Understanding the Container Ecosystem: A Taxonomy of Building Blocks for Container Lifecycle and Cluster Management

Dominik Ernst, David Bermbach, Stefan Tai
Information Systems Engineering Research Group
TU Berlin
Berlin, Germany
Email: de,db,st@ise.tu-berlin.de

***Abstract*—Container technologies, skyrocketing in popularity, are diverse and difficult to compare. In particular, components which are involved in running containers in dedicated and managed environments are hard to grasp but essential for a practical adoption. By providing an analysis of a large number of state-of-the-art software projects we create a taxonomy of building blocks, which constitute the cornerstones of the container ecosystem. The taxonomy is shown to be applicable to systems in the context of container technology, but requires an understanding of their inside functionality.**

I. Introduction

Container technology is rapidly gaining popularity among developers and scientists alike. Docker [1] plays a major role with respect to this gain in popularity, as it attracts users by abstracting from low-level implementations through a unified API. Ever since, a lot of other projects integrating or relying on containers have been established. Frameworks for the management of container clusters, such as Kubernetes [2] by Google, declare the abstraction from physical hosts as their goal and include a set of tools to manage large numbers of containers independently of the underlying infrastructure. Also, operating systems are developed being optimized for the execution of containers on large physical clusters, called Datacenter Operating Systems (DCOS). Similarly, projects targeting networking, packaging, scheduling and many other aspects for containers are created, which lead to the establishment of an entire ecosystem around this technology.

A major issue of this ecosystem, which is mainly driven by open source projects, is the lack of clear functional demarcation of different components. In fact, capabilities and scopes of existing projects overlap. Similarly, we found terminology in literature to be inconsistent. Motivated by this lack of clarity, we contribute to an overview of the currently evolving technological cornerstones in the container ecosystem. We identify and coin features, which typically are provided as a self-contained software component, by the assessment of existing projects and their employed terminology. This results in the following contributions: (1) a taxonomy of *building blocks* for the container ecosystem, (2) an overview of current state-of-the-art software projects, which match identified building blocks, (3) an application of the proposed taxonomy to a set of projects.

The paper is structured as follows: Section II provides and overview of technology fundamentals and outlines current issues related to the adoption of containers, particularly in the Cloud. Section III introduces the taxonomy of building blocks, which are applied to a set of open source projects in section IV. In section V we show the lack of clarity in related work, by comparing used terminology with the proposed taxonomy. At last a conclusion and outlook on possible future research are presented in section VI.

II. Fundamentals and Open Issues

This section provides a short summary of technology fundamentals and current challenges, which inevitably occur during the adoption of containers to run applications in a manageable and productive fashion.

The basic, technological enablers for containers have existed for a few years. Containers are based on a set of kernel features of Unix and Linux systems. The two most critical among these features are (1) logical separation and isolation of process execution and (2) resource limitations for processes. In Linux kernels, the respective capabilities are realized by implementations called *namespaces* and *control groups*, which in turn find adoption by various higher-level implementations, for example LinuXContainers(LXC) [3] and Docker's runC [4] (formerly called *libcontainer*).

Containers as such are a means of Operating System Virtualization, because to processes running inside, the environment appears to be a bare metal system. Similar to the use in [5], the term container in this work is used to refer to one or more processes running in an isolated fashion, where isolation is enforced by OS-kernel features. The term *container* is coined by the aforementioned Linux implementation for process isolation. In this paper, we will stick to the term container, because we feel it to be the most commonly used within the community. If it is necessary to distinguish multiple types of containers, we will also use the term *OS-containers*.

Containers find adoption among practitioners across all phases of the software development process. The DevOps

paradigm can be seen as a conceptual counterpart for that. For developers and especially operators, however, advantages of containers are not obvious and come at a price: the up-front setup of infrastructure, which is required to avoid a completely manual wiring of applications being deployed in containers. Cloud platforms and providers also make use of containers, with PaaS platforms as the pioneers of adopting container technology in their products. Major IaaS providers, such as Google with their Container Engine and Amazons Elastic Container Service, are also offering solutions to run containers. VMs, however, remain their first-class citizens in the datacenter due to unproven tenant isolation and lacking technological maturity of container technology. In general, gaining an overview on concepts and existing projects within the container ecosystem already proves difficult. We contribute to break up this entrance barrier, by providing a holistic, conceptual view on containers and their environment in this work.

