Towards a container-based architecture for multi-tenant SaaS applications

Eddy Truyen, Dimitri Van Landuyt, Vincent Reniers, Ansar Rafique, Bert Lagaisse, Wouter Joosen
iMinds-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

ABSTRACT

SaaS providers continuously aim to optimize the cost-efficiency, scalability and trustworthiness of their offerings. Traditionally, these concerns have been addressed by application-level middleware platforms that implement a multi-tenant architecture.

However, the recent uprise and industry adoption of container technology such as Docker and Kubernetes, exactly for the purpose of improving the cost-efficiency, elasticity and resilience of cloud native services, triggers the unanswered question whether and how container technology may affect such multi-tenant architectures.

To answer this question, we outline our ideas on a container-based multi-tenant architecture for SaaS applications. Subsequently, we make an assessment of the technical Strengths, Weaknesses, Opportunities, and Threats (SWOT) which should be taken into account by a SaaS provider when considering the adoption of such container-based architecture.

1. INTRODUCTION

Recently, there has been an increasing industry adoption of Docker containers for simplifying the deployment of software [7, 24, 10]. Almost in parallel, container orchestration middleware, such as Kubernetes [3], Docker Swarm, Mesos and Openshift 3.0 [10] have arisen which provide support for automated container deployment, scaling and management.

Docker offers a user-friendly command line interface for running multiple, isolated application instances on the same node by means of user-level virtualization. As such, these different application instances run on a shared linux kernel, but have isolated file systems, namespaces and computing resources [8]. Kubernetes builds upon and extends Docker with additional support for deploying and managing a multi-tiered distributed application as a set of containers on a cluster of nodes. One of the strengths of Kubernetes is improved cost-efficiency which means that less machines are needed for running a certain workload within a certain quality-of-service level [3, 18].

SaaS providers are faced with the increasingly relevant question whether and how this container technology can play a significant architectural role in the development and operation of multi-tenant SaaS applications. This question is not trivial because multiple factors are at play. For example, the popular *shared-everything* multi-tenant architecture is more cost-efficient than containers: tenants run not only on a shared OS, but also within a shared application instance. However, such shared-everything architectures trade the gain in efficiency for a weaker security isolaton between tenants. Another consideration is that multi-tenant SaaS applications typically rely on complex application-level middleware services for performance isolation [20], tenant data management [14] and multi-cloud deployments [22]. In principle, it is possible to simplify these middleware services by shifting some functionality to a container orchestration layer. Of course, container orchestration middleware is no silver bullet: hard problems such as unified configuration management and automated management of service dependencies remain an open challenge [3]. In summary, there does not exist a simple answer to whether container technology supports a strengthening or weakening of multi-tenant SaaS architectures.

In this paper, we present our initial ideas and an in-depth analysis of the Strengths, Weaknesses, Opportunities, and Threats (SWOT) of a container-based architecture for multi-tenant SaaS applications. Section 2 presents an overview of the contemporary requirements of multi-tenant SaaS applications, and Section 3 subsequently provides the necessary background on Docker and Kubernetes. Subsequently, Section 4 outlines our ideas on a container-based multi-tenant architecture for SaaS applications and Section 5 presents the SWOT assessment with respect to the outlined SaaS requirements. Section 6 then outlines related work, whereas Section 7 concludes the paper.

2. SAAS APPLICATIONS: CHALLENGES AND OPPORTUNITIES

This section summarizes our analysis of the main challenges and opportunities SaaS providers are facing, which is based on our extensive and frequent interaction with industry-level SaaS providers in a wide range of application domains. The top-level objective is to optimize **cost-efficiency**. The [service level/price] ratio determines the competitiveness of the SaaS offering, and the continued efforts to maximize this ratio has led to a number of trends in how SaaS applications are built and deployed:

ARM 2016, December 12-16, 2016, Trento, Italy

DOI: http://dx.doi.org/10.1145/3008167.3008173

1. Cloud deployment: SaaS applications are increasingly hosted on top of scalable cloud infrastructure, for example on top of Platform-as-a-Service (PaaS) offerings [21], to avoid large investments in on-premise resources. This accomplishes a number of benefits: (i) **Resilience**: failure of a node can be recovered with minimum data loss and or data synchronization; (ii) **Scalability**: Performance remains when usage load increases; (iii) **Elasticity**: new nodes are added dynamically based on user load.

