Performance overhead of container orchestration frameworks for management of multi-tenant database deployments

Eddy Truyen, Dimitri Van Landuyt, Bert Lagaisse, Wouter Joosen
imec-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

Abstract

The most common and preferred approach in the literature on service-level objectives (SLOs) for multi-tenant databases is to group tenants according to their SLA class (e.g. golden and bronze SLAs) in separate database processes and find optimal co-placement of golden and bronze tenants across VMs. In order to implement admission control of aggressive tenants, request throttling is preferred over running database processes in separate virtual machines that introduce a significant performance overhead. A relevant question is whether more light-weight container technology such as Docker is a viable alternative for running high-end performance database workloads. Moreover, the recent uprise and industry adoption of container orchestration (CO) frameworks for the purpose of automated placement of cloud-based applications raises the question what is the additional performance overhead of CO frameworks in this context. In this paper, we evaluate the performance overhead introduced by Docker engine and two representative container orchestration frameworks, Docker Swarm and Kubernetes, when running and managing a CPU-bound Cassandra workload in OpenStack. Firstly, we have found that Docker engine deployments that run in host mode exhibit negligible performance overhead in comparison to native deployments. Secondly, we have found that service networking approaches of both Docker Swarm and Kubernetes introduce a substantial overhead due to virtual network bridges when compared to the Docker engine deployments. This demands for service networking approaches that run in true host mode but offer support for network isolation between tenants. Thirdly, storage plugins have a large impact on the overall resource model of the database tier. More specifically, we show that the CPU-bound Cassandra workload changes into an I/O-bound workload when Cassandra is deployed using Kubernetes and Docker Swarm because the persistent volume operations of their local volume plugins are a performance bottleneck that does not appear in Docker engine deployments. This implies that solved placement decisions for native or Docker engine deployments cannot be reused for container orchestration frameworks.

ACM Reference format:
Eddy Truyen, Dimitri Van Landuyt, Bert Lagaisse, Wouter Joosen. 2019. Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. In *Proceedings of The 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, April 8–12, 2019 (SAC '19),* 6 pages.
DOI: 10.1145/3297280.3297536

DOI: 10.1145/3297280.3297536

1 Introduction

Multi-tenancy is an architectural design principle for Database-as-a-Service (SaaS) providers to enable the hosting of tenants by a single database instance in order to reduce development and operational costs for the DaaS provider [9]. In particular, data of tenants is stored in the same database process or even database entity (e.g., table, document, collection).

Tenants and DaaS provider operate according to a service level agreement (SLA), which defines among others a contract with specific service level objectives (SLOs) about performance and availability. A performance SLO is typically expressed as a contract with mutual rights and obligations: if the tenant keeps below a *maximum allowed request rate*, the SaaS provider is able to guarantee a *minimum response latency and/or throughput* expressed in terms of percentiles between the range of 95-99th. To enforce the SLO, admission control of aggressive tenants is required. Moreover to offer different custom SLOs to tenants as part of different SLA classes (e.g. bronzen and golden SLA), QoS differentiation between golden and bronze tenants must be implemented.

The most common and preferred approach in the literature for SLO management of multi-tenant databases consists of a two-fold approach: (i) group golden and bronze tenants in separate database processes [16, 21] and (ii) find optimal placements of golden and bronze database processes across different (virtual) machines so that SLOs are met and VM resources are maximally utilized. Note to implement performance isolation between tenants that run on the same VM, a request scheduler approach [12] is preferred over running database processes as separately nested VMs that are not performing well enough for high-end performance SLOs such as 1000 transactions per second [16, 22].

Recently, there has been a strong industry adoption of Docker containers due to their lower memory footprint and more adaptive resource allocation among different co-located containers, leading to improved resource utilization in comparison to virtual machines [28]. This raises a first relevant question whether container technology such as Docker is a viable alternative for running high-end performance database workloads.

Container orchestration (CO) frameworks, such as Kubernetes [14] and Docker Swarm [4] that provide support for automated container placement, scaling and management have also arisen [10]. These frameworks include by default three kinds of automated management functionalities: (i) cost-efficient scheduling of different types of workloads, (ii) inter-container networking, service load balancing and discovery and (iii) QoS assurance and adaptive resource allocation. This raises a second relevant question about the performance overhead of using CO frameworks for automated management of database clusters.

