# Container Orchestration Engines: A Thorough Functional and Performance Comparison

Isam Mashhour Al Jawarneh[1], Paolo Bellavista[1], Filippo Bosi[2], Luca Foschini[1], Giuseppe Martuscelli[1], Rebecca Montanari[1], Amedeo Palopoli[1]

[1]Dipartimento di Informatica – Scienza e Ingegneria, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
{isam.aljawarneh3, paolo.bellavista, luca.foschini, giuseppe.martuscelli, rebecca.montanari}@unibo.it, amedeo.palopoli@studio.unibo.it

[2] Imola Informatica, Via Selice, 66/A, 40026 Imola (BO), Italy – fbosi@imolinfo.it

**Abstract — In the last decade, novel software architectural patterns, such as microservices, have emerged to improve application modularity and to streamline their development, testing, scaling, and component replacement. To support these new trends, new practices as DevOps methodologies and tools, promoting better cooperation between software development and operations teams, have emerged to support automation and monitoring throughout the whole software construction lifecycle. That affected positively several IT companies, but also helped the transition to the softwarization of complex telco infrastructures in the last years. Container-based technologies played a crucial role by enabling microservice fast deployment and their scalability at low overhead; however, modern container-based applications may easily consist of hundreds of microservices services with complex interdependencies and call for advanced orchestration capabilities. While there are several emerging container orchestration engines, such as Docker Swarm, Kubernetes, Apache Mesos, and Cattle, a thorough functional and performance assessment to help IT managers in the selection of the most appropriate orchestration solution is still missing. This paper aims to fill that gap. Collected experimental results show that Kubernetes outperforms its counterparts for very complex application deployments, while other engines can be a better choice for simpler deployments.**

**Keywords—containerization; container orchestration engine; Docker Swarm; Kubernetes;**

## I. INTRODUCTION

The need for business agility has led to pressure for more frequent software delivery and software development techniques, known as "agile", and related DevOps methodologies and tools to make software development and delivery a continuous lifecycle [1]. Along these trends, the microservice architectural pattern has been adopted to decompose the monolithic structure in independent components that are easier to develop, manage, and (horizontally) scale. In brief, this pattern encapsulates individual core application functionalities in microservices and builds larger systems by composing microservices as building blocks. Each application consists of deployable independent services that perform specific business functions and communicate via Application Programming Interfaces (APIs).

Containers provide an ideal means to realize those microservices due to their low overhead and speed of deployment. Furthermore, they are suitable for efficient horizontal scaling that can be obtained by deploying multiple identical containers. Therefore, modern complex applications can consist of hundreds or even thousands of services with several (potentially intricate) interdependencies. According to these new trends of design and deployment, the usage of container solutions for large applications can result difficult to be adopted; that justified the introduction of a higher containerization layer known as container orchestration. Container orchestrator engine (or *container orchestrators*) automate container provisioning and management including resource scheduling, coordination, and communication across microservices, and resource booking and accounting [2, 3]. Currently, after a decade of research and development in container technologies, there are several container orchestrators available on the market, such as Docker Swarm, Kubernetes and Mesos (just to cite some of the most diffused ones).

In the last years, some related works in the literature addressed and benchmarked performances of traditional virtualization solutions vs container-based ones, in general, and in vertical domains [4, 5, 6]. However, to the best of our knowledge, focusing on container orchestrators there are still neither well-established frameworks to qualitatively compare them, nor thorough assessments of their performances, especially under heavy-load situations. To fill those voids, this paper presents a novel and original comparison of container orchestration engines that presents the following main features. First, we isolate the main functional elements to analyze existing solutions and we apply them to four main representative market players, namely, Docker Swarm, Kubernetes, Apache Mesos and Cattle. Second, we propose some performance metrics to benchmark them. Third, we show several experimental results to assess their performance behavior. Fourth, we provide some technology selection guidelines that we believe could help telco provider IT managers in the design and migration to 5G fully-software-based telco infrastructures.

