2016 IEEE International Conference on Cloud Engineering

Container and Microservice Driven Design for Cloud Infrastructure DevOps

Hui Kang, Michael Le, Shu Tao
IBM T. J. Watson Research Center
{kangh, mvle, shutao}@us.ibm.com

***Abstract*—Emerging container technologies, such as Docker, offer unprecedented agility in developing and running applications in cloud environment especially when combined with a microservice-style architecture. However, it is often difficult to use containers to manage the cloud infrastructure, without sacrificing many benefits container offers. This paper identifies the key challenges that impede realizing the full promise of containerizing infrastructure services. Using OpenStack as a case study, we explore solutions to these challenges. Specifically, we redesign OpenStack deployment architecture to enable dynamic service registration and discovery, explore different ways to manage service state in containers, and enable containers to access the host kernel and devices. We quantify the efficiency of the container-based microservice-style DevOps compared to the VM-based approach, and study the scalability of the stateless and stateful containerized components. We also discuss limitations in our current design, and highlight open research problems that, if solved, can lead to wider adoption of containers in cloud infrastructure management.**

I. Introduction

Infrastructure as a Service (IaaS) provides users with immediate access to scalable and seemingly "infinite" infrastructure resources without the expensive overhead and burden of having to maintain and manage their own datacenters. To offer IaaS, providers must deploy complex software stacks (aka., cloud infrastructure management software) to allocate, manage, monitor, and configure the underlying physical resources, e.g., CloudStack [1], Eucalyptus [2], and OpenStack. With IaaS quickly becoming commoditized and with businesses relying more on IaaS to serve as a foundation on which more lucrative managed services are built, the need to efficiently deploy and operate these infrastructure services grows in importance.

Recent advancements in container technology have led to renewed interests in using containers in the cloud. PaaS systems, e.g., CloudFoundry [3], Heroku [4], OpenShift [5], have long been the early adopters of containers. In these systems, containers are commonly used to host and allocate resources to applications. More recently, efforts to containerize cloud infrastructure services have emerged, i.e., running the infrastructure management software in containers [6], [7], [8]. Whether applied to applications or to cloud infrastructures, containers are attractive as they provide good isolation with low overhead and fast start-up time leading to highly agile solutions [9], [10]. This agility can be further amplified when combined with a microservice architecture where software is broken into small functional units; each unit can independently fail, scale, be maintained and reused [11].

DevOps is a set of techniques for streamlining and integrating the software development process with the deployment and operations of said software. Some of the DevOps challenges include, for example, how to easily deploy and upgrade cloud infrastructures, how to smoothly deliver code from development to production, how to scale the infrastructure with less human intervention, etc. Getting DevOps correct is one of the keys to the overall success of cloud-based software solutions. When dealing with cloud infrastructure software, besides the challenges of DevOps, being able to efficiently make use of the underlying physical resources is also critical, i.e., how to run more services with less resources. Efficient DevOps combined with efficient use of resources leads to high IT performance, which is a must for today's highly competitive IaaS business. Being highly agile and lightweight, containers combined with the microservice architecture have shown great promises in achieving the desired level of efficiency.

Although promising, significant challenges remain in containerizing and applying the microservice design principles to cloud infrastructure services. First, since infrastructure management software are broken into individual components with different roles, it is unclear what are the definitive interfaces between these separate components in the microservice architecture. Second, containerizing stateful components of the infrastructure layer (e.g., database, messaging server, in-memory key-value store) is difficult, due to limited support for data portability in containers [12]. Although such challenges also apply to the application layer, cloud applications can live with workarounds, such as running stateful components, e.g., databases, as external services [3] (therefore, obviating the need for containerization). However, this is a sub-optimal solution for infrastructure software, because service states are an integral part of the runtime environment. Third, infrastructure services often interact with physical resources such as compute, network, and storage. While physical resources are increasingly becoming "software-defined," they still lack common, programmable interfaces. This fundamentally impedes portability as a containerized service built for one system may fail to interface with the physical resources on another.

In this paper, we explore both the opportunities and challenges in using containers and applying the microservice design principles to operate and manage cloud infrastructure services. The goal of this work is to develop a more agile, reliable, and efficient DevOps approach for such infrastructure software. Specifically, to make the discussion more concrete, we present the design, implementation, and deployment of a microservice based Dockerized OpenStack instance. We use this instance to measure and quantify different aspects of operational efficiency discussed above. Lessons from our deployment along with limitations of our approach will also be discussed along with key open research problems, that if resolved, can lead to wider adoption of containers in the cloud.

The remainder of this paper is organized as follows: in Section II we describe how containers and microservice-style design can be used to achieve operational efficiency.

978-1-5090-1961-8/16 $31.00 © 2016 IEEE
DOI 10.1109/IC2E.2016.26

In Section III, we describe our prototype which is the application of microservice style DevOps to OpenStack using Docker containers. In Section IV, we present measurements quantifying operational efficiency of our prototype architecture to a reference OpenStack architecture. Section V discusses limitations of current implementation and related work is presented in Section VI. Section VII concludes the paper.

II. Cloud Infrastructure DevOps

In this section, we discuss how containers and microservice-style design can be used to achieve high operational efficiency for cloud infrastructure software. We start with discussing the characteristics of cloud infrastructure software and their impact on DevOps (Section II-A), how container and microservice-style design can improve operational efficiency (Section II-B), and finally, the challenges of applying container and microservice-style design to cloud infrastructure software (Section II-C).

A. Characteristics of Cloud Infrastructure Software

Large Configuration Space: Due to the need to support and anticipate a wide array of use cases, infrastructure software typically provides a myriad of configuration options. These configuration options allow the fine-grained tuning of software and hardware components of the infrastructure. Unfortunately, such abilities result in increased complexity which in turn can cause operational failure. Such failures are often caused by improper setting of certain parameters by operators who do not have the detailed knowledge or good understanding of the software as the development team. To reduce the chance of failures, only a minimum set of configurations should be exposed to the software.

Loosely Coupled Components: Cloud infrastructure software is often composed of many components (e.g, database, networking, computing, storage, etc.). These components collaborate with each other to provide the overall function of managing infrastructure resources. For the sake of performance and high availability (HA), each component is instantiated in more than one instances (e.g., multiple virtual routers). When new instance joins or existing one fails, the rest of the cluster must be informed of such event. Therefore, to make it easier to manage the cloud infrastructure, individual components should be loosely coupled.

