provide users with interfaces to interact with the cluster. Kubectl is a mainly used command line interface for Kubernetes users. In addition, Kubernetes also provides users with APIs for effectively utilizing clusters by programming. Docklet provides users with a friendly web-based interface. Users create customized workspaces and process data directly in browsers. Besides, users could access their own workspaces via web terminals supported by Jupyter Notebook in Docklet. If a user launched an ssh server in his workspace, he can also access the workspace through command line terminal. Kubernetes and Docklet both have good support for user interfaces. The former focuses on efficiency while the latter focuses on ease of use.

D. Usage Scenario

Migrating workloads to container-based microservices on Kubernetes requires users to learn a lot of new knowledge to model, develop, debug, and manage applications. These are not easy to achieve, especially for beginners who are new to use containers. However, once mastered corresponding skills, developing efficiency and resource utilization can be improved. For IT companies and departments, who emphasize rapid product iterations, cost savings, and coordination of multiple teams, container-based microservices approach is beneficial and worth learning.

In contrast, the learning curve of migrating workloads to WaaS on Docklet is far less steep. A workspace is not significantly different from a real server cluster. WaaS cares about ease of use and convenience, not aiming at improving the efficiency of multiple involved teams in an organization. In WaaS, workspaces are independent. Different users share cluster resources but do not interfere with each other. Research institutions and universities are suitable scenarios for WaaS. There are usually limited physical resources for research and teaching at these places. Each researcher or student needs a personal workspace to do his own jobs. Not only does WaaS provide a resource-sharing approach for these organizations, but its ease of use allows non-IT professionals like researchers, students not obtain additional learning costs.

Combined with cloud computing, both approaches and systems could make use of cloud's elasticity and scalability, providing better management of resources. Currently, container-based microservices have been widely supported by mainstream cloud vendors. WaaS steps towards the direction of becoming the virtual cloud for special purposes[6] in the JointCloud[7] environment, where computing, storage, and network resources can be coordinately provided, audited and utilized from multiple clouds. The virtual cloud for special purposes will be able to utilize resources provided by JointCloud environment to create workspaces.

IV. Related Work

With the benefits of fine-grained resource encapsulation and isolation, containers have been adopted for cluster management and relevant technologies have been extensively studied. Google proposed Borg[5] for large-scale cluster management using container-based isolation. Omega[8] used parallelism, shared state, and lock-free optimistic concurrency control to efficiently schedule flexible, scalable clusters. Mesos[9] proposed a two-level scheduling mechanism to meet the resource requirements of different computing frameworks in a shared data center. Kubernetes[4] built a framework for developing microservices architecture applications in a cluster with Docker containers. Docklet[2] used LXC to provide users with independent customized working environments in a shared data center. In this paper, we investigate Kubernetes and Docklet because they are open source systems and represent two distinctly different approaches to using containers manage clusters.

There are other comparative studies on containerization or PaaS platforms. Roberto Morabito[10] investigated the energy consumption differences among four virtualization technologies, including Docker and LXC. Mario Villamizar et al.[11] compared the costs of running a web application using AWS Lambda, monolithic and microservice architectures. Tasneem Salah et al.[12] compared the performance differences between container-based and VM-based public cloud services. Rajdeep Dua et al.[13] did a comparison among several container-based PaaS architectures. Our work mainly compares two container-based approaches rather than the underlying virtualization technologies.

V. Conclusion and Future Work

Container-based microservices and Workspace as a Service are two approaches that use containers to manage clusters. This paper makes a comparison of these two approaches from designing ideas and supporting platforms. Loosely coupled container-based microservices are more flexible, independent and easy to release and deploy, helping software departments improve efficiency and resource utilization. However, developing microservices-based applications sets a high threshold for normal users, which requires them to master a set of new mechanisms. WaaS provides users with customized, easy-to-use and convenient workspaces on shared clusters, which simplifies the creation and setup of a working environment. From the comparison, we conclude that container-based microservices approach is better suitable for IT companies to coordinate different teams while WaaS fits education and research institutions better.

Currently, this work only qualitatively compares some features of these two approaches. Our future work will try to make a more detailed quantitative comparison in which some performance experiments and evaluation metrics will be included.

Acknowledgment

This work is supported by the National Science and Technology Major Project under Grant No.2016YFB1000105; the National Natural Science Foundation of China under Grant No. 61272154, 91318301; the Science Fund for Creative Research Groups of China under Grant, No. 61421091.

References

[1] M. Fowler, "Microservices: a definition of this new architectural term," https://martinfowler.com/articles/microservices.html.

[2] B. An, X. Shan, Z. Cui, C. Cao, and D. Cao, "Workspace as a service: An online working environment for private cloud," in *Service-Oriented System Engineering (SOSE), 2017 IEEE Symposium on*. IEEE, 2017, pp. 19–27.

[3] F. Pérez and B. E. Granger, "Ipython: a system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[4] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 167–167.

[5] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[6] D.-G. Cao, B. An, P.-C. Shi, and H.-M. Wang, "Providing virtual cloud for special purposes on demand in jointcloud computing environment," *Journal of Computer Science and Technology*, vol. 32, no. 2, pp. 211–218, 2017.

[7] H. Wang, P. Shi, and Y. Zhang, "Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1846–1855.

[8] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[10] R. Morabito, "Power consumption of virtualization technologies: an empirical investigation," in *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE, 2015, pp. 522–527.

[11] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 179–182.

[12] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "Performance comparison between container-based and vm-based services," in *Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on*. IEEE, 2017, pp. 185–190.

[13] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 610–614.