2. Multi-tenancy is an architectural tactic aimed at increasing cost-efficiency by sharing the available resources maximally among many customer organizations (tenants). Multi-tenancy introduces a number of additional challenges: (i) **Customization**: multi-tenant SaaS applications must be adapted towards tenant-specific and user-specific requirements; (ii) **Performance isolation** [20]: performance can be guaranteed for different types of workloads and performance can be guaranteed per tenant. Agressive tenants cannot impact loyal tenants; (iii) **Manageability**: the SaaS application remains manageable, also in the context of an increasing number of tenants and co-existing tenant customizations.

3. Multi-cloud and hybrid cloud deployment: By leveraging the combined benefits of different cloud resources, both private and/or public, and having the ability to dynamically add resources offered by different providers, a number of additional benefits can be attained: (i) **Cloud portability**: SaaS applications must be able to run on different cloud providers, which is not straightforward due to the high degree of heterogeneity between cloud technologies; (ii) **Dynamic reconfiguration**: by observing the underlying cloud platforms and offerings, SaaS applications can be built that support varying degrees of dynamicity, for example to automatically migrate the application (or parts thereof) to different cloud providers [22, 14].

As indicated by many surveys on cloud adoption, **Security** and **Privacy** remain the number one concerns for SaaS providers who outsource part of their operations to third-party cloud providers. Also, **security isolation between tenants** is important when multiple tenants are served by the same application instance. These concerns become even more stringent in light of the trends outlined above.

SaaS applications increasingly rely on complex middleware solutions to support the above requirements [22, 20, 14]. These solutions involve complex policy engines [14, 22], etc.

This paper highlights and discusses the potential of addressing some of these requirements differently, by leveraging container technology, and outlines the main research challenges and gaps.

3. DOCKER AND KUBERNETES

The recent popularity of container technology such as Docker, LXC and OpenVZ is in part attributable to the fact that it makes a different trade-off between cost-efficiency and security isolation when compared to virtual machine technology such as Xen, VMWare and KVM [7, 24].

One key difference between containers and virtual machines is that containers run on a shared Linux kernel, whereas virtual machines run on a hypervisor which abstracts computer hardware architecture and connected devices. As a result, virtual machine images consist of a separate operating system on top of which the application runs, and these are therefore more heavy-weight. In contrast, containers only need to contain the application with its dependent libraries. The isolation between virtual machines is performed by the hypervisor, whereas the isolation between containers is achieved by means of different mechanisms of the Linux kernel. Resource isolation is achieved by means of `cgroups`, file system isolation is achieved with `chroot`, and finally, isolation between process ids, IPC mechanisms, network stack and mount spaces is achieved by means of kernel namespaces. Xavier et al. [8] present a comprehensive overview of these technologies.

Containers have become popular thanks to Docker which offers a daemon and a user-friendly command line interface for running and managing containers. The application and libraries which are run in Docker containers are packaged in images. These images are stored in a local or a remote Docker registry. To speed up download from such registry, an image is built as a set of incremental layers which can be separately transmitted and stored.

Kubernetes is an open-source container orchestration middleware created by Google [3] that offers a uniform API for deploying and managing distributed multi-tiered applications as sets of containers on a cluster of physical or virtual machines. Kubernetes offers many concepts and abstractions for constructing and managing complex applications [16, 1], the most notable of which are:

Nodes: a Node is a physical host or virtual machine on top of which containers can be scheduled. The scale of a Kubernetes cluster can range from two to thousands of Nodes.

Pods: a Pod is a `set` of containers that logically belong together and are therefore always deployed together on the same node. It is therefore a unit of failure. Since Pods do not hold persistent state, all changes to its running containers are lost when a Pod fails or crashes. An example of a Pod is an instance of a web server.

Volumes contain persistent state. They can be implemented by services for attaching block storage devices to guest virtual machines such as Cinder, distributed file systems such as NFS, or simply a directory in the path of the local host. Volumes are linked to a container where they become available as mounted directories.