Short Release Cycles: To adapt to constant changes in user demands and needs, features and fixes to infrastructure software are constantly being added. Updates to different components are done by different teams and with different release cycles. For example, OpenStack has a 6-month release cycle with milestone releases every one and a half months. Each release can contain hundreds of features and bug fixes. For instance, from OpenStack IceHouse to the Juno release, more than 300 new features were added and more than 3000 bug fixes committed across all components and libraries [13]. To keep up with the rate of updates, operators need automation and continuous integration of updated software.

Runtime Consistency: Infrastructure software requires the runtime environment to be configured with appropriate OS version, kernel modules, and supporting libraries. These dependencies are more restrictive and numerous than application-layer software. For example, a successful deployment/upgrade in a development environment does not necessarily imply the same in the staging or production environment where the OS or library versions might differ. Hence, mechanisms to enable consistent execution environment is critical.

TABLE I. Number of configuration points in deploying OpenStack: Chef-based vs Docker image-based approaches.

Complexity metric	Chef-based	Docker-based
Dependencies	22	5
Download links	$>$ 50 https	6 images
Configurable variables	80	26

B. Container and Microservice-based DevOps

Software-defined systems have enabled developers to treat infrastructure as code. This has allowed for the possibility of infrastructure to be managed in an automated, flexible, and efficient manner. Unfortunately, due to the nature and complexity of the infrastructure software (see previous section), many of these potentials are not fully realized. By leveraging containers and microservice-based designs to achieve code portability, ease lifecycle management, and utilize resources more efficiently, developers come closer to unlocking the full potential of software-defined systems.

Code Portability: One of the most appealing features of container is its support for code portability. By packing software along with its dependencies into a single image [14], [15], [16], container enables image-based deployment process, which offers the freedom of "develop once, deploy everywhere." It should be noted that while image-based management can also be achieved with VMs, VMs are not as portable nor lightweight as containers, thereby reducing the effectiveness of the approach.

The image-based approach can alleviate many of the pain points discussed above with regards to configuration complexity and runtime consistency of infrastructure software. Conventional deployment of cloud management software often relies on automation tools (e.g., Chef, Puppet) that tend to create many configuration points, in order to codify the dependencies between software and the deployment environment. This makes the install/uninstall or update process cumbersome. If not handled carefully, these configuration points can turn into break points or errors. In contrast, running services in containers simplifies the deployment process, as most of the software dependencies are packaged in a single image, leaving few necessary parameters for operators to configure.

To illustrate the above point, Table I compares the number of configuration points in OpenStack installation, when using Chef cookbooks [17] or Docker [14] container images developed for our own cloud operational environment. As can be seen from this example, the image-based approach eliminates much complexity in the configuration steps, thereby simplifying the entire deployment process.

Lifecycle Management: As discussed in the previous subsection, today's cloud infrastructure software demands continuous delivery and integration. To satisfy this demand, operators prefer managing the lifecycle of the infrastructure code using image-based approaches (e.g., versioned Docker images), so

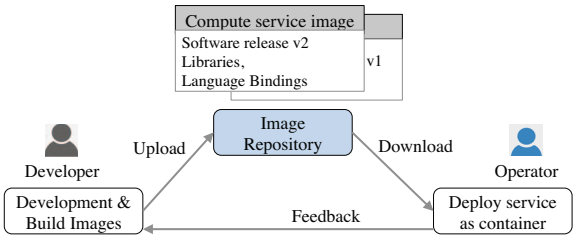


Fig. 1. Lifecycle management of infrastructure code using container images.

as to avoid in-situ updates. Furthermore, they aspire to capabilities such as easy deployment and tracking of different releases simultaneously in different environments, online "A/B testing," fast roll back, and so on. When problem occurs in a new release, operators do not need to debug and fix them in a production environment. Instead, they simply roll back to the previous container image (see Figure 1) and wait until new images with fixes become available. Image-based approaches combined with a microservice-style design amplify the benefits as components of the system are decoupled preventing interfering with one another during maintenance.

Resource Utilization: Container increases the utilization of computing resources from two aspects. First, like VM, multiple containers can be hosted on a single server to increase its resource utilization. Second, containers are more efficient than VMs in accessing memory and I/O devices, while still providing good isolation between applications in different containers [10]. Therefore, at single-host level, containers can run more cloud services than VMs, while achieving the same performance. Moreover, lightweight, portable and fast-booting container images facilitate the application of microservice-style design by running many cloud infrastructure components as "micro-services" that can be scaled out (or back) easily with workload changes. This leads to more efficient resource utilization at multi-host level.

C. Design Challenges

This subsection describes the three main challenges associated with applying containers and microservice-style design to cloud infrastructure management software.

Minimize Cross-configuration: Cloud infrastructure software such as OpenStack, often adopts a componentized and distributed architecture. Hence, different service components naturally become the units for containerization. How these individual components are managed and organized can greatly affect the portability and scalability of the design (see Subsection III-A).

Some services are typically replicated for fault tolerance and high availability (HA) purposes. With a containerized approach, one of the main challenges is to allow multiple instances of the same service to be instantiated from the same container image while still allowing some level of customization and configuration of the service. Too many configuration points would greatly complicate an image-based deployment (see Subsection II-A) while too few can make the container image less portable. Another challenge is to minimize the amount of cross-configuration that occurs in the overall setup. Cross-configuration occurs when changes to one component will cause configuration updates on all other components. Hence, adding or removing a service can result in impacting all other existing services. This phenomenon can be found in conventional HA setups using cluster management tools such as Pacemaker [18] where each node must keep track of the IP addresses of all other nodes in the cluster. Such approach is not suitable for a microservice-style deployment.

Maintain State: The cloud infrastructure layer manages many software state. One type of state is the runtime configurations of the cloud infrastructure, e.g., service IP addresses and ports, master-slave relationship in HA setups, etc. These runtime configurations should be managed in a way that suits container-based deployment, i.e., 1) few configurations need to be managed outside of the containers, and 2) any configurations that have to be managed outside of the container can automatically be fed to service registration and discovery processes (see Subsection III-A).

The other type of state comes from the running application in the container. Application state can either be on disk or in memory. For example, a database may store user profiles on disk, while messages between components may be stored in a messaging server's memory. To be fault-tolerant and/or support the dynamic scaling out of services, mechanisms must be developed to allow the replication of application states.