The remainder of the paper is divided as follows. Section II provides the background about functional analysis and the qualitative comparison. In Section III, we introduce main performance metrics, and in Section IV we use them for

performance assessment. Section V and Section VI summarize recent related works and draw conclusions and future work.

## II. Background

In this section, we introduce needed background about the container orchestrator model, and then we use it to qualitatively compare a selection of four widely diffused container orchestrators.

### A. Container Orchestration: Model and Functional Elements

Container orchestration allows to define automated provisioning and change management workflows to operate so to always grant agreed policies and service levels. Fig. 1 depicts our reference layered orchestration engine architecture where a set of machines, through their kernel and container runtime realize the support substrate. The orchestration engine structure sits atop and consists of three layers: resource management, scheduling, and service management. In the following, due to space limitation, we provide a fast overview of main functional elements and then we use them for our qualitative analysis (see also Tables I, II, and III).

The resource management layer manages low-level resources; in this case, functional elements are the resources that can be managed/composed and include: *memory*, *CPU/GPU*, *disk space*, *volumes* (i.e., possibility to interact with the file system of the local hosting machine), *persistent volumes* (i.e., possibility to interact also with a remote cloud file system), *port* and *IP* (i.e., configuration of UDP/TCP ports and IPs within the container virtual network). It aims at maximizing utilization and minimizing interference between containers competing for resources. Table I reports the resources supported by the compared solutions.

The scheduling layer aims at using cluster resources efficiently. It typically receives user-supplied indications (e.g., placement constraints, replication degree, etc.) as input and then decides how to place all containers composing the applications. Most important capabilities (see Table II) include: *placement*, to directly control the scheduling decisions; *replication/ scaling* to express the number of microservice replicas; *readiness checking* to include a container only when it is ready to answer; *resurrection* to reenact fast long-lived processes whose job requires being always up and running; *rescheduling* to automatically restart and schedule crashed containers running on a failed node; *rolling deployment* to automatically up-/down-grades the application version; *co-location* to assert deployment constraints, such as to co-locate containers so to take advantage of local inter-process communication.

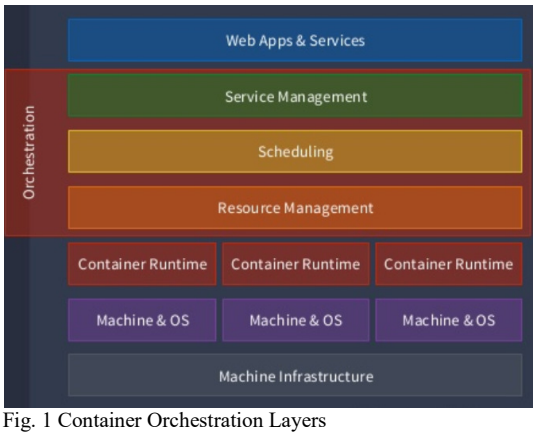

Fig. 1 Container Orchestration Layers

Finally, the service management layer provides functional capabilities for building and deploying (complex) enterprise applications. It manages high-level aspects (see Table III) that include: *labels* to attach metadata to container objects; *groups/namespaces* to isolate containers and support multi-tenancy; *dependencies* to express dependencies between microservices; *load-balancing* to divide incoming load; and *readiness checking* to make the application available online only when it is ready to accept incoming traffic.

### B. Docker Swarm, Kubernetes, Apache Mesos, and Cattle

Without any pretense of being exhaustive, in the remainder of the paper, we focus on four container orchestrators, namely, Docker Swarm, Kubernetes, Apache Mesos and Cattle. Docker Swarm and Kubernetes have been selected because they are the most diffused in the market, Mesos because it represents a very significant baseline seminal effort in the field, and finally Cattle was included since it is the reference orchestrator for Rancher. Rancher is not a container orchestrator, but rather a complete container management platform that we leveraged to deploy and run our experimental results.