The remainder of this paper is structured as follows. First, Section 2 presents related work. Then, Section 3 evaluates the performance overhead of Docker engine and two representative CO frameworks, Docker Swarm and Kubernetes. Finally, Section 3 discusses the findings of the performance evaluation. The experimental setup scripts and research data of this work are available on GitHub [5].

2 Related work

Existing research [6, 7, 20] has focused most attention on comparing the performance of a single container against a single virtual machine, both running directly in Linux on top of a bare-metal machine. Sharma et al. [20] also evaluate the so-called hybrid model where a containerized Redis database, inside a VM, is compared to Redis natively installed in the OS of the VM. Their results show that the hybrid model performs slightly better. As will be shown below, we observe similar findings when comparing a Docker+VM deployment of Cassandra in comparison to a VM-only deployment.

Containers do not provide the same level of performance isolation as virtual machines however [6, 17, 25]. Therefore, it is still preferred to implement admission control by means of a request scheduler that is placed before or within the load-balancing tier of the DaaS provider [12]. QoS differentiation between golden and bronze tenants can then be realized by allocating more resources to the database containers of the golden tenants [24].

Literature on design and evaluation of CO frameworks originates mostly from Google [25] and Kubernetes is also originally created by Google [1]. The Borg system, a predecessor of Kubernetes, is currently the main platform for orchestrating and managing various Google services. For example, the Google Cloud Engine IaaS relies on Borg for scheduling VMs inside containers. Verma et al. [25] shows that the Borg system supports improved resource utilization in terms of number of machines needed for fitting a certain workload on.

Kratzke et al. [11] has already studied the overhead of virtual networks between containers in public cloud providers and concludes that operating container clusters with highly similar core machine types is the best strategy in public cloud provider platforms to minimize the data-transfer rate-reducing effects of containers. As such, we have taken the findings of this work as premise for our experiments.

Gehberger et al [8] evaluates the performance of a specific network plugin for Kubernetes that exploits hardware-supported features such as SRV-IO, in the context of low-latency robot-to-robot communication. Their results show that there is still an overhead of more than 30% in comparison to native deployments that don't use Kubernetes.

Truyen et al. [23] presents an extensive evaluation of Docker Swarm and Kubernetes for deploying and managing a MongoDB database cluster. Similar to our work, they use the YCSB benchmark for various workload types. Their results show similar response latency overheads as we have measured for Cassandra. Our work differs in that we have also analyzed resource usage metrics, hereby revealing that storage plugins can have a negative impact on the overall resource model of CPU-intensive database workloads. As such we identify the causes of the observed performance overhead.

3 Performance overhead of CO frameworks

In this section we will measure what is the performance overhead of using CO frameworks for running a CPU-bound Cassandra workload on top of an OpenStack private cloud in a closed research lab. We compare the performance overhead with a native deployment and a Docker engine deployment of Cassandra.

3.1 Selected database workload

We have selected a Cassandra write-only workload that is known to be highly performing because Cassandra uses an in-memory write-back cache technology, called Memtables [2]. A Cassandra cluster comprises a masterless ring of nodes to replicate data. To process a write request, Cassandra first determines which nodes should process the request and sends messages to those nodes. Those nodes then store the data of the write request in a Memtable and eventually flush this data to persistent storage. Such flushes occur periodically or when the Memtable's memory limit has been reached. Before a Memtable is flushed, a new one is created. Before storing the data in a Memtable, Cassandra will first persist the write request into a highly performing commit log on disk that is continuously appended so that no additional disk seek time is incurred. This design choice makes that write-heavy Cassandra workloads are CPU-bound unless the commit log has performance issues. As such, Cassandra is well fit for measuring the impact of container technology when high-end database performance is a requirement.