III. Container Building Blocks

This section introduces the taxonomy of *building blocks*, which we found to be distinguishable when using containers as the unit for deploying and managing software in a scalable manner and across a cluster of host machines. Each building block is derived from existing projects or systems providing the respective features. We further subdivide building blocks along two classes: those involved in the lifecycle management of a single container (sect. III-A) and those which support the management of groups of containers across a cluster of hosts (sect. III-B). An overview of all building blocks of the taxonomy is shown in Fig. 1. We limit building blocks

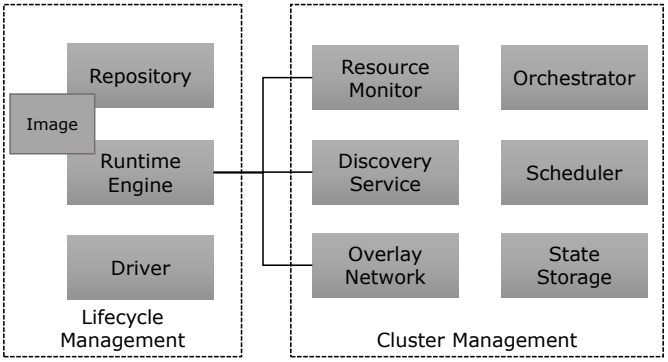

Fig. 1. A Taxonomy of Building Blocks

to functional elements of the container ecosystem. Building blocks may depend on each other. The connecting element between lifecycle management and cluster management is the *container engine*, which is described next.

A. Building Blocks: Lifecycle Management

Elements of lifecycle building block, described here, are grouped under the common criterion of enabling lifecycle management for single containers. We identified four building blocks: **(a) Runtime Engine**, **(b) Driver** and **(c) Repository**.

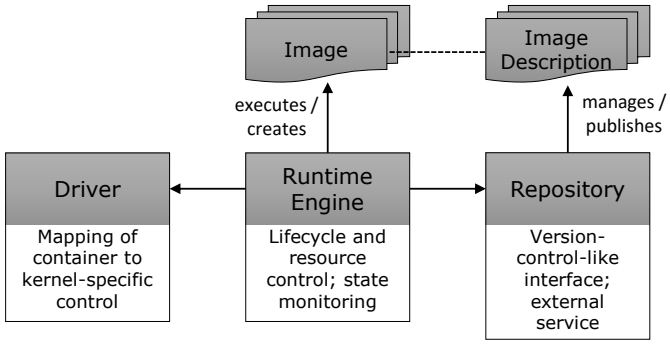

Fig. 2. Building Blocks: Lifecycle

These components rely on additional artifacts, a central one being **(d) Container Images**. An overview of the components including a "uses" relationship indicated by arrows, is shown in Fig. 2.

a) Runtime Engine: A (Container) Runtime Engine provides the interface for lifecycle-management of single containers. Thus, a runtime engine *spawns, starts, stops* and *destroys* a container. Runtime engines also provide *snapshot* and *clone* functionality for a container and control *resource consumption* and container access to host system resources. A runtime engine also manages the mapping of devices across a container's boundaries. As a result, virtual network interfaces and filesystem access are to some extent managed by a runtime engine. In particular, the host-container-interaction wrt. network features is an intersection point with cluster-wide management, where an *Overlay Network* takes control. Resource consumption for containers is also controlled by the engine. Runtime engines rely on *Container Drivers* to map lifecycle controls and resource control to lower-level system calls. Runtime engines use specific types of Container Image Descriptions and Images as input. A runtime engine unpacks and interprets these images and manages their execution.