For on-disk state in the container, existing options include using shared storage, disk volume migration [19], and application-level data replication. Depending on the applications, these choices lead to different performance and scalability trade-offs. For in-memory state, however, there is no mechanism available yet for containers to transparently perform memory replication. In this case, the current practice is for containerized services to avoid storing state in memory altogether (e.g., caching), or rely on the clients to tolerate the errors caused by the loss or inconsistent state of failed services. We discuss some of our efforts in dealing with state in memory in Subsection III-C.

Provide Host Resource Access: The lack of a dedicated kernel in a container makes it difficult to access host resources needed by some infrastructure service components. For example, some services need to interact with the kernel through `IOCTL` or `/proc`. Others need to access the host's hardware device to handle high performance tasks (e.g., a software router). One potential solution is to enhance the capability or privilege of these containers. However, this decreases the portability of such containers, and may result in a conflict when multiple privileged containers access the same host resource.

III. Case Study: Applying Containerization and Microservice-style Design to OpenStack

OpenStack is a widely adopted, open source infrastructure cloud management software. Its distributed, service-oriented architecture makes it an ideal candidate for demonstrating the feasibility of containerizing cloud infrastructure software. In this section, we focus on addressing the challenges identified in Subsection II-C, while preserving the benefits of containerization discussed in Subsection II-B.

A. Container-based Deployment

A typical OpenStack architecture is comprised of three types of nodes: control plane, compute, and network. The control plane nodes run management services, while compute nodes and network nodes are responsible for provisioning VMs and network resources.

Services across the different nodes can be containerized at different levels of granularity, based on the DevOps requirements. In our current implementation, we opt for a relatively simple setup for the services in the control plane: one container hosts the *database* service (MySQL), another hosts *messaging* service (RabbitMQ [20]), and the third hosts the *controller* services (including Keystone, Glance, Nova, Neutron servers, etc.). To support HA, each control plane container is replicated on two additional hosts. On the compute nodes and network nodes, there is one container that runs the Nova and Neutron agent services. Figure 2(a) shows the Docker container-based OpenStack deployment, which we will refer to throughout the rest of the paper.

Each service in OpenStack consists of its application code and configuration files. We have built the application code together with all its dependent libraries into container images, which can be downloaded and used to create container on any Docker host. Meanwhile, container images contain application-specific scripts to generate configuration file at runtime.

B. Microservice Architecture

Containerizing OpenStack is more than just running those management services in containers. For example, in the control plane the controller container should be informed where the database and the messaging services are, whenever these services are newly created, recovered from failure, or shut-down. Specifically, this means the services must have ways to register themselves, be discovered by other services, record their configuration/runtime state, and be generally orchestrated in their deployment and update processes. Therefore, we need an architecture that eases the operation of scaling-out, HA, and load balancing. Based on these requirements, the key components of the proposed architecture include:

- *Service Proxy:* all services are configured to be only accessible via a proxy (e.g., HAProxy [21]), such that changes to a service instance do not affect others in the HA setup. Meanwhile, the proxy must have the up-to-date configurations of each service, e.g., IP address and port number. These configurations are retrieved from the configuration state manager.
- *Configuration State Manager (CSM):* manages the configurations of all services in the control plane. We implemented the CSM as a distributed key-value store based on `etcd` [22]. To register the service in CSM, each container has its "sidekick" process. The sidekick process monitors the status of the service in the container periodically (e.g., through test API calls); depending on the status, the sidekick process will register or de-register the service in the CSM.
- *Service Orchestrator:* all services are managed by an orchestrator (built on `fleet` [23]) that decides how many instances of a service need to be run and where to run, based on certain policies (e.g., co-location or anti-colocation). The orchestrator not only launches containers along with their sidekicks on the same host, but also tracks the status of the container and takes action accordingly. For example, if an existing container is terminated due to rebooting the host, the orchestrator will relaunch a new one automatically.

Figure 2(b) illustrates how these three components cooperate in the proposed architecture. In step (1), the service orchestrator launches a controller container and its sidekick process on a host. In step (2), when the sidekick process finds that the service is up and running, it registers the well-known service name (e.g., `/controller/ip`) and the IP address (e.g., `172.16.1.101`) as a key-value pair in the CSM. A `confd` [24] process running with the proxy watches the changes to the key-value pair in CSM, generates new configuration files, and reloads the HAProxy process to make use of the new configurations (step 3). Later, if the controller container or service in the container fails, the service orchestrator will be informed and start a new controller container. The same service registration process will then commence.

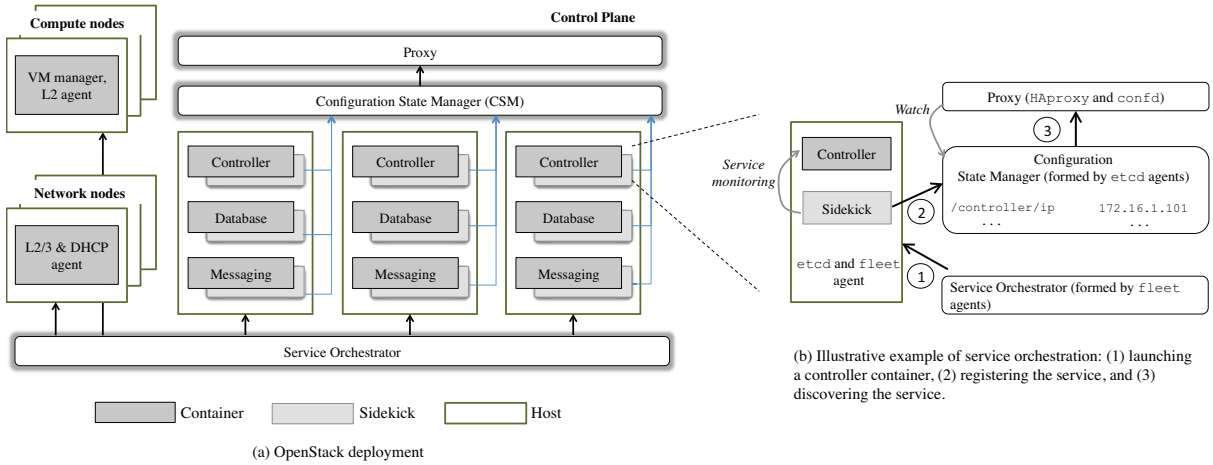
C. Handling Stateful Services

The architecture described above offers clear benefits to stateless services in OpenStack. However, handling stateful services requires careful considerations, so as not to obviate the benefits from containerization. The first type of service states are the application data residing *on disk*. The main challenge is how to efficiently replicate these states in an HA setup. Here we use MySQL database as an illustrative example.