Services: a set of Pods is exposed to a customer via a Service which embodies a stable IP address, a network protocol and one or more ports. All Pods and Services have a unique cluster IP address. Services can also have external IP addresses that are either supported by an external load balancer or by so-called NodePorts. The latter means that each service is accessible on all physical nodes of the cluster via a unique port number.

Replication controller: Replication controllers ensure that the amount of replicated Pods keeps up with the desired number of replicas. The Replication Controller keeps track of the total amount of replicas per Pod by monitoring each Node in the cluster.

Deployment: a Deployment provides declarative updates to Pods and Replication Controllers. Deployments can be gradually performed at the desired pace (e.g. to support canary testing where one deploys a new release of a service only to one replica of a Pod). They can be monitored for success and can be rolled-back to a previous deployment.

Namespaces: a Namespace is a mechanism to partition resources created by different users into a logically named

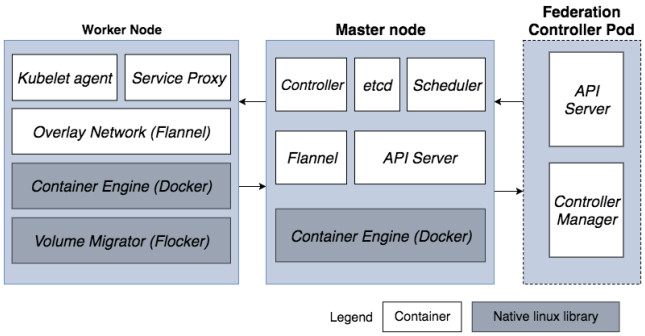

Figure 1: A concrete Kubernetes architecture (based on the portable docker-multinode project [2]).

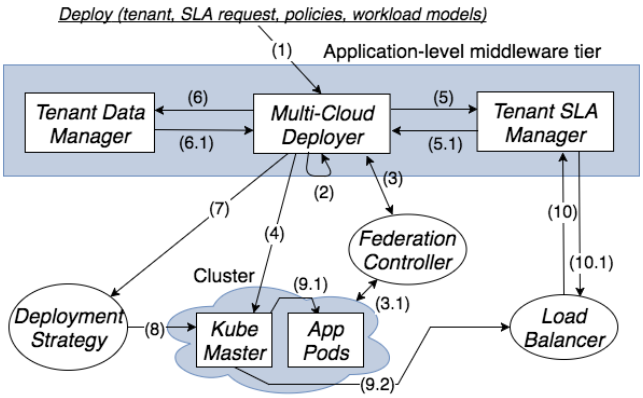

Figure 2: Container-based architecture for multi-tenant SaaS applications

group. A single cluster should be able to satisfy the needs of multiple user communities. Each user community has its own resources (Pods, Services, etc.), policies and resource quotas [11].

Request and Limits: Computational resources can be allocated to Pods and Containers by defining a `<request, limit>` pair for each resource. A Request defines the minimal resource quantity that should be reserved at all times for the container (i.e. 1 cpu and 512 MB of RAM), while a Limit specifies the maximum resource quantity that can be used by this container (e.g. 1.5 cpu and 1.5 GB of RAM). Request and Limits associated to containers are enforced at runtime in Docker using the Linux `cgroups` kernel feature. They are also used by Kubernetes to determine on which Node a Pod is best placed. Currently, this resource allocation model is only supported for CPU and memory resources, but support for other resources is planned in the near future [1].

As shown in Figure 1, the overall architecture of Kubernetes features a master node that offers a REST-based API for deploying and managing all the entities discussed above, and multiple worker nodes on top of which Pods are deployed. The master node consists of the following components: an *API Server* for processing the API requests, a policy-based *scheduler* for allocating Pods to nodes and a *controller* for checking the status of a deployment, for example whether the actual replica count of a Pod matches the desired replica count. Master node and worker nodes also encapsulate a local *kubelet* agent for deploying and managing the containers on the local node and a *service proxy* that acts as a client-side load-balancer for forwarding service requests to Pods. Each Pod and Service is assigned a unique virtual cluster IP address. To manage the mapping between cluster IP addresses and real host IP addresses, overlay network software (e.g. *Flannel*) is used. When a Pod dies and is rescheduled to a new Node, any attached Volumes should be migrated to the new Node and remounted with the new Pod. To achieve this, Kubernetes can be configured to use a specific distributed volume management system such as *Flocker* [19]. In addition, the master and worker nodes use a distributed key-value store (e.g. *etcd*) for registration and discovery of services and nodes. A single Kubernetes cluster is meant to be run within the scope of a private data center or an availability zone of a public cloud provider. To support managing multiple Kubernetes clusters across multiple cloud providers or availability zones, the Federation Controller Service can be used [9].