Docker started out as a runtime engine and at time of writing still is the most popular one, but has developed into a more comprehensive tool. Other container engines for Linux are rkt [6], LXD [7] and Warden [8]. From both a terminology and functional perspective there are a lot of similarities between a container engine, for OS-container virtualization, and a hypervisor, for hypervisor-based virtualization.

b) Driver: A Driver is responsible for the execution of kernel-specific functions which make up the "core" of a container. Staying with the Linux example, this comes down to *namespaces* and *cgroups*. There are various container drivers. The default one of Docker is runC [4], which was recently donated to the Open Containers Initiative[1] but started as their proprietary driver, called *libcontainer*. Others include LXC [3], OpenVZ [9], FreeBSD Jails [10] and Solaris Zones [11].

c) Repository: A repository is a central place for storing, publishing and sharing of Container Image Descriptions and Images. A repository provides access to pre-built container

[1] http://www.opencontainers.org/

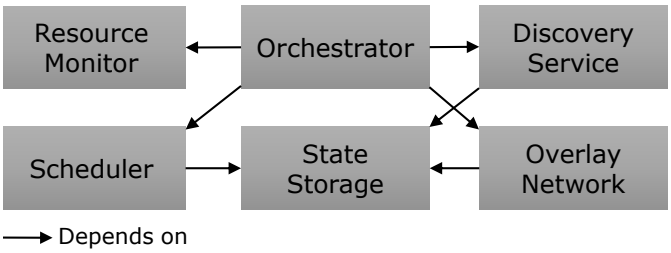

Fig. 3. Building Blocks: Cluster Management

images and offers an interface for the management of image descriptions, which is similar to that of version control systems. This description is derived from the features of Docker's DockerHub [12], as to the authors' knowledge, no other dedicated open source projects for container image repositories exists.

d) Container Image and Image Description: A (Container) Image essentially consists of an archive of a filesystem tree. This includes all installed libraries and user changes on top of the very basic OS kernel. Images are passed to a container runtime engine, which prepares the image (decompressing, possibly downloading requirements) sets up the environment and creates a runnable copy of it - the container. An image can be (re-)created by building it from the corresponding *Image description* which is a document containing a machine-readable description for the creation of an image and provided by a *repository*. Images can also be created based on the snapshots of existing containers.

There are multiple formats for image descriptions, which differ in semantics and scope. The AppC container image description [13] includes runtime dependencies on other containers, as well as environment requirements to run an image. Docker's container image descriptions, the so-called "Dockerfiles", are more minimalistic and focus on a single container, excluding requirements regarding the execution environment. Docker images by default implement a layered file system, AuFS[2] for containers. Combined with the Dockerfile image descriptions which contain references to base-images, an incremental image creation and sharing process is possible.

B. Building Blocks: Cluster Management

The second group of buildings blocks of the container ecosystem are used for the management of containers on distributed hosts. Six building blocks are distinguished: **(a) Orchestrator**, **(b) Discovery Service**, **(c) Overlay Network**, **(d) State Storage**, **(e) Scheduler** and **(f) Resource Monitor**. An overview of the components with a "depends on" relationship, indicated by arrows, is shown in Fig. 3.

a) Orchestrator: Orchestration, as a term from the software architecture domain, comprises concepts on how software artifacts are organized and interact at runtime. For containers, an Orchestrator manages deployment procedures and environmental constraints of containers across multiple servers.

[2]http://aufs.sourceforge.net/

To accomplish the management of different types of containers with interdependencies and specific requirements, an orchestrator relies on the concept of *coherent units*. This concept plays an important role for managing containers across a cluster: A coherent unit logically combines a set of containers, which will be managed as one. Containers which are part of the same coherent unit share resources and a common lifecycle. In addition, a coherent unit has to be assigned to a suitable host, where environment constraints are fulfilled. This placement decision is made by a Scheduler. Communication between coherent units is typically handled by an Overlay Network, as it provides a homogeneous view of networking for both intra- and inter-host communication. Finally, a Discovery Service, with agents typically placed on every host of a cluster, acts as an endpoint for the orchestrator to information on liveness and provides other state-related data of containers. An orchestrator may also use additional monitoring data provided by Resource Monitors, which collect data on a per-host basis.