3.2 Overview of compared deployments

We compare 4 deployments of a Cassandra cluster with 3 nodes: (i) a native deployment where Cassandra v2.0.17 is directly installed in the Ubuntu OS, (ii) a Docker engine deployment where 3 database containers are started from the official cassandra:2.0 Docker image, (iii) a Docker Swarm deployment where the three database containers are exposed by means of a stable Service IP address, (iv) a Kubernetes deployment where the database containers are exposed via a stable DNS name (this is the recommended endpoint configuration for databases in Kubernetes).

We have excluded Mesos-based CO frameworks from the experiments because at the time of experiments the container networking solution of Mesos has not yet been used in any Mesos-based CO framework. We have installed Kubernetes v1.7.2 using the kubeadm v1.7.2 tool [15]. We installed Docker Swarm integrated mode as part of Docker engine 17.04.0 ce-0 ubuntu-xenial.

All database instances store their state in the local file system of the VM. Swarm and Kubernetes deployments respectively use a local Docker volume and a hostPath volume, while the Docker deployment also directly writes to a path in the host's file system. Moreover, the Cassandra cluster is configured with a replication level of 2 nodes and a write consistency level of `ONE`, i.e., a write operation will be acknowledged after a write request has been written to the commit log and memtable of at least one replica node.

Docker Swarm and Kubernetes deployments are configured with the Weave NET network plugin [26] for setting up a virtual network between containers. Weave NET installs a virtual bridge to isolate network traffic between containers. The DockerOnly deployment runs in host mode which means that it has direct access to the networking stack of the host OS. The advantage is a higher networking performance [7], but there is no security isolation of network traffic.

3.3 Benchmark and testbed

We have used as benchmark the well-known YCSB benchmark [3] for evaluating the performance of databases. More specifically, we have run the load phase of the YCSB workload A for measuring the performance of the above four deployments. To measure the performance, a scalability test is performed by means of 15 runs of 10^5 write requests where the number of client threads in increased with 5 additional threads per consecutive run. We also measure resource utilization at the database nodes by means of the dstat tool [27].

The testbed for running all the experiments of this paper is an isolated part of a private OpenStack cloud, version Mikata. The OpenStack cloud consists of a master-slave architecture with two controller machines and 5 droplets, on which VMs can be scheduled. The droplets have Intel(R) Xeon(R) CPU `E5-2650 2.00GHz` processors and 64GB DIMM DDR3 memory with Ubuntu xenial, while the master controller is an Intel(R) Xeon(R) CPU `E5-2430 2.20GHz` machine with Ubuntu xenial. Each droplet has two 10Gbit network interfaces. The three database instances of each deployment are installed in 3 VMs with 2 vCPUs and 4 GB of RAM and an Ubuntu xenial 4.4.0-112-generic OS. The two CPU cores are exclusively reserved for the VM. Swapping is turned off. Each of these VMs runs on a separate droplet. The YCSB VM is also deployed on a separate droplet with a c4m8-flavored VM.

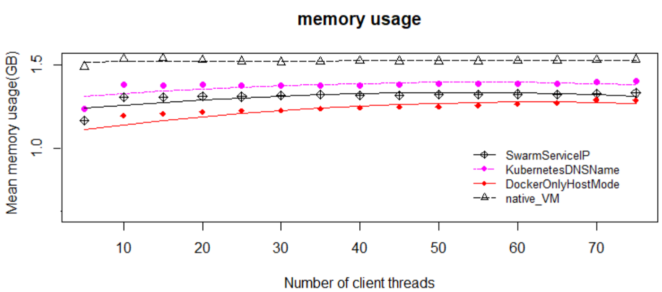

Figure 1. Memory usage

3.4 Results

First of all, we inspect memory usage (see Figure 1). This shows that the official Docker image of Cassandra 2.0 consumes less memory than the natively installed deployment. Presumably, this is because the particular Docker image has been configured for memory-efficiency. Moreover, Kubernetes consumes more memory than Docker Swarm because Kubernetes is a larger system with more features than Docker Swarm.