Kubernetes provides many tools for governing a distributed deployment of containers onto a pool of heterogeneous resources.

4. A CONTAINER-BASED ARCHITECTURE FOR MULTI-TENANT SAAS

Multi-tenant SaaS applications typically adopt a multi-tiered architecture with a front-end load balancer. In recent work, we have extended such architecture with an additional tier of *application-level middleware services* [14, 22, 20]. These services provide support for specific non-functional requirements across multiple cloud providers that cannot be fully supported by a single third-party PaaS provider. We now illustrate how this tier can be simplified and strengthened by relying on a container orchestration middleware such as Kubernetes.

A possible high-level architecture for the application-level middleware tier is presented in Figure 2. It consists of three services: (i) the *Tenant SLA Manager* service for tenant SLA management and performance isolation [20], (ii) the *Tenant Data Manager* service for adaptive data management across multiple database technologies (e.g. MongoDB and Cassandra) and cloud providers [14] and (iii) *the Multi-Cloud Deployer* service for deploying SaaS application components across multiple cloud providers [22]. Each of these middleware services implement a policy-based architecture, possibly extended with application-centric montoring and an autonomic control loop. Figure 2 shows a collaboration diagram for requesting the deployment of a new tenant to an already-running SaaS application (see message (1) in Figure 2). This request consists of a tenantID, tenant-specific SLA requirements and policies, and one of more workload models[1]. This request is then handled as follows: (2) select a suitable deployment strategy for the tenant, (3) get the URLs of the relevant Kubernetes clusters, (4) retrieve or create the Namespace for the tenant in each cluster, (5) get updates for the Requests and Limits of the containers and

[1]These workload models describe various dimensions such as the expected number of users and usage volume, whether the computations are cpu-, memory or i/o-bound.

Pods running in each Namespace, (6) get updates for the configuration of the database nodes, (7) execute the selected deployment strategy, (8) create a new Deployment with updated Services and Replication Controllers for the application and data tiers of the SaaS application, (9.1) add new Pods and/or reconfigure the running Pods, (9.2) update the external load balancer with Service and Pod IP routing info, and (10) retrieve admission control data for the tenant.

Alternative SaaS deployment strategies. We envision three different alternative deployment strategies that can be selected in step (2). In essence, each strategy represents a different trade-off between cost-efficiency and security isolation.

1. Shared container: in this deployment strategy, one instance of the existing multi-tenant application is allocated to and executed in one Container and one Pod per Node, and so each Node of the cluster runs an application instance which is able to process requests from any tenant. These application instances are stateless in the sense that tenant data and tenant features are activated at run time, based on run-time parameters such as tenant identifiers associated to service requests. To deploy these Pods, a single Service and single Replication Controller configuration file is specified.

Each of the pods have a mutable cluster IP address. This implies that if a single pod dies or is rescheduled to another Node, the existing Pod is replaced by a new Pod and a new Cluster IP address is created for this new Pod. This IP address change is not visible for external users who access the Pods via the stable service IP address.

If the Docker image is preloaded on each Node, it will only take a few seconds to deploy the entire service configuration and bring it into a running state. When a container crashes, the Kubernetes master will detect this failure and schedule the creation of a new Pod on the same or another Node.

2. No shared container/tenant namespaces: this deployment strategy extends the above in the sense that one or more tenants may be given a dedicated Service and Replication Controller. To ensure that computational and object resources are spread across the different tenants in accordance with tenant SLAs, each tenant is associated to a separate Namespace. Within each Namespace, default container Requests and Limits for a particular resource can be defined as well as a quota on the total amount of Requests and Limits . Finally, quota on the amount of Services, Replication Controllers, Pods etc. can also be set[2].