An example for an orchestrator is Kubernetes [2]. The project is described as a "container cluster management software" and coherent units are referred to as *pods*. Similarly, Docker's Compose [14] allows developers to specify coherent units of containers for single hosts, while Swarm [15] offers a similar functionality for a cluster of hosts. Marathon [16] also provides orchestration functionality, but its distinction from a scheduler is not completely clear.

b) Discovery Service: Application design favors a static access to different layers of services. However, containers, encouraging the use of more fine-grained components and shorter lifecycles, are highly dynamic in nature. To address this mismatch, a Discovery Service exposes static endpoints which are dynamically mapped to running containers. Each host runs a component, called discovery agent, which monitors its local containers' lifecycles and publishes this state. Discovery agents coordinate and store data in a consistent manner using State Storage. A cluster-wide discovery is usually supported by an overlay network, which wraps containers' interconnections in a transparent way, and also relies on State Storage.

Existing orchestrators either include discovery agents, e.g., Kubernetes [2], or can be integrated with an external discovery service. The Mesosphere [17] DCOS comes with Mesos-DNS as an integrated discovery service. Weave [18], an overlay networking project, also includes a discovery service.

c) Overlay Network: Using containers to run distributed applications creates three challenges related to networking:

1) Communication between containers on the same host naturally becomes network communication.
2) A mapping of logical network interfaces to physical network interfaces of the host system is necessary.
3) Containers must be able to communicate across hosts.

These three challenges are approached by various solutions from the domain of software-defined networking. (1) is partially solved by container engines, which manage the virtualized network interfaces of containers on a single host. This means network traffic is IPC, virtualized and masked.

Docker, for example, offers a multitude of options for single-host networking, which sometimes require deep knowledge of the networking stack in Linux. By default, virtual Ethernet interfaces are created for each container, which then can communicate via the virtual network interface *docker0* with other network interfaces through exposed ports. This Docker bridge also supports forwarding of traffic through the external host interface thus enabling conditions (2) and (3). For few, mostly static containers, this is feasible by manually setting up the hosts. However, for larger scale deployments and more practical management of possibly ephemeral containers, *Overlay Networks* are used to address both (2) and (3).

Overlay networks are software solutions, which run agents on each host, through which the mapping of network ports and traffic is handled. Flannel [19] is such an overlay network. Flannel can be attached to Docker networking and allows the abstraction from physical hosts by maintaining the network configuration for the entire cluster. Network configuration is stored using State Storage. Other network solutions for containers include Calico [20], Weave [18], Pipework [21] and Docker's libnetwork [22].

d) Scheduler: Schedulers in the context of containers assume a global point of view and are responsible for the distribution of containers or groups thereof across available hosts. A scheduler allocates resources to tasks according to a set of rules. Container schedulers, due to a possibly much higher amount of containers than hosts, should scale well with both cluster size and number of containers. Other than that, they ensure that resource capacity constraints are not violated and handle load distribution over a cluster of machines. Container schedulers consequently manage resource allocation and placement of containers, but also handle operations affecting scale and depend on up-to-date information of cluster state, relying on State Storage. Schedulers are also the interface for deployment: a scheduler receives the specification for a container or a coherent unit of multiple containers and manages lifecycle and placement over multiple hosts.

Kubernetes [2] includes a scheduler whose architecture resembles Google's Omega [23], which in turn inspired a few other schedulers for containers. Other projects which provide scheduling capabilities include Marathon [16], fleet [24] and Swarm [15]. Schedulers are often pluggable components of orchestrators.

e) State Storage: Configuration of containers is required in a similar manner to VMs. Containers, however, can be deployed with a higher density and are more dynamic in nature, for example batch jobs may be run in ephemeral containers. Cluster-wide features, such as the already discussed scheduling and discovery, require a consistent and up-to-date view of cluster state. This state is stored in a database service, which consequently becomes an integral part of the container ecosystem and we call State Storage.