MySQL supports two HA modes: active-standby and active-active. In the active-standby mode, only the master node serves client requests, while the remaining nodes are slaves and replicate data from the master. This mode requires distinction between master and slaves roles and triggers cross-configuration during failover or scaling out. In the active-active mode, every node in the cluster are identical and actively serve client requests; the joining or loss of individual nodes does not impact the others. Therefore, the active-active setup is more suitable for the container-based architecture.

Since every node in the active-active MySQL cluster can update its local state, consistency must be guaranteed by synchronizing data between instances. Common approaches for data synchronization are shared storage and application-level replication. In the former case, all MySQL servers access the same data volume via a distributed filesystem such as Global File System 2 (GFS2) or OCFS. In the latter case, the database process (e.g., Galera patched MySQL [25]) synchronously replicates data among all nodes in the cluster. In both setups, data on disk survive the loss of any database server. However, as presented below, they have different performance and scalability trade-offs.

Table II compares the latency of accessing the database using shared storage and application-level data replication. In this experiment, two MySQL containers across two hosts form an HA cluster, where data synchronization is either achieved by GFS2, or by Galera replication. We use SysBench [26] to simulate various workload combinations and numbers of concurrent users accessing both servers. We observe that while



(a) OpenStack deployment

(b) Illustrative example of service orchestration: (1) launching a controller container, (2) registering the service, and (3) discovering the service.

Fig. 2. A container-based microservice architecture of OpenStack. (a) OpenStack deployment, (b) Service registration and discovery in microservice architecture.

TABLE II. LATENCY OF 95^{th} PERCENTILE OF REQUESTS: GFS2 SHARED STORAGE VS. GALERA-BASED DATA REPLICATION.

Workload type	10 threads		100 threads	
	GFS2	Galera	GFS2	Galera
Read-only	51ms	57ms	966ms	1082ms
Read-write	2981ms	27ms	28482ms	1845ms

the results for read-only workload are similar, the Galera setup significantly outperforms the GFS2 setup for the read-write workloads ($100\times$ and $15\times$ faster). This is because with shared storage, a write request will lead to filesystem lock on the entire table file (aka., table-level locking). In contrast, Galera allows multiple MySQL servers to update different rows in a table simultaneously, as it uses row-level locking. This leads to the superior performance of the Galera setup. However, the benefit of Galera setup comes at the cost of state replication during scaling out or recovery. That is, when a new MySQL container is created to join an existing cluster, DB states need to be transferred to the new server. Even though there are ways of minimizing such state transfer, e.g., through incremental state transfer (IST), data transfer still incurs network overhead and delays in starting new service instances.

The other type of states are the *in-memory* application data, such as network sessions or cached values. To handle in-memory state, mechanisms for container memory snapshotting, replication, and migration are required. These mechanisms provide operators the ability to gracefully handle planned host maintenance owing to OS/hardware upgrade or zero-downtime load balancing and failover for HA.

The abilities of snapshotting and restoring applications are available for both containers [27] and VMs [28]. With VMs, these mechanisms have been widely adopted by many modern VM management tools such as VMware's vCenter, virt-manager, etc. Linux implementation of container memory snapshotting also exists for LXC [15] containers but unfortunately, not for Docker containers. We have worked with the open source community to integrate the abilities of snapshotting LXC containers through the CRIU [29] project to Docker containers [30]. In essence, snapshotting involves dumping all the runtime information (e.g., mapped memory page, content of CPU registers, pending signals) of the processes in a container along with the context of the container (e.g., namespaces, network interfaces, configuration) into a set of image files located locally on disk. The size of the image files are proportional to the amount of memory actively used by the container. The abilities to take a snapshot of the memory state of a container is the foundation for other services such as replication and migration, which are also under development by the community.

Given the immaturity of this technology, we do not yet handle in-memory state of our deployed OpenStack services. We do, however, provide some evaluation of the performance of container memory snapshotting abilities in Section IV.

D. Handling Services Accessing Host Resource

In OpenStack, infrastructure services on compute, storage and network node are responsible for creating VM instances, provisioning storage and network resources, respectively. As these resources are tightly coupled with the host OS and require some degree of privilege to access, it is a challenge to run compute, storage or network functions inside containers in a safe and portable way as explained below.

Containers that run services that need access to host resources require both privileged access to the host kernel and access to specific directories on the host. As a consequence, this can jeopardize the security of the host. For example, the `/proc/modules` directory must be shared so that the container can load kernel modules, such as KVM, to launch VM instances [31]. Access to this directory allows a container access to other modules on the host that it is not privy to. This lack of fine-grained control for privileged access is a limitation of current container technology that we will briefly discuss in Section V.

Fortunately, not all services that require access to host resources need to run in privileged containers. For instance, containers requiring access to data volumes on the host for storing VM images do not require privileged access.

IV. EVALUATION

This section evaluates and validates the design choices discussed in Section III. Specifically, we highlight the benefits

TABLE III. DEVOPS PRACTICES THAT IMPROVE IT PERFORMANCE [33]

DevOps Practices	IT Performance Metrics		
	Deployment Frequency	Time for changes	Mean Time to Recover from Failure
Continuous delivery	•	–	–
Version control	•	•	–
Monitoring system & Application health	–	–	•
Automated test	–	•	–

of using containers and microservice-style architecture for DevOps of OpenStack over more conventional VM-based approaches. We do not present analysis of script-based DevOps mechanisms such as using Chef or custom shell scripts as these techniques lack the agility, flexibility, and speed necessary for efficient DevOps in the cloud. For example, script-based techniques are time consuming because they require downloading and installing software packages when new instances of services are instantiated. In addition, they are error prone due to the complex logic needed to handle installation and configuration failures thus requiring specialized knowledge from developers to debug (see results in Table I).

In subsection IV-A, we briefly describe common DevOps practices that are most critical to IT and the main differences between DevOps using VM-based vs. container-based approaches. Specifically, we show that OpenStack DevOps using containers benefit greatly from (1) fast deployment time, (2) ease of rolling upgrade, and (3) simplification of failure recovery and HA. In subsection IV-B, we assess how well the control plane scales under varying workload.