In comparison to the shared container strategy, the main advantage is that this scheme realizes an improved security isolation between tenants. Another advantage is that the SaaS application can be optimized performance-wise for a single tenant. First, resource constraints and replication levels can be configured per tenant. Second, tenant-specific customizations can be loaded when starting the container, and therefore do not have to be activated at runtime. Optimally, the dependency injection middleware, which is used for loading and invoking the tenant-specific customizations, should be easy configurable to switch from dynamic to static dependency injection.

The disadvantage of this deployment strategy is reduced cost-efficiency, especially for memory-bound workloads. For example, if the aim is to divide the resources of a test cluster with five 2GB nodes equally among tenants, it is possible to host approximately at most 20 Pods of a Tomcat web server. This implies that maximum 10 tenants can be hosted with replication level 2.

3. Shared container or namespace per SLA class: The third deployment strategy combines the above two by distinguishing between different SLA classes (e.g. gold, silver, bronze). For each class, a separate Namespace is created with a specific quota of resources and default Requests and Limits for containers. In each Namespace, a single Service and Replication Controller is created. Tenants are associated with a certain SLA class, and therefore connect to the Service associated with the corresponding Namespace.

The main advantage of this strategy is that it combines the benefits of the previous two: there is an improved security isolation between different classes of tenants, and only a single Service needs to be created per SLA class.

The main disadvantage is that within a specific SLA class, there is no support for security isolation between tenants.

The next section discusses our in-depth SWOT analysis of the adoption of container orchestration for supporting multi-tenant SaaS applications. As the implementation and in-depth validation of the architectural ideas sketched above is currently ongoing work, this analysis is based on our ongoing experiments and findings with Kubernetes in Openstack and Google Cloud [17].

5. SWOT ANALYSIS

In a multi-tenant architecture, available resources are cost-effectively multiplexed across tenants. All tenants share the same application stack, and in some cases share application instances. In general, containers do not provide an improvement on the top-level objective of cost-efficiency. However, when taking all SaaS requirements (outlined in Section 2) into account, container technology can certainly play a role.

Strengths.

Resilience: Automated volume migration drastically reduces the time to synchronize data from a failed Pod to a newly-started Pod. Moreover, clients can continue to access the new Pod via the existing Service IP address.

Elasticity: The amount of Pods behind a Service can be scaled relatively easy when the container image is already stored on all the Nodes. The time to add a new Node to a Kubernetes cluster depends among other factors on the time to install the Kubernetes Worker component. In our ongoing experiments, a Worker Node is running within less than two minutes in our test cluster (using the portable deployment of Kubernetes). We found that installing Flocker natively in Linux requires a lot of manual and error-prone work, however, and therefore should be automated by a configuration management tool.

Dynamic reconfiguration: (1) Up- and downgrading a service to a new version is simplified significantly using the mechanism of Deployments. (2) It is easier to ensure configuration parameter consistency between development, testing and production environments because a single Docker image packages the entire application and libraries in a component that is identified by a globally unique URL and version number. (3) Bootstrapping a Docker image is less error-prone and generally takes less time than installing software via

[2]More information on the resource allocation model of Kubernetes can be found in [1].

configuration management tools such as Puppet or Chef.
Cloud portability: Several factors contribute to an improved portability of applications: (1) in theory, Docker can run on any Linux-based operating based system; in practice very few parts of the OS interface expose slight variations between different Linux distributions [3], (2) Kubernetes is offered as a service by several cloud providers, (3) a portable deployment of Kubernetes, where the Kubernetes components themselves are started inside Docker containers, exists [2].

Weaknesses.

Security: There are two fundamental security weaknesses to containers. (1) Containers run on the same host operating system, enlarging the attack surface considerably in comparison to virtualization. So-called privileged containers even have root access to the host OS. This is one of the reasons why public cloud providers offer containers that are started inside virtual machines: the cloud providers rely on the stronger security guarantees of VMs to protect their data center assets against external customer code. (2) Another security issue arises in multi-user environments where multiple users have the permission to deploy containers, and thereby are able to run potentially malicious code that is hidden inside a publicly available container image [24]. The beta release of Kubernetes v1.4 supports security policies for controlling access permissions of Pods.