Due to the potentially high number of containers which are spread across multiple hosts, the database should be fast, natively distributed and scale well with the number of requests. As the data model for state information can (and should) be kept rather simple, key-value stores dominate the currently existing solutions. Current systems also share a subscription/listening capability, which allows clients to be automatically informed about updates. All popular projects rely on consensus algorithms to ensure some form of consistency. Apache Zookeeper [25] is a well-known system fulfilling these requirements, implementing the ZAB protocol [26] for consensus. Others are etcd [27] and Consul [28], which both use the Raft [29] consensus protocol.

f) Resource Monitor: Managing a larger cluster of containers and hosts requires information about resource utilization and health on multiple levels. For users of a platform, a detailed status of their own containers is desirable, while providers will be interested in resource utilization and status of a whole cluster of containers or machines. In a VM it is possible to use OS means to measure resource consumption, but for a container this is not possible. The reason is that each container sees all host resources. As a result, measures like CPU utilization are invalid when measured from inside a container, as they reflect overall state of the host system. One option is to run a monitoring system, which hooks into the container driver's local resource allocations. Docker's *stats* command does this for containers running with Docker and can provide data either in textual or JSON format. cAdvisor [30] is a solution with pluggable driver support and also requires root access on the host system. A cluster-wide monitoring can be built on top of these solutions. Heapster [31] is such a project, which is closely coupled to Kubernetes. Getting application-specific metrics from containers into a dashboard or storage back-end must be done using software not specific to containers. We found the container ecosystem to be generally lacking well-integrated monitoring solutions. Host resource utilization for CPU and RAM are the most common metrics.

IV. Applying the Taxonomy: A Snapshot of the Container Ecosystem

Using the proposed taxonomy, we map the described building blocks onto the container ecosystem, which is represented by a set of software projects and tools. The ecosystem, at the time of writing, is evolving rapidly in the context of open source projects. As a result, we can only provide a snapshot and have no claim of providing a comprehensive list of existing projects. We selected one project each, which is clearly positioned within the respective block. Other than that, we selected projects which do not match distinct blocks. The application of the taxonomy is shown in Tab. I, where an *x* indicates a complete integration of the respective feature, while *(x)* signals a partial integration. Partial integration means functionality to occupy the respective building block is implemented, but either is very simple and limited or not clearly distinguishable from other features central to the projects goal. We found it possible to identify at least one project which matches exactly the boundaries of a building block - the exception being container discovery. Our experience showed that the mapping to our taxonomy was difficult. The main reason for that was

	Runtime Engine	Driver	Repository	Orchestrator	Scheduler	State Storage	Discovery	Resource Monitor	Overlay Network
rkt [6]	x								
runC [4]		x							
DockerHub [12]			x						
Marathon [16]				x	(x)				
Chronos [32]					x				
etcd [27]						x			
cAdvisor [30]								x	
flannel [19]									x
Docker [1]	x								(x)
Kubernetes [2]				x	(x)		(x)	(x)	
Docker Swarm [15]	(x)			x	(x)		(x)		
Weave [18]							x		x

TABLE I
APPLICATION OF THE TAXONOMY TO EXISTING SOFTWARE PROJECTS.

unclear terminology. Discrepancies between documentation, claims and actually provided features of software projects added to the required effort. We shortly want to discuss some projects, which we categorized as having partial integration of building blocks.

Marathon, according to our taxonomy, is an orchestrator as it allows the cluster-wide management of services running in containers and includes a model for grouping containers. However, it also allows users to directly influence the placement of specific containers, without delegating this to a distinct scheduling component or providing other functionality expected of a scheduler, for example automated assignment of ephemeral tasks to hosts. A similar case can be made for Kubernetes, which on top of scheduling does the same for discovery and monitoring: the orchestration functionality incorporates some functionality of these blocks, for example simplified monitoring and a DNS-like discovery.

V. Related Work

So far, there is no explicit effort in literature[3] to distinguish and name different elements within the ecosystem of container technology. In conclusion, related work is taken from two fields, which employ related terminology in their research.