Our experimental setup consists of machines connected by a 1Gb switch. Each machine has two Xeon E5649 processors with 64GB RAM and runs Linux 3.13. KVM is used as the hypervisor to instantiate VMs on the Linux hosts. For experiments using containers, we use Docker 1.6. Each Docker host also runs *etcd* and *fleet* agent to form the complete microservice architecture. These orchestration tools are available in most Linux distributions hence the selection of a particular Linux distribution (e.g, Ubuntu, CentOS, CoreOS [37] is not critical to our evaluation. For the OpenStack control plane, we use components in OpenStack Juno release, MySQL Galera, and RabbitMQ. We use the Rally benchmark tool [32] to generate OpenStack workload. For experiments involving container memory state snapshotting, we use CRIU 1.7. The architecture of the experimental testbed is shown in Figure 2.

A. DevOps Practices: VM- vs Container-based Approaches

OpenStack DevOps refer to a wide range of tasks that manipulate the lifecycle of individual OpenStack components. The most important ones are summarized in the first column of Table III. The latencies of performing these operations (*deployment*, *rolling upgrade*, and *failure detection/recovery*) are used to quantify the efficiency of container vs. VM-based approaches.

Table IV compares the time to perform the three operations using VM-based and container-based approaches. The results show that container-based approach outperforms VM-based approach in terms of time to perform the DevOps tasks. In general, the major differences between between VMs and containers with respect to DevOps are the provisioning overhead and speed. Bringing up a container is not only faster than bringing up a VM, but uses less system resources. Besides the fast container creation time, container images are composed of different layers, a feature that avoids having to transfer the complete history of the image, resulting in faster image transfer time and more fine-grained version control. Detailed analysis of each DevOps task are provided in the following paragraphs.

TABLE IV. EXECUTION TIME OF OPENSTACK DEVOPS TASKS: VM OR CONTAINER-BASED APPROACHES (IN SECOND)

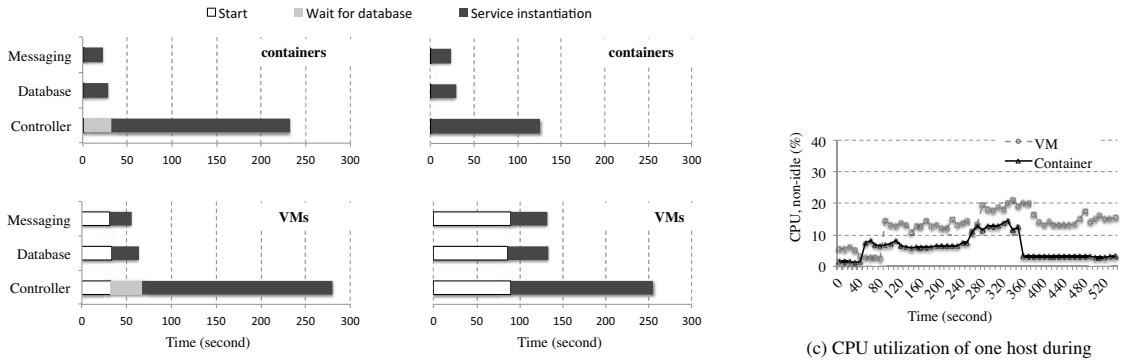
DevOps Tasks	VM	Container
Deployment	535	358
Upgrade the three controller instances	326	207
Failure recovery of a MySQL instance	74	32

Deployment (validating Section III-A): As shown in Figure 2, the OpenStack control plane consists of three types of nodes: messaging, database, and controller. The deployment of these nodes has two phases: initialization and scale out. In the initialization phase, we create one instance of each node type on each host (create a VM/container) to stand up a fully functional OpenStack control plane. We then scale out the services by adding another two instances of each type to form an HA cluster. As shown in row one of Table IV, container-based deployment is $1.5\times$ faster than VM-based deployment.

To explain why the container-based approach leads to faster deployment, we show the break down of the deployment time of each node type in Figure 3. Figure 3(a) shows the initialization phase of the first instance of each node type. The instantiation of database and messaging are in parallel (dark gray), while the instantiation of services in the controller instance (light gray) has to wait for the completion of the database and messaging components because services in the controller node need to register users and endpoints in the database and certain services (e.g., Nova conductor) coordinates via the messaging service. As can be seen, the controller node takes the longest time to finish and its duration is almost the same between VM and container. Therefore, although launching a container is much faster than a VM (specifically, $<$ 1s vs. 30s), the actual benefits of using containers in completing the initialization phase across the three node types is relatively small.

In contrast, the scale out time when using containers is reduced by $>$ 50% as shown in Figure 3(b). During the scale out phase, we start two additional VMs/containers of each node type. While it takes the same amount of time to spin up the containers as in the initialization phase, the startup time of individual VM is tripled (90s vs. 27s). This is the result of resource contention caused by VM creation (Figure 3(c)). With either serial or parallel creation of the instances, VMs typically cost more system resources and incur more interferences than containers, leading to longer startup time. Therefore, for the entire deployment, the container-based deployment finishes more quickly than VM-based approach. Further, as initialization is a one-time process for the OpenStack deployment, we expect more benefits from scaling out services using containers.

Rolling upgrade (validating Section III-B): A rolling upgrade is an upgrade procedure that causes zero-downtime. This ability is essential to operators to deal with the high rate of bug fixes and new features typical in modern cloud deploy-



(a) Initialization (b) Scale out (c) CPU utilization of one host during deployment

Fig. 3. Deployment comparison: containers vs. VMs. (a) Deploy one instance of each type on three hosts respectively; (b) Scale out instances on three hosts; (c) Host CPU utilization.

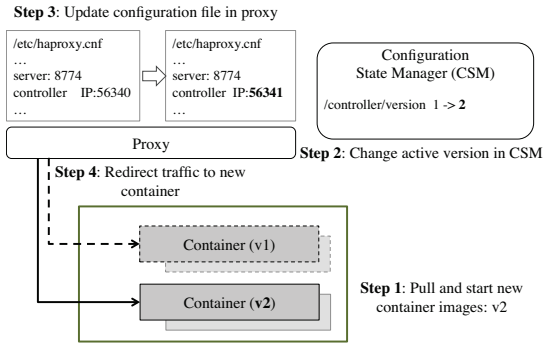


Fig. 4. Rolling upgrade in a container-based microservice style architecture.

ment. Figure 4 illustrates how rolling upgrade is performed in a container and microservice style DevOps. In step 1), a container containing upgraded software is created on the same host as the existing service. Then the active version of the service is updated either manually or automatically triggered by the upgrade event (step 2). By continuously watching the `version` key, the configuration file of the proxy is updated (step 3), followed by redirecting the traffic to the service running in the new container (step 4). This process is transparent to the external clients, causing minimal interruption to the other components of OpenStack.