Opportunities.

Configurable security isolation between tenants: Even though the security isolation of containers is less strict than in virtual machines, containerized applications are better isolated from each other than multiple tenants are in a traditional multi-tenant SaaS application. Each of the three deployment strategies discussed in Section 4 in effect represent a different trade-off between security isolation of tenants and cost-efficiency. As such, there lies a big opportunity for SaaS providers to offer tenants a configurable trade-off between security isolation and cost-efficiency.
Simplified performance isolation: To enforce tenant-specific SLAs and implement performance isolation, existing multi-tenant SaaS applications depend on (1) application-level monitoring, (2) request scheduling across all tiers, and/or (3) admission control in the Load balancing tier [20]. The Kubernetes resource model of Requests and Limits and the underlying `cgroups` implementation offers a flexible way to enforce resource constraints across different workloads with best-effort, bursty or hard resource demands [3, 1]. Moreover, Kubernetes offers application-centric monitoring services [3]. These two mechanisms may drastically simplify application-level monitoring of SaaS applications and potentially make request schedulers obsolete. However, this advantage only exists in the second and third deployment strategies, since the first one involves executing all tenants in the same Namespace.

More importantly, this opportunity depends on the assumption that it is possible to map tenant SLAs to container resources: given a certain workload type (which is either CPU-, memory-, network-, or disk-bound), it should be possible to determine a well-defined relation between a service level target (e.g. request latency, availability) of a tenant and the appropriate resource quota and default container Request and Limits for the Namespace assigned to that tenant. It seems logical that Namespaces with higher default requests or limits can guarantee better service level targets, which are typically expressed as percentiles. However, if the goal is to achieve both improved cost-efficiency and performance isolation, it is not clear from existing literature if such correlations exist.

Threats.

Hybrid virtualization technologies threaten cloud portability: a number of hybrid virtualization solutions have emerged that aim to offer different flavors between virtualization and containers. Kubernetes currently provides also support for the *rkt* container engine which is part of CoreOS. Therefore a container image standard is needed similar to the Open Virtualization Format [6].
Increased management complexity: Besides the management risks and costs of new technology adoption, there are also a number of structural problems which increase engineering complexity [3]:
(i) Uniform configuration management: while the REST-based Kubernetes API is a big step towards uniform and consistent service configuration, still a lot of hetereogeneous and programmatic configuration is required for setting up real applications. To illustrate, even the relatively common scenario of adding a new MongoDB replica to an existing replica set requires a complex sequence of actions [4], which in practice involves creating a new Flocker volume, manual scripts to generate YAML files, running javascript code to update the replica set in the MongoDB primary node in a so-called sidecar container. Existing configuration management systems such as Chef and Puppet, and cloud-based server orchestration tools such as juju also seem to work only for specific use cases and often need to be combined [23].
(ii) Management of service dependencies: the automated management of service dependencies is considered a difficult problem because of various issues discussed in [3]. Application-level middleware services may resolve part of this problem. For example, our existing work on policy-based service dispatchers in hybrid cloud environments [22] enable tenants to declaratively express which services should be used for which service requests.

6. RELATED WORK

Existing studies of containers consistently report and confirm that containers trade improved cost-efficiency and performance for reduced security isolation in comparison to virtual machines [24, 15, 7]. Existing literature on container orchestration systems [18] demonstrates improved resource utilization in terms of number of machines needed for fitting a certain workload on.

In terms of performance isolation, containers do not yet provide complete isolation of resources as virtual machines do [8, 7]. Verma et al. [18] report that the implementation of the `cgroups` mechanism requires substantial tuning of the standard Linux CPU scheduler in order to achieve both low latency and high utilization for the typical latency-sensitive, user-facing workloads at Google. Leverich et al. [12] also propose an improved Linux CPU scheduler. An evaluation of this scheduler shows that the 95th-percentile latency is negatively affected by co-located workloads but this decrease does not devolve to asymptotic delays. Therefore, more research is needed to evaluate the impact of container orchestration middleware on 95th or 99th-percentile latency

in an experimental setting that compares software running in containers on top of VMs versus the same software natively installed in the Linux OS of the VMs.