The first field is focused on the technological foundations of containers. This currently amounts to *systems research*, where containers have been looked at particularly in the context of high-performance computing, and the *application of container technology in the context of PaaS*. Within the systems research domain Matthews et al. in [34] were one of the first to consider OS-containers an alternative to hypervisor-based virtualization. They compared containers to VMs along the dimension of performance isolation. Soltesz et al. [35] were the first to emphasize advantages of containers regarding performance overhead. Both of these works do not differentiate components used to create an environment for containers and subsume everything under "VM", including containers. Felter et al. [36], also comparing performance overhead between containers and VMs, provide an overview of Linux container drivers' functionality. They lack a differentiation between container driver and container engine, calling both "tools to manage containers" [36, p. 4].

Still within the first field of related work, we further identified two contributions focusing on the application of container technology in the context of PaaS [37], [38]. These approaches analyze technical properties of different container engines and drivers and their impact on relevant PaaS-aspects, such as tenant isolation. We found these works to be lacking precise and distinct terminology for containers and their environment, which is a consequence of the aforementioned viewpoint. Dua et al. simply refer to both drivers and engines as "container implementation" [37, p. 610]. Pahl refers to the container driver as "engine" [38, p. 25], while tools like Docker receive no label. He also mentions a container ecosystem, which according to his understanding only includes a container driver (again called "engine") and a repository. Pahl also looks at cluster management for containers across hosts, but does not label or categorize elements clearly.

The second field of related work is centered on design and architecture of systems within the context of cloud applications using container technology. Two contributions exist looking at orchestration [39] and microservice-architectures [40] for containers. While Tosatto et al. distinguish container driver and engine (specifically for Docker), they call the latter an "orchestrator" [39, p. 73], but later, contradictorily, identify fleet and Docker Compose as orchestration solutions on top of Docker. Toffetti et al. [40] focus on the management of clusters of containers, emphasizing discovery and distributed state storage. They implement their own framework for enabling a "self-managing atomic service" and name orchestration and discovery as features. While they are not inconsistent wrt. terminology, they do not distinguish clearly between discovery and state storage, as well as orchestration and scheduling.

VI. Conclusion and Future Work

Containers are the driving technology for the DevOps paradigm and considered a technological counterpart for microservices. Containers have been compared to VMs, but pose new challenges regarding operation and management. Looking at the lack of conceptual clarity in existing literature and state-of-the-art software projects related to container technology, we contribute to the identification and distinction of relevant cornerstones within this newly emerging ecosystem. This

[3]There is, however, a very extensive tutorial series on the container ecosystem with a more practical, technology-centric focus, which employs similar terminology for cluster management features by DigitalOcean [33]

results in the set of building blocks as described in sect. III. We subdivide building blocks into two groups: the first group is involved in the lifecycle management of containers on a per-host basis, while the second group contributes to the management of clusters of containers across multiple hosts. Each group has one focal point, which is the container engine for lifecycle management and a container orchestrator for cluster management. By applying the taxonomy to existing projects, we show that a disambiguation is feasible. However, this insight required a look at either documentation or implementation details of the respective software projects, proving the lack of commonly used terminology.

For future work, two research directions are very promising. The first direction is container-based software application architectures, in particular microservice-based architectures. While a lot of best practice approaches related to the development process of software along the concept of microservices exist, architectural viewpoints on the design of applications and storage are lacking. The efficient usage of microservices depends highly on the environment and consequently is subject to limits of the container ecosystem. The second research direction is the experimental evaluation of container-based systems and architectures wrt. different QoS aspects, in particular scalability and elasticity.

REFERENCES

[1] Docker. [Online]. Available: https://www.docker.com/ [Accessed: 01/31/2016]

[2] Kubernetes cluster management. [Online]. Available: http://kubernetes.io/ [Accessed: 01/31/2016]

[3] LXC: Linux Containers - What's LXC? [Online]. Available: https://linuxcontainers.org/lxc/introduction/ [Accessed: 01/31/2016]

[4] runc. [Online]. Available: https://github.com/opencontainers/runc [Accessed: 01/31/2016]

[5] V. Marmol, R. Jnagal, and T. Hockin, "Networking in Containers and Container Clusters," 2015.