Docker Swarm is a clustering and scheduling tool for Docker containers [7]. It allows IT operators to manage a cluster of Docker nodes as a single system. This is an important aspect because it creates a cooperative group of machines that provide redundancy and enable the failover mechanism if one or more nodes experience an outage. The orchestrator is based on the master/slave model for which

Table I. Resource Management Layer Comparison

| ✓ yes O partial | Swarm | Kubernetes | Mesos | Cattle |
|---|---|---|---|---|
| Memory | ✓ | ✓ | ✓ | ✓ |
| CPU | ✓ | ✓ | ✓ | ✓ |
| GPU | | | O | |
| Disk Space | | | ✓ | |
| Volume | ✓ | ✓ | ✓ | |
| Persist. Volume | | O | O | |
| Port | ✓ | ✓ | ✓ | ✓ |
| IP | O | O | O | |

Table II. Scheduling Layer Comparison

| ✓ yes O partial | Swarm | Kubernetes | Mesos | Cattle |
|---|---|---|---|---|
| Placement | ✓ | ✓ | ✓ | ✓ |
| Replication/ Scaling | ✓ | ✓ | ✓ | |
| Readiness checking | | ✓ | ✓ | ✓ |
| Resurrection | ✓ | ✓ | | ✓ |
| Rescheduling | ✓ | ✓ | ✓ | |
| Rolling Deployment | | ✓ | ✓ | ✓ |
| Co-location | | ✓ | | |

Table III. Service Management Layer Comparison

| ✓ yes O ext/part | Swarm | Kubernetes | Mesos | Cattle |
|---|---|---|---|---|
| Labels | ✓ | ✓ | ✓ | ✓ |
| Groups / Namespaces | | ✓ | ✓ | |
| Dependencies | | | ✓ | |
| Load Balancing | | ✓ | O | ✓ |
| Readiness checking | | ✓ | ✓ | |

master dominate. The master (known as Manager) is the node that is responsible for scheduling containers, whereas a slave (commonly known as Agent) is responsible for launching received containers. Both redundancy and placement are managed by the scheduling layer (see Table II). To connect containers hosted on different nodes under the same network, Docker Swarm offers an overlay network which exploits the VXLAN tunnel for creating a virtual network among hosts. The Manager node keeps tracks of the state of all nodes in a cluster(in the context of Docker Swarm this service is called discovery and is based on heartbeat mechanism that overlay networking module uses in determining whether a Docker daemon on a remote host in a cluster is still functioning). In cloud environments, elasticity is an important feature, Docker Swarm allows the fine-grained scalability of part of the service scaling one or more replicated services either up or down to the desired number of replicas.

Kubernetes is an open-source platform introduced by Google in 2014 for managing containerized applications across a cluster of machines [8]. By analogy, Kubernetes is also based on a master/slave architectural pattern for which a developer submits a list of applications to a master node and, subsequently, the platform deploys them across slave and master nodes. Master node represents a control plane of the cluster, and it can be replicated to guarantee high-availability and fault-tolerance exploiting the scheduling layer. Slave nodes (known as minions) are those nodes where application containers are executed. Kubernetes provides a containerized application as a set of containers, each of which is specific for a single microservice. Pod is a basic unit in Kubernetes and it represents a group of containers that are co-scheduled. All containers within a pod are controlled as a single application and thus share the same environment. There may be one or more containers within a pod. Since pods are co-scheduled and run in a shared context, containers within the pod can be scaled and scaled as a unique application. Pods replication is managed by Kubernetes component called Replication Controller, which is responsible for ensuring that a certain number of pods are currently providing a specific service. If current state departs from expectations such as in cases of an outage node the replication controller automatically starts the scheduling of a new instance on a different slave node. The heartbeat controller mechanism is not too aggressive and is designed with a subscriber notification system.