The performance of the deployments is evaluated by how quickly they can reach 100% CPU utilization and the performance overhead on response latency. The DockerOnly deployment reaches full CPU utilization quite faster than the container orchestrated deployments (see right graph of Figure 2(a)). Correspondingly, Docker's performance with respect to response latency is also much better (see left graph of Figure 2(a)). We also observe that for the container orchestrated deployments a substantial percentage of CPU time is spent at soft interrupts (see left graph of Figure 2(b)), which in turn can be explained by the usage of a virtual network between

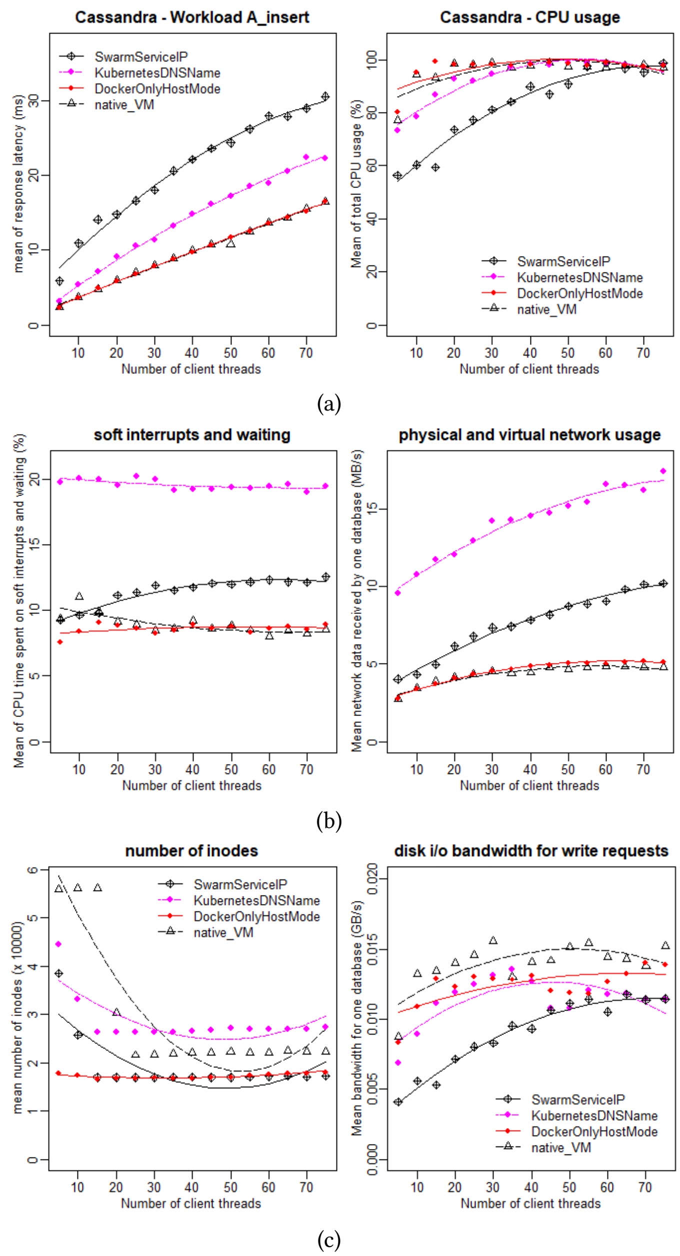

Figure 2. Performance overhead of Kubernetes and Docker Swarm in comparison to VM+Docker and VM-based deployments

containers. Indeed, total network usage shows an increased network usage for Docker Swarm and Kubernetes (see right graph of Figure 2(b)). It is also shown that although Kubernetes consumes more CPU time at soft interrupts than Docker Swarm, it still has a better overall performance in terms of response latency. This better performance can be explained by inspecting the disk I/O rate at which Cassandra is able to persist write requests to its commit log (see right graph of Figure 2(c)). The disk I/O rate of the Docker Swarm deployment (that uses the `local` volume driver) is in comparison with the DockerOnly deployment much lower, while the disk I/O rate of the Kubernetes deployment (that directly writes to a directory on the host's file system using a hostPath volume) is a bit lower.