[6] rkt app container runtime for linux. [Online]. Available: https://github.com/coreos/rkt [Accessed: 01/31/2016]

[7] LXD - "container hypervisor". [Online]. Available: https://linuxcontainers.org/lxd/introduction/ [Accessed: 01/31/2016]

[8] Warden. [Online]. Available: https://docs.cloudfoundry.org/concepts/architecture/warden.html [Accessed: 01/31/2016]

[9] OpenVZ. [Online]. Available: http://openvz.org/Main_Page [Accessed: 0/30/2015]

[10] FreeBSD Jails. [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=jail&format=html [Accessed: 01/31/2016]

[11] Solaris Zones. [Online]. Available: http://docs.oracle.com/cd/E26502_01/html/E29024/preface-1.html [Accessed: 01/31/2016]

[12] Docker Hub. [Online]. Available: https://hub.docker.com/ [Accessed: 01/31/2016]

[13] appc app container specification. [Online]. Available: https://github.com/appc/spec [Accessed: 01/31/2016]

[14] Docker Compose. [Online]. Available: https://docs.docker.com/compose/ [Accessed: 01/31/2016]

[15] Docker Swarm. [Online]. Available: https://docs.docker.com/swarm/ [Accessed: 01/31/2016]

[16] Marathon. [Online]. Available: https://mesosphere.github.io/marathon/ [Accessed: 01/31/2016]

[17] Mesosphere. [Online]. Available: http://docs.mesosphere.com/ [Accessed: 01/31/2016]

[18] Weave. [Online]. Available: http://weave.works/ [Accessed: 01/31/2016]

[19] flannel. [Online]. Available: https://github.com/coreos/flannel [Accessed: 01/31/2016]

[20] Project Calico. [Online]. Available: http://www.projectcalico.org/ [Accessed: 01/31/2016]

[21] Pipework. [Online]. Available: https://github.com/jpetazzo/pipework [Accessed: 01/31/2016]

[22] libnetwork. [Online]. Available: https://github.com/docker/libnetwork [Accessed: 01/31/2016]

[23] M. Schwarzkopf and A. Konwinski, "Omega: flexible, scalable schedulers for large compute clusters," *EuroSys '13 Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 351–364, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2465386

[24] fleet. [Online]. Available: https://github.com/coreos/fleet [Accessed: 01/31/2016]

[25] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems." *USENIX Annual Technical ...*, vol. 8, pp. 11–11, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855851$\delimiter"026E30F$nhttps://www.usenix.org/event/usenix10/tech/full{_}papers/Hunt.pdf

[26] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 245–256, 2011.

[27] etcd. [Online]. Available: https://coreos.com/etcd/ [Accessed: 01/31/2016]

[28] Consul. [Online]. Available: https://www.consul.io/ [Accessed: 01/31/2016]

[29] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014, pp. 305–320. [Online]. Available: http://74.207.243.117/var/blurbs/pubs/raft-tr14.pdfhttp://dl.acm.org/citation.cfm?id=2643634.2643666

[30] cAdvisor. [Online]. Available: https://github.com/google/cadvisor [Accessed: 01/31/2016]

[31] Heapster. [Online]. Available: https://github.com/kubernetes/heapster [Accessed: 01/31/2016]

[32] Chronos. [Online]. Available: https://mesos.github.io/chronos/ [Accessed: 01/31/2016]

[33] J. Ellingwood. Tutorial Series: The Docker Ecosystem.

[34] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the Performance Isolation Properties of Virtualization Systems," 2007, p. 6. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281706

[35] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization," *ACM SIGOPS Operating Systems Review*, vol. 41, p. 275, 2007.

[36] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," *Technology*, vol. 25482, 2014.

[37] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, mar 2014, pp. 610–614. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6903537

[38] C. Pahl, "Containerization and the PaaS Cloud," 2015.

[39] A. Tosatto, P. Ruiu, and A. Attanasio, "Container-Based Orchestration in Cloud: State of the Art and Challenges," *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 70–75, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7185168

[40] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud - AIMC '15*. New York, New York, USA: ACM Press, 2015, pp. 19–24. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2747470.2747474