Apache Mesos is a seminal open source project developed by the University of California, Berkeley. The architecture consists of a master/slave design pattern in which the execution of tasks is delegated to slave nodes. The master process, running on a manager node of the cluster is responsible for managing and monitoring the whole cluster architecture. Therefore, it communicates with frameworks that aim at scheduling jobs on slave nodes [9]. Mesos solutions are often developed by installing on top of a Mesos cluster an application-level management called Marathon. Marathon interacts with the master component providing orchestration functionalities to the whole Mesos cluster. Thus, in the case of slave faults, Marathon starts a new instance to guarantee the fault-tolerance. Mesos offers high-availability replicating master nodes in order to provide failover mechanisms in case of master failures. To achieve this, it depends on Apache Zookeeper that consists of an election algorithm which elects a new node to play a master role.

In the same vein, Cattle is an orchestration engine powered by Rancher which is extensively exploited by Rancher users for creating and managing applications based on Docker containers. One of the key reasons for its extensive adoption is its compatibility with standard Docker yaml file (also known as docker-compose) syntax and Docker commands. The architecture is based on a master/slave architectural pattern and the application deployment is based on the concept of "stack". Each stack is a composition of "services" which are primarily docker images, characterized by application requirements such as scaling, health checks, service discovery links and configuration parameters.

## III. PERFORMANCE METRICS FOR CONTAINER ORCHESTRATION ENGINE

This section defines the set of metrics that we used for comparing container orchestrator performances. First, we considered the time needed to complete the deployment of the container orchestration solution (Subsection III.A). Then, we focused on the performance analysis of the scheduling and service management layers (see Fig. 1) to evaluate the time to make the application ready (for different scenarios, see Subsection III.B and III.C) and to evaluate the time to reschedule and recover from container/node faults (Subsection III.D).

### *A. Cluster provisioning time analysis*

Elastic scalability is one of the core properties to be granted by cloud computing that has been boosted and facilitated by the advent of container-based technologies that enable fast bootstrap. Along with that direction, the first performance metric evaluates the provisioning time required to provision a new cluster and to properly configure it with the required container orchestration support. Indirectly, this is also a measure of the container orchestrator complexity.

### *B. Provisioning time of applications with different complexity with local image and Docker registry*

The second performance metric, aiming at evaluating the behavior of provisioning time of container orchestrator when deploying different applications with increasing complexities. We chose three applications: Jenkins, WordPress, and GitLab.

- Jenkins is a continuous integration server. It automatically runs software tests on a non-developer machine every time someone pushes new code into the source repository. This application consists of a single container.

- WordPress is a Content Management System (CMS) based on PHP and MySql. This supports the creation and modification of websites. This application consists of two containers.
- Gitlab is a web-based Git repository manager with wiki and issue tracking features. This is important for tracking changes in computer files and coordinating work on those files among multiple users and teams. This application consists of four containers.

Moreover, we claim the importance of investigating two different scenarios. The first one exploits Docker images pre-installed on each Docker host. The second one, assuming no Docker image is available locally, downloads it dynamically from a private Docker registry running on a different physical server on the same cluster.

### C. *Provisioning time of a web application with a high number of replicas using local images and Docker registry*

This third performance metric evaluates the behavior of the container orchestrator with respect to the provisioning a high number of replicas. We use the WordPress application that represents a medium complexity application with two containers hosting, respectively, the front-end and the database. The goal is to analyze how the provisioning time is influenced by the increase in the number of replicas of the front-end container component. In addition, as for the previous metric, we claim the relevance of repeating the experiments for local and remote scenarios (i.e., w/out and with local image).

### D. *Failover Time*

Failover mechanisms increase the reliability and availability of IT resource typically using clustering technologies to provide redundant implementations. For the analyzed container orchestration solutions, failover was configured to automatically switch over to a redundant resource instance whenever the currently active IT resources become unavailable. The failure can involve a single container or the whole hosting cluster node; hence, in our analysis, we investigate both types of failures in terms of the time to fully recover from the fault and make the application usable again.

## IV. EXPERIMENTAL RESULTS

This section first introduces the experimental setup of the testbed used to collect the experimental results and then reports a through performance evaluation and discussion structured according to the four performance metrics introduced in the previous section.