In summary, the VM+Docker deployment of Cassandra that runs in host mode behaves in par with the VM-only deployment. However, volume and network plugins of container orchestration frameworks have an effect on the resource usage profile of the Cassandra workload: (i) Due to the use of volume plugins, a CPU-bound workload in the DockerOnly and nativeVM deployments becomes I/O-bound in Docker Swarm and Kubernetes; (ii) network plugins, which install a virtual bridge for isolating network traffic between containers, require a substantial part of the CPU time.

The performance overhead of using container orchestration frameworks for the database tier seems thus substantial. Moreover, the impact on the overall resource model of an application with respect to what resources are the primary and secundary performance bottlenecks implies that optimal placement decisions for multi-tenant databases using Docker Engine cannot be reused when using CO frameworks.

4 Conclusion

We have evaluated in an OpenStack private cloud the performance overhead of container orchestration frameworks for running and managing database clusters in comparison to VM-based deployments and VM+Docker deployments. The performance overhead of VM+Docker deployments with host mode networking is very low in comparison to VM-based deployments. However, service networking solutions of CO frameworks introduce a substantial overhead because of additional CPU consumption for virtual network bridges. Moreover storage plugins change a CPU-bound database workload into an I/O-bound. This is undesirable because solved placement decisions for a native or Docker engine deployment need to be recomputed when using a container orchestration framework.

We conclude that CO frameworks entail various benefits for automating SLO-aware placement of multi-tenant applications, but in order to truly reap these benefits for multi-tenant database deployments, CO frameworks should further develop the isolation of container networking approaches that rely on host mode networking and improve the performance of volume plugins for local persistent storage.

Acknowledgments

This research was funded by the Agency for Innovation and Entrepreneurship IWT, grant DeCoMAdS, grant number 179K2315, and the Research Fund KU Leuven We thank Vincent Reniers for developing a script generator for running multiple YCSB experiments and development of R-scripts for statistical analysis of YCSB results.

References

[1] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[2] Apache Cassandra. 2018. Understanding the architecture. URL: http://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archTOC.html, accessed 2018-01-29. (2018).

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (2010), 143–154. DOI: https://doi.org/10.1145/1807128.1807152

[4] Docker. 2018. Swarm mode overview. https://docs.docker.com/engine/swarm/. (2018). https://docs.docker.com/engine/swarm/ Accessed: February 14 2018.

[5] Eddy Truyen et al. 2018. Performance overhead of container orchestration frameworks for multi-tenant database deployments. (2018). https://goo.gl/PWwR9f Accessed: December 12 2018.

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

[7] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2015). http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7095802

[8] D. Gehberger, D. Balla, M. Maliosz, and C. Simon. 2018. Performance Evaluation of Low Latency Communication Alternatives in a Containerized Cloud Environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE.

[9] Dean Jacobs and Stefan Aulbach. 2007. Ruminations on Multi-Tenant Databases. *BTW Proceedings* 103 (2007).

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

[11] Nane Kratzke. 2015. About Microservices, Containers and their Underestimated Impact on Network Performance. In *Proceedings of CLOUD COMPUTING 2015 (6th. International Conference on Cloud Computing, GRIDS and Virtualization)*. 165–169.

[12] Rouven Krebs, Christof Momm, and Samuel Kounev. 2014. Metrics and techniques for quantifying performance isolation in cloud environments. *Science of Computer Programming* 90 (2014), 116–134. DOI: https://doi.org/10.1016/j.scico.2013.08.003

[13] Kubernetes. 2018. Evicting end-user pods. URL: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/tasks/administer-cluster/out-of-resource.md#evicting-end-user-pods, accessed 2018-09-18. (2018).

[14] Kubernetes. 2018. Production-Grade Container Orchestration. URL: https://kubernetes.io/, accessed 2018-01-23. (2018).

[15] Kubernetes. 2018. Using kubeadm to Create a Cluster. URL: https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/, accessed 2018-01-29. (2018).

[16] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. 2014. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1447–1463.

[17] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM.

[18] Linux. 2008. CFS Scheduler. URL: https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-design-CFS.txt, accessed 2018-05-18. (2008).

[19] Mesosphere. 2018. Documentation from Mesosphere DC/OS 1.10. https://docs.mesosphere.com/1.10/. (2018). https://docs.mesosphere.com/1.10/ Accessed: February 14 2018.