Slominski et al. [5] present their findings on the usefulness of containers for migrating a legacy web application to a multi-tenant application. Several works analyze the influence of container technology on the architecture of PaaS clouds [10, 13].

7. CONCLUSION

This paper outlined our architectural vision on leveraging container orchestration technology such as Kubernetes to build flexible, cost-effective and trustworthy multi-tenant SaaS applications, and included our in-depth SWOT analysis of such an approach. We have contrasted this approach to the existing state of the art of building multi-tenant architectures on top of complex application-level middleware services, and highlight potential synergies (i) to strengthen and simplify these middleware services by relying on container orchestration platforms such as Kubernetes, and (ii) to help addressing open problems in container orchestration middleware such as the automated management of service dependencies. As we report on our ongoing efforts, clearly more research is required to better understand the true potential and risks of container orchestration for multi-tenant SaaS applications.

8. REFERENCES

[1] The kubernetes resource model. https://github.com/kubernetes/kubernetes/blob/release-1.3/docs/design/resources.md.

[2] Running multi-node kubernetes using docker. https://github.com/kubernetes/kube-deploy/tree/master/docker-multinode.

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[4] Sandeep Dinesh. Mongodb replica sets with kubernetes,. https://medium.com/google-cloud/mongodb-replica-sets-with-kubernetes-d96606bd9474.

[5] Aleksander Slominski et al. Building a multi-tenant cloud service from legacy code with docker containers. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 394–396, 2015.

[6] Dana Petcu et al. Portable cloud applications - From theory to practice. *Future Generation Computer Systems*, 29(6):1417–1430, 2013.

[7] Miguel G. Xavier et al. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 299–306, 2014.

[8] Miguel G. Xavier et al. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260, 2015.

[9] Quinton Hoole. Kubernetes cluster federation,. https://github.com/kubernetes/kubernetes/blob/release-1.3/docs/proposals/federation.md.

[10] Nane Kratzke. A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms. *Open Journal of Mobile Computing and Cloud Computing*, 1(2):17–30, 2014.

[11] Kubernetes. Sharing a cluster with namespace,. http://kubernetes.io/docs/admin/namespaces/.

[12] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[13] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[14] Ansar Rafique, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. Policy-driven data management middleware for multi-cloud storage in multi-tenant saas. In *2nd IEEE/ACM International Symposium on Big Data Computing*, pages 78–84. IEEE, December 2015.

[15] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy C. Bavier, and Larry L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 275–287. ACM, 2007.

[16] MongoDB (TM). Running mongodb as a microservice with docker and kubernetes,. https://www.mongodb.com/blog/post/running-mongodb-as-a-microservice-with-docker-and-kubernetes.

[17] Eddy Truyen. Kubernetes on openstack. https://github.com/eddytruyen/kubernetes_on_openstack.

[18] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. *Eurosys*, 2015.

[19] Ryan Wallner. Tutorial: Deploying a replicated redis cluster on kubernetes with flocker, https://clusterhq.com/2016/02/11/kubernetes-redis-cluster.

[20] Stefan Walraven, Wouter De Borger, Bart Vanbrabant, Bert Lagaisse, Dimitri Van Landuyt, and Wouter Joosen. Adaptive performance isolation middleware for multi-tenant saas. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 112–121, December 2015.

[21] Stefan Walraven, Eddy Truyen, and Wouter Joosen. Comparing paas offerings in light of saas development. *Computing*, 96(8):669–724, August 2014.

[22] Stefan Walraven, Dimitri Van Landuyt, Ansar Rafique, Bert Lagaisse, and Wouter Joosen. Paashopper: Policy-driven middleware for multi-paas environments. *Journal of Internet Services and Applications*, 6(1), January 2015.

[23] Johannes Wettinger, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel. *Future Generation Computer Systems*, 2015.

[24] Mingwei Zhang, Daniel Marino, and Petros Efstathopoulos. Harbormaster: Policy Enforcement for Containers. *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 355–362, 2015.