### A. *Experimental Setup*

Our testbed consists of 8 physical servers, namely, MicroServer Hewlett Packard Enterprise (HPE) Proliant Gen 8. They include 2x Intel (R) Celeron(R) CPU G16610T @ 2.30 GHz processors with 12 GB of RAM. Each server is equipped with 2 hard drives of 750GB. Both devices work with 7200 rpm.

Since customizing and setting up orchestrators is a daunting task, we used Rancher [10]. Rancher includes everything required for managing containers working at a staggering level compared to orchestration perspective. It automates the setup of container orchestrator environments by allowing to neglect orchestration-specific configurations and facilitates the update and (re)configuration of new stable releases.

We have directly installed Rancher on top of our four physical servers, each operating on Ubuntu system. We have used Rancher UI tool and its monitoring internal system for measuring performance test's times. To reduce error factor, each experiment has been repeated 33 times and we show average values; we are not reporting standard deviations that are typically rather limited (always below 6% across all tests).

### B. *Results and Discussion*

This section shows performance results collected for our four representative container orchestration engines: Docker Swarm, Kubernetes, Mesos, and Cattle.

#### 1) *Cluster provisioning time*

The first set of results inspects cluster provisioning time required for deploying a single orchestration tool via Rancher. As the Fig. 2 shows, Cattle requires the shortest time to deploy a single cluster since it is the Rancher native orchestration tool, and so, the entire architecture is optimized to deploy the container orchestrator that is provided by default.

Kubernetes requires the highest provisioning time. We believe this is because its architecture is the most complex one. In fact, Kubernetes is the solution that provides the most capabilities to manage and deploy production-level services (see Tables I, II, III). Docker Swarm and Apache Mesos, instead, showed a shorter provisioning time according to their reduced complexity.

#### 2) *Provisioning time with local image and Docker registry using applications with increasing complexity*

Fig. 3a and Fig. 3b shows our second set of results for the three applications with increasing complexities (i.e., Jenkins, Wordpress, and Gitlab) using local/remote images. Obtained results show that application complexity significantly impacts the provisioning time. In fact, passing from the simplest Jenkins application (with one container) to the most complex Gitlab application (4 containers) the provisioning time increases of more than 25% for all container orchestrator. About differences due to local/remote image download and between all compared solutions, instead, they are almost

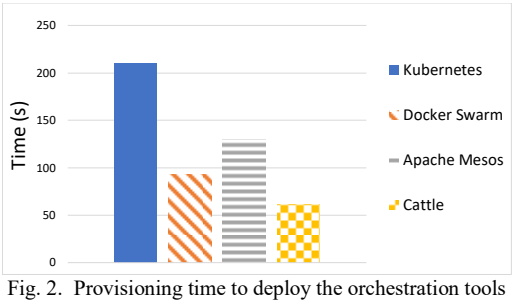

Fig. 2. Provisioning time to deploy the orchestration tools

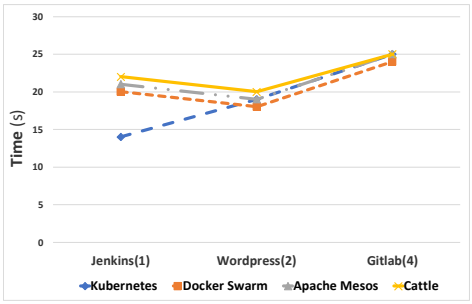


Fig. 3a. Provisioning time of applications with different complexity using local image.

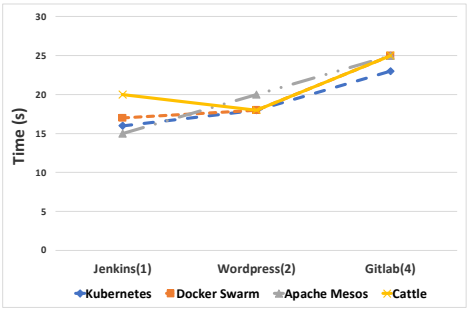


Fig. 3b. Provisioning time of applications with different complexity using Docker registry image.

negligible because we do not have a high number of containers (replicas) to manage. To better study this scalability aspect, in our second set of results we stress orchestrators with an increasing number of replicas.