[20] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y C Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA.

[21] David Shue, Michael J Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. *10th USENIX Symposium on Operating Systems Design and Implementation* (2012), 1–14.

[22] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J Elmore, Michael Stonebraker, and David Dewitt. 2016. STeP : Scalable Tenant Placement for Managing Database-as-a-Service Deployments. *Proceedings of the Seventh AMC Symposium on Cloud Computing* (2016), 388–400.

[23] Eddy Truyen, Matt Bruzek, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. 2018. Evaluation of container orchestration systems for deploying and managing NoSQL database clusters. In *Cloud Computing (CLOUD), 2018 IEEE 11th International Conference on*. IEEE.

[24] Eddy Truyen, Dimitri Van Landuyt, Vincent Reniers, Ansar Rafique, Bert Lagaisse, and Wouter Joosen. 2016. Towards a container-based architecture for multi-tenant SaaS applications. In *ARM 2016 Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*. ACM. DOI: https://doi.org/10.1145/1235

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

[26] weaveworks. 2018. Weave Net. https://www.weave.works/oss/net/. (2018). https://www.weave.works/oss/net/ Accessed: February 14 2018.

[27] Dag Wiers. 2009. Dstat: Versatile resource statistics tool. URL: http://dag.wieers/home-made/dstat/, accessed 2018-01-29. (2009).

[28] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. 2014. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2014), 299–306. DOI: https://doi.org/10.1109/PDP.2014.78

Appendix: Container orchestration frameworks

This section provides an overview of common features of popular container orchestration frameworks such as Docker Swarm [4], Kubernetes [14] and the Mesos-based DC/OS [19] that are relevant for adaptive performance isolation of multi-tenant applications. We group the discussion of related features in functional aspects that each correspond with a specific architectural component or logical functional substrate of CO frameworks. Functional aspects are indicated by a **bold font** while features are indicated by an *italic font*.

4.1 Scheduling of workloads

All CO framework offer rich concepts for **declarative application configuration**. First of all, *several workload types are supported* such as jobs; elastically scalable, stateless services; and stateful applications such as databases.

In Kubernetes, the smallest unit of deployment is a *Pod*, which is a set of co-located containers that logically belong together and therefore are always deployed together on the same node. This concept has also been adopted by Mesos, Marathon and DC/OS. Docker Swarm names this concept Task but Tasks can only encapsulate one container. In the remainder of this paper we refer to the unit of container deployment as Pod.

All CO frameworks supports the configuration and deployment of *container-based services* as a load-balanced pool of replicated Pods. The Pod replicas of a service can be *elastically scaled out or scaled in* in all CO frameworks. A *built-in Layer 4 load balancer* that is installed at every node of the cluster will automatically be informed of additions or removals of Pod replicas.

Containers are by default stateless; when a container dies or is stopped, any internal state is lost. Therefore, so-called *persistent volumes* can be attached to store the persistent state of the container on disk. Different type of volume drivers can be distinguished: local volumes with support for automated rescheduling of Pods, shareable volumes between Pods, and external persistent volumes.

An important element of every CO framework is a **simple yet highly customizable scheduling algorithm** that is responsible for computing on which nodes Pods should be placed. Various types of user-specified constraints give users control over the performance of the application: first, CO frameworks allow to restrict the set of nodes on which a specific Pod can be scheduled by means of *evaluating over predefined or custom node labels or attributes*. The CO frameworks also support *highly expressive constraints* such as affinity and anti-affinity constraints between Pods and nodes on the one hand, and affinity and anti-affinity constraints between Pods on the other hand.

4.2 Service discovery and load balancing

All CO framework support one or multiple approaches to **services networking**, i.e support for exposing the service of a container via a network. There are two different approaches to services networking that are fully supported by Docker Swarm, Kubernetes and DC/OS: a *routing mesh for global node ports* where every service is identified by means of a unique port that is opened at each node of the cluster where the container runs, and a *virtual IP network for containers* where each service is identified by means of a stable cluster IP address. Requests to a node port or cluster IP of the service are forwarded by a distributed L4 load balancer.