Image-based DevOps, regardless of VM-based or container-based simplifies rolling upgrade by eliminating the complexities during the conventional software upgrade process (such as uninstalling/installing the old/new binaries along with dependencies and resolving conflicts). In essence, the upgrade process is to replace the existing VM/container with a new instance while minimizing interruption of the provided service. Since the size of container image is typically smaller than that of a VM and starting a container is faster than a VM, the total time to upgrade a service using containers, is therefore, faster than using VMs. This can be seen in our results (Table IV, 2nd row). Replacing the three controller containers in the testbed is 1.6× faster than replacing three VMs running the same service. However, while not shown, the downtime experienced by the two approaches is not noticeably different.

Failover/recovery (validating Section III-C): Service availability is critical in any production environment, but failure is inevitable. A service becomes unavailable due to either planned maintenance or unplanned outage. Depending on the cause of the downed service, maintaining service availability requires performing different operations. For planned maintenance, typical solutions include snapshotting the runtime state of the service and later restoring the service on the same host (after it has been fixed), or temporarily migrating the service to a different host. The solution to unplanned service outage is more complicated and relies on an HA framework with support for continuous redundancy. In the following paragraphs, we will compare the differences in VM and container-based DevOps with regards to failover and recovery.

High availability: The design of a high available system aims at minimizing the number of failed transactions during service outage and quickly bringing system back online. Moreover, given the diversity of the services in OpenStack, from DevOps perspective, it is desirable that the HA framework can provide a unified and simplistic interface to manage these services.

For existing VM-based approach, failure detection/recovery relies on clustering software such as Pacemaker (approach recommended by the OpenStack installation guide), while with containers we can leverage generic service discovery/registration mechanisms such as *etcd* and *fleet*. In Pacemaker, individual services must use a Pacemaker specific way to write their own custom resource agents that monitor service availability and run application-specific scripts to perform service recovery after failure. In our microservice architecture, a simpler and more generic mechanism is used to monitor availability (e.g., shell scripts used by sidekick for probing) as well as for recovery (kill failed service and instantiate new service). In the following paragraphs, we first evaluate and compare the recovery time and transaction error rate of failed OpenStack services deployed in VMs using Pacemaker for HA with the container-based microservice style HA explored in this work. We then describe the complexity in implementing HA using containers vs. Pacemaker.

The first experiment compares the recovery time when failure occurs in the MySQL database component of OpenStack. Recovery time is determined by the latency to detect the failure and the time to recover the failed service. In this work, the detection latency is configured to be the same for both approaches. Hence, the major difference between

TABLE V. PERCENTAGE OF FAILED REQUESTS DURING SERVICE FAILOVER AND THE RECOVERY TIME. 240 REQUESTS IN TOTAL.

Failover stats	Failed container type	
	Controller	MySQL
Error percentage	5.8%	1.8%
Recovery time	48s	32s

the two approaches is the time to recover a failed instance. To cause a failure, we explicitly shut down a VM and a container, respectively[1]. As a consequence, the recovery action is to reboot the VM in the Pacemaker case and start a new container running the service in the container-based approach. As expected, the recovery of a failed container takes about half of the time rebooting a VM (row three in Table IV).

The second experiment examines the transaction error rate during node failure. In active-active HA mode, the traffic intended for the failed instance will be directed to the remaining nodes. The failed instance can still return error for those outstanding transactions before the proxy can redirect traffic. We run Rally to generate transactions of adding new users and querying existing users. Table V lists the percentage of failed requests under different failed node types. As can be seen, failure of the MySQL server led to 1.8% failed requests, while that of the controller led to 5.8% failed requests. This is because not all user requests trigger database transactions (controller node uses `memcached` to cache the returned value of frequently issued requests in order to avoid unnecessary database transactions). Therefore, when the controller container fails, all in-memory state containing ongoing sessions are lost, thus resulting in the observed higher error rate.

The HA framework in the microservice architecture simplifies the process of managing, monitoring, and recovering OpenStack services. In a Pacemaker cluster, each service is considered as a resource and managed by an executable (aka., resource agent [34], [35]). The resource agent has to follow certain specifications such as Open Cluster Framework (OCF), a complexity that not only imposes additional burden on the developers, but is also known to be notoriously difficult for operators to debug. Further, since services are different from each other, operators has to manipulate resource agents for individual resources. In a microservice architecture, all resources are deployed in containers. Thus, the HA framework only requires implementing the mechanisms to monitor the status of corresponding services. Regardless of how the service fails, the HA framework will perform the same operations: kill the existing container and create a new one in its place. By following such "cattle vs. pet" principle, HA in microservice architecture achieves simplicity, flexibly, and efficiency, compared to conventional HA approach.

Snapshot/Migration: Snapshot and migration mechanisms provide the ability to recover the system from planned host maintenance owing to hardware upgrade, soft reboot, etc. However, snapshot and migration introduce overhead when persisting state to local disk or transferring state across hosts. In the following paragraphs, we measure such overhead with respect to time and size of state for snapshotting and migrating the VM/container holding the MySQL database service in OpenStack.

[1]It is possible to just cause the MySQL process to fail but recovery of such failure is more complicated and the time to start a new MySQL process would be close to starting a new container.

TABLE VI. SNAOPSHOT/MIGRATION COMPARISON OF VM AND CONTAINER (THE SERVICE INSIDE IS MYSQL HANDLING OPENSTACK KEYSTONE USER CREATION REQUESTS

Snapshot/migration cost	**VM**	**Container**
Image size	530MB	107MB
Snapshot duration	7s	$<$ 1s
Restore duration	2s	$<$ 1s
Image transfer duration (via scp)	5s	$<$ 1s

To perform a VM snapshot, we use `virsh save` and for Docker containers, we use `docker checkpoint`. Given the immaturity of container memory migration mechanisms (see subsection III-C), we emulate migration by using *scp* to send the checkpointed state from the source to the destination host.

Table VI shows the results of our experiments. For both VM and container, the majority of the snapshot operation involves dumping the memory pages into local disk. Thus, the duration is proportional to the resident set size (RSS) of the process. Since the VM process [2] typically consists of a large number of processes, the RSS of the VM is significantly larger than a container where there are a small number of microservice processes. Therefore, we observe that it takes about 7 seconds to snapshot the VM (RSS is 530MB), while the time is $<$ 1 second for container (RSS is 107MB). As expected, the smaller RSS of the container leads to faster transfer time for the pre-copy round of migration.