*3) Provisioning time with local images and Docker registry using an application with a high number of replicas*

Fig. 4a and Fig. 4b shows the provisioning time of the medium-complexity WordPress application for an increasing number of replicas, and when Docker image is located on either each server node or downloaded from a local Docker private registry.

Results are quite different because each of the two scenarios involves different operations. In the first local download scenario (see Fig. 4a), Kubernetes takes the shortest provisioning time, while in the remote one the longest. We believe the reason for this behavior is that the interaction between the Kubernetes agent and the Docker registry introduces a considerable overhead. Cattle exhibited the best running time (let us note that it is the Rancher-native orchestration tool), thus optimizing the interaction between orchestrator agent and Docker Registry. The other two orchestrators collocate in the middle. Finally, focusing on scalability, although we heavily stressed orchestrators with a very high number of replicas (up to 100), all solutions tend to scale linearly and showed a reasonably good behavior (always below 3 minutes even in the worst-case scenario).

*4) Failover time*

Rescheduling allows restoring container services and it is natively supported by Kubernetes and Docker Swarm. However, Rancher introduces the possibility to define a minimum number of running containers for each microservice.

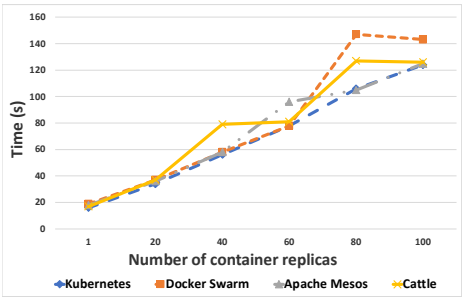


Fig. 4a. Provisioning time of a single application using local image.

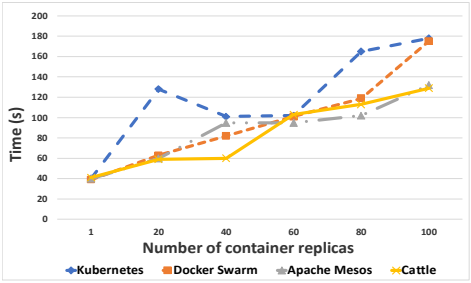


Fig. 4b. Provisioning time of a single application using Docker registry image.

Therefore, there is a dedicated Rancher functionality that provides this feature, even if the underlying container orchestrator does not offer this capability. Hence, for this set of experiments, we used the native feature for Kubernetes and Docker Swarm, whereas for Cattle and Apache Mesos we exploited the Rancher capability.

As we can see in Fig. 5a, Kubernetes is the best solution to the failure of the single container. On the contrary, as shown in Fig. 5b, it is the worst in case of a node failure. This is not surprising since in the first case failure is detected by local Kubernetes agent that guarantees the availability of running containers. In the second case, instead, there is a replication controller that is responsible for guaranteeing the rescheduling of failed pods. This approach is event-based, and the Kubernetes object is notified by the Kubernetes controller when the number of pods changes. That chain effect is the reason for the highest failover time value with Kubernetes. Docker Swarm and Cattle, instead, exploit the heartbeat functionality of the Docker architecture and that allows them to provide the shortest failover time.

## V. RELATED LITERATURE

While there are already several works in the literature focusing on containerization, such as [4, 5, 6], we were able to find only a few works that aim at addressing container orchestration and its performance evaluation.

[11] focuses on container networking in cloud virtualization architectures. They perform systematic experiments to study the performance of container networking technologies. [12] presents DockerSim, a cloud simulator which incorporates a full container deployment and its behavioral layer. It shows a set of experiments simulating behaviors of a containerized web application varying the number of executing containers. Authors in [13] focus on

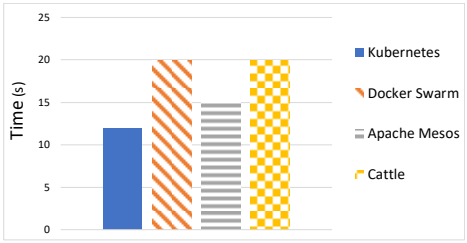

Fig. 5a. Failover Time - Container Failure.