Kubernetes also allows to *bypass the built-in L4 load balancer* by exposing the service in an *internal DNS service* as a DNS record that refers directly to the cluster IP addresses of the containers. This feature is useful for deploying stateful applications such as a database cluster where the different database instance interact in a peer-to-peer fashion and therefore each require a stable DNS address. Kubernetes names this concept a Headless service, i.e., a service that is not exposed via a cluster IP. Docker Swarm v17.06 and DC/OS v1.11 support a different approach where a service's DNS record refers to the IP addresses of the nodes on top of which the containers of that service run. Swarm and DC/OS thus requires that the containers can be accessed via the networking stack of the node's native operating system.

In order to support *access to services from external clients outside the cluster*, an external load balancer solution must be used. In CO frameworks with a routing mesh, the built-in L4 load balancer can play the role of such load balancer if the public or private IP addresses of one or more nodes of the cluster are reachable for external clients. All CO frameworks also support an L7 HTTPS load balancer that must be separately configured for each service. Kubernetes additionally supports automated integration with load balancing services of private and public cloud providers.

4.3 QoS assurance

Resource quota management features enables a cluster administrator to organize the hardware resources of a cluster among different teams or organizations. First, all container orchestration frameworks offer concepts for partitioning one or more types of API objects (e.g. services, volumes) into a logically named *user group* that corresponds with a specific organization or tenant that is able to contract resources from the cluster. The typical use case of a user group is to reserve a subset of resources for a tenant of the cluster. Secondly, Kubernetes and Mesos provide support for *declaring a minimum guarantee and/or maximum quota on CPU, memory and disk resources per user group.*

Container QoS control features enable an application manger to efficiently use the resources of a user group while also achieving the intended QoS level of applications in that user group. Supporting high utilization of allocated resources while also maintaining desired QoS levels of applications, during either normal execution or resource contention and failures, is a complex goal. To support this complex goal, CO frameworks offer the following two features: 1. CO frameworks offer *resource allocation models* that support QoS differentiation between containers while also allowing for over-subscription of CPU, memory and local disk resources. More, specifically Kubernetes and Docker Swarm supports minimum guarantees and maximum limits for various types of resources, while Mesos-based frameworks only supports maximum limits. For example, Kubernetes manages a `<request, limit>` pair for CPU, memory and disk space for each container and Pod. Very similarly, Docker Swarm integrated supports a `<reservation, limit>` pair. A `request` or `reservation` defines an absolute resource quantity that is always guaranteed to the container, while a `limit` specifies the maximum absolute resource quantity that can be used by this container. When $request < limit$, the container is thus guaranteed the request but it can opportunistically consume the difference between `limit` and `request` if some resources are not being used by other containers. Note that `request/reservation` and `limit`

are higher-level abstractions that hide the complexity of using the lower-level parameters of the Linux CPU scheduler [18].
2. *Pod preemption and priorities*. Default schedulers of Kubernetes and Docker Swarm ensure for each resource type that the sum of the `requests` of the Pods scheduled to the same node is less than the capacity of that node; when all co-located Pods with $request < limit$ demand the use of their limit, the node is oversubscribed; as a result, in Kubernetes some Pods will be evicted from the node or throttled based on a prioritization scheme [13].
Application and cluster management features of CO frameworks support managing various non-functional requirements of the cluster or the containerized applications. All CO frameworks support *inspection of cluster-wide resource usage and cluster health* via GUIs or CLIs. Docker EE's Universal Control Plane shows CPU and memory resource consumption of nodes. Kubernetes' dashboard shows CPU and memory usage at different levels: cluster-wide, per node, and for each separate pod. DC/OS's dashboard shows CPU, memory and disk usage at similar levels: cluster-wide, per node, per service.

Kubernetes and DC/OS also offer *central monitoring of resource usage of services and Pods*. Kubernetes provides the Metrics API that allows monitoring of CPU, memory, storage and network resources at again different levels: Pods, Nodes, etc. Kubernetes also supports a more generic API that allows monitoring of custom defined resources. DC/OS supports a central Metrics API for monitoring Pods, including support for monitoring network usage per container.