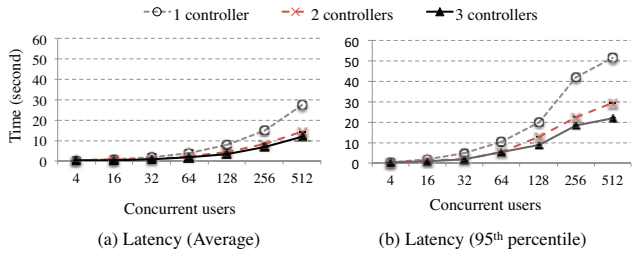
Overall, lightweight containers are faster to recover from failure and have less overhead in snapshot/migration. In addition, the simplicity of the HA model of the microservice architecture allows it to be more easily extended to support different services compared to existing HA model exemplified by Pacemaker.

B. Scaling

This subsection presents our evaluation of the scalability of the proposed architecture when handling increased number of requests. Workload requests are generated to stress the Keystone component (stateless) and the MySQL galera cluster (stateful), with a variety of concurrent connections. Since these workloads are neither CPU nor memory intensive, VMs and containers have comparable performances. Here, we report the results demonstrating the ability of containers in handling increasing workload by quickly instantiating new service containers into the system.

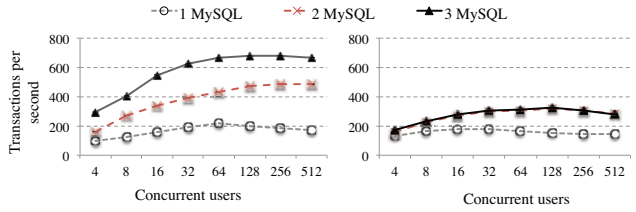
Figure 5 plots the Keystone's user authentication latency curves as a function of concurrent users under different number of controller containers. For each curve, the latency constantly increases as the number of concurrent uses varies from 4 to 512 – each Keystone process has to process more requests. As additional controller containers are added to the cluster, the load balancer can distribute less requests to each of the Keystone processes, leading to faster authentication time. Therefore, with higher number of concurrent users (from 128 to 512), the 3-controller case noticeably reduces the latency to more than half, compared to 1-controller case. Since containers can often be instantiated in seconds, the container-based microservice style DevOps allows the OpenStack control plane to better respond to workload variations in a timely fashion.

[2]A KVM VM is viewed as a single `qemu-kvm` process on the host.



(a) Latency (Average) (b) Latency (95th percentile)

Fig. 5. Average and 95th percentile latency of user authentication under different number of controller containers (lower is better))



(a) Throughput (Read-only) (b) Throughput (Read-write ratio: 3.5:1)

Fig. 6. Scale the database performance (higher is better) by increasing the number of MySQL containers under increased workload with fixed number of controller containers (three controller containers).

Figure 6 plots the throughput of the MySQL database with different cluster sizes. Workloads are generated by varying the number of concurrent users (up to 512) and the transaction type (read-only and read-write). The results demonstrate the performance benefit of scaling up the database cluster. For both workload settings, the 3-node cluster outperforms the 1-node and 2-node cluster. This happens because like Keystone, more database node in the cluster implies less overhead on individual node. For the read-write transactions (Figure 6(b)), however, the performance improvement is less than the read-only transactions (Figure 6(a)) since write operations can not be committed until the write transactions are certified by all the nodes in the cluster. The result is that adding additional nodes may decrease the write performance. We do not experiment with write-only workload, because in reality, OpenStack workloads are a mix of read and write transactions.

We note that although stateful components like databases have replication overhead for write operations, the overheads are tolerable since the write operation does not dominate the workload. As a result, when a bottleneck is found for certain components, it is possible to scale up the performance by provisioning more containers of the corresponding node type.

V. Limitations and Discussion

This work reveals several open issues in current container technology, in the context of containerizing infrastructure cloud services, a non-typical usage scenario for containers. The infrastructure services need access to OS-level facilities, e.g., kernel modules, in-memory state, and physical devices. Such requirements make it difficult to containerize these types of services without sacrificing portability and security, which we further elaborate below.

OS for Containers: Our current containerization approach for interacting with host OS is somewhat ad-hoc in that, in some cases, we run the containers in privileged mode and develop specific workarounds to circumvent restrictions imposed by the Linux kernel. This presents a barrier to containerization and exposes security risks.

To address this issue, we argue that OS should expose more kernel functions to containers via special interfaces. For example, fine-grained access control can be defined for privileged OS services (e.g., loading kernel modules) without sacrificing safety. Atomic [36] and CoreOS [37] are Linux distributions for containers but have complex mechanisms for specifying access control. Potential solutions include micro-kernel and multikernel OS architectures that expose selected kernel functions at the user-level [38], [39]. Thereby, containers can manage kernel services through well-defined interfaces. Multikernel further allows kernel state to be migrated between containers, which provides additional benefits to the cloud infrastructure services such as upgrading compute/network node without the need to reboot.

Container State Replication: We believe that container state replication should be further improved. Things to consider include weighing the overhead of replication with the new opportunities made possible by this mechanism in allowing the deployment of critical services in containers. With state replication enabled, we expect to see reduced error rate during failover (see Table V).

Container state replication can also be directly used for container live migration [40]. While there are some active development efforts in this area [29], it is worth debating whether this capability is really beneficial in a container world.

Direct I/O in Container: Having access to I/O devices (e.g., NIC) is critical for some infrastructure services (e.g., OpenStack Neutron). However, existing container mechanisms for accessing host devices are either inefficient (via NAT) [10], or ad-hoc (e.g., via privileged containers). We believe enabling direct I/O access from container solves the performance issue, but introduces portability and security concerns, especially when migration of these containers needs to be supported.

VI. Related Work

Our work is enabled by advancement in modern container technology and draws inspiration from best practices in deploying and operating large-scale distributed systems. Specifically, our design is largely based on techniques for delivering and managing applications using Docker container.

A large number of applications have been shipped by Docker, because Docker provides a portable execution environment including code, runtime, system library, etc. Examples found in Docker Hub [41] include well-known applications, from front-end web server (e.g., httpd, nginx) to backend data store (e.g., MySQL, redis, mongodb), from tiny OS (e.g., CirrOS) to full-fledged programming runtime (e.g., golang, ruby), from standalone analytic service (e.g, single Spark instance) to complex distributed applications (e.g., storm, hadoop). Our work extends the scope to cloud infrastructure management software such as OpenStack.

One recent project that also explores containerizing OpenStack is Kolla [8]. Kolla aims at providing production-ready containers that allow rapid OpenStack deployment with proper customization. However, Kolla lacks the support for either

dynamic configuration or high availability. Our work is orthogonal to Kolla because we show that the missing DevOps tasks in Kolla – scale out and failure recovery – are difficult to achieve with human intervention. We fully automate DevOps by integrating containers into the microservice architecture, making DevOps more efficient and scalable.

Microservice architecture requires an ecosystem of functionalities that allow container-based DevOps to obtain all the desired benefits. For example, like fleet, kubernetes [42] and Docker Swarm [43] are clustering tools that manage a cluster of containers across hosts as a single system. Consul [44], Eureka [45], SkyDNS [46], and ZooKeeper [47] are alternatives to etcd which provide key/value store to support dynamic service registration and discovery. Dynamic configuration can also be achieved by combining Registrator [48] and HttpRouter [49]. However, existing works built on these tools are limited to user-space applications. Instead of comparing different tools, this paper demonstrates that proper implementation of microservice architecture is suitable to accelerate the DevOps of infrastructure management software.

VII. Conclusion

We have discussed and shown, using OpenStack as a case study, that containerizing cloud infrastructure services and combining with a microservice style architecture greatly improve operational efficiency. When dealing with infrastructure services, we have shown that simply running the services in containers is not enough to reap the full benefits of containerization and microservice style design. We identified three main challenges to improving operational efficiency: (1) minimize cross-configuration of services, (2) maintain state of running services, and (3) provide safe access to host resources. We addressed most of these challenges and identified fundamental shortcomings of current container technologies that will need to be addressed before a complete solution can be had with using containers for infrastructure software. Our work have explored and evaluated different approaches for providing portable container images and handling stateful services. Combined with our proposed orchestration architecture, our prototype offers dynamic service registration and discovery which are essential for dealing with service scaling and failures. Based on our work, we identified areas for further research including: in-memory state replication, host kernel state management for containers, and efficient device access/sharing among containers.

References

[1] "Apache CloudStack," http://cloudstack.apache.org/index.html, Accessed 2015.

[2] "Eucalyptus cloud-computing platform," https://github.com/eucalyptus/eucalyptus, Accessed 2015.

[3] "CloudFoundry," http://www.cloudfoundry.org, Accessed 2015.

[4] "Heroku," https://www.heroku.com, Accessed 2015.

[5] "OpenShift," https://www.openshift.com, Accessed 2015.

[6] S. Murakami, M. Silveyra, D. Krook, and K. Bankole, "A practical approach to dockerizing OpenStack high availability," OpenStack Paris Summit, 2014.

[7] J. Labocki and B. Holden, "Docker all the OpenStack services," OpenStack Paris Summit, 2014.

[8] "Kolla," https://github.com/stackforge/kolla, Accessed 2015.

[9] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *SANE*, 2000.

[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *ISPASS*, 2015, pp. 171–172.

[11] J. Thones, "Microservices," *IEEE Software*, vol. 32, no. 1, p. 116, 2015.

[12] "The microservice revolution: Containerized applications, data and all," http://www.infoq.com/articles/microservices-revolution, Accessed 2015.

[13] "OpenStack Juno, Release Notes," https://www.openstack.org/software/juno/, Accessed 2015.

[14] "Docker," https://www.docker.com/, Accessed 2015.

[15] "LXC," https://linuxcontainers.org, Accessed 2015.

[16] "Rocket," https://coreos.com/blog/rocket/, Accessed 2015.

[17] "OpenStack chef repository," https://github.com/stackforge/openstack-chef-repo.git, Accessed 2015.

[18] "Pacemaker, a scalable high availability cluster resource manager," http://clusterlabs.org, Accessed 2015.

[19] "flocker," https://clusterhq.com/, Accessed 2015.

[20] "RabbitMQ," http://www.rabbitmq.com/, Accessed 2015.

[21] "HAProxy," http://www.haproxy.org/, Accessed 2015.

[22] "etcd," https://coreos.com/using-coreos/etcd/, Accessed 2015.

[23] "fleet," https://coreos.com/using-coreos/clustering/, Accessed 2015.

[24] "confd," https://github.com/kelseyhightower/confd, Accessed 2015.

[25] "Galera cluster documentation," http://galeracluster.com/documentation-webpages/, Accessed 2015.

[26] "SysBench: a system performance benchmark," https://launchpad.net/sysbench, Accessed 2015.

[27] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *USENIX*, 2007.

[28] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI*, 2005.

[29] "CRIU," http://criu.org/Main_Page, Accessed 2015.

[30] "Contributions to docker checkpoint and restore," https://github.com/huikang, Accessed 2015.

[31] B. Baude and S. Collier, "Running libvirtd in a container," http://www.projectatomic.io/blog/2014/10/libvirtd_in_containers/.

[32] "Rally," https://wiki.openstack.org/wiki/Rally, Accessed 2015.

[33] "2014 state of DevOps report," puppet report, 2014.

[34] "Combined repository of ocf agents from the rhcs and linux-ha projects," https://github.com/ClusterLabs/resource-agents, Accessed 2015.

[35] "Openstack resource agents," https://github.com/madkiss/openstack-resource-agents/ Accessed 2015.

[36] "Project atomic," http://www.projectatomic.io/, Accessed 2015.

[37] "CoreOS," https://coreos.com/, Accessed 2015.

[38] J. Liedtke, "On u-kernel construction," in *SOSP*, 1995, pp. 237–250.

[39] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *OSDI*, 2014, pp. 1–16.

[40] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration," in *Linux Symposium*, 2008, pp. 85–90.

[41] "Docker hub," https://hub.docker.com, Accessed 2015.

[42] "kubernetes," http://kubernetes.io, Accessed 2015.

[43] "Swarm: a docker-native clustering system," https://github.com/docker/swarm, Accessed 2015.

[44] "consul," https://www.consul.io, Accessed 2015.

[45] "Eureka," https://github.com/Netflix/eureka, Accessed 2015.

[46] "Skydns," https://github.com/skynetservices/skydns, Accessed 2015.

[47] "Apache zookeeper," https://zookeeper.apache.org, Accessed 2015.

[48] "Registrator," https://github.com/gliderlabs/registrator, Accessed 2015.

[49] "HttpRouter," https://github.com/julienschmidt/httprouter, Accessed 2015.