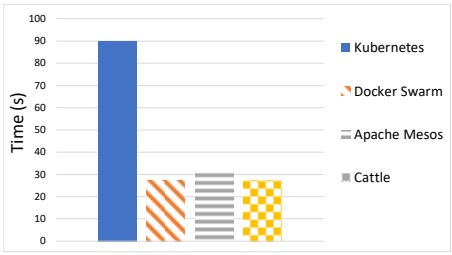

Fig. 5b. Failover Time - Host Failure.

virtualization technologies in an Internet of Things (IoT) context. It presents a performance evaluation of container technologies on constrained devices such as Raspberry Pi. Their empirical investigation evaluates, by means of various benchmark tools, the performance of Docker when running on this single board computer.

Focusing on works that, similar to ours, are addressing container orchestration, [14] investigates how container technology is employed to facilitate multi-tenancy and multi-cloud deployment of SaaS applications. Along the same line, [15] presents an easy-to-use and extensible workbench exemplar, named K8-Scalar, by implementing and evaluating different self-adaptive approaches based on Kubernetes for autoscaling container-orchestrated services.

To the best of our knowledge, there is still no effort similar to our study in terms of coverage of both qualitative and performance aspects, as well as a comparison of multiple container engines. Presented functional analyses and performance assessment discussions will help IT managers in making informed choices, especially for complex softwarized 5G telco infrastructures [16].

## VI. Conclusions and Future Works

In this paper, we have analyzed container orchestration structure comparing features and services offered by container orchestrators. We have also designed a cohesive set of metrics to compare their performances at scheduling and service management layers and we believe those metrics are reusable for all types of container orchestrators. Finally, we chose four representative container orchestrators, namely, Docker Swarm, Kubernetes, Apache Mesos, and Cattle, that we deeply inspected and assessed.

In terms of functional comparison, we notice that Kubernetes is one of the most complete orchestrators nowadays on the market. That explains why practitioners gravitate toward preferring it to other ones. At the same time, its complex architecture introduces, in some cases, a significant overhead that may hinder its performances.

The obtained results are stimulating our further research activities in the field. On the one hand, we are working to enable scalable monitoring of the Kubernetes coordination substrate by using and scaling the Istio service. On the other hand, as a longer-term goal, we are working to improve the Kubernetes coordination support to lower its overhead to cope with complex applications with stringent latency requirements.

## Acknowledgment

This research was supported by the SACHER (Smart Architecture for Cultural Heritage in Emilia Romagna) project funded by the POR-FESR 2014-20 (no. J32I16000120009) through CIRI.

## References

[1] Canonical company, “For CTO’s: the no-nonsense way to accelerate your business with containers”, February 2017, white paper.

[2] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes”, *IEEE Cloud Computing,* vol. 1, pp. 81-84, 2014.

[3] Karl Isenberg, “Container Orchestration Wars”, in *Velocity 2016*, Santa Clara, USA.

[4] Z. Li et al., “Performance Overhead Comparison between Hypervisor and Container Based Virtualization”, in *IEEE Int. Conf. on Advanced Information Networking and Applications (AINA)*, 2017, pp. 955-962.

[5] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs Containerization to Support PaaS”, in *2014 IEEE Int. Conf. on Cloud Engineering*, 2014.

[6] R. Morabito et al., “Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge”, *IEEE Internet of Things Journal,* vol. 4, pp. 1019-1030, 2017.

[7] M. Rouse, “What is Docker Swarm?”, TechTarget, August 2016. http://searchitoperations.techtarget.com/definition/Docker-Swarm.

[8] J. Ellingwood, “An Introduction to Kubernetes”, Digital Ocean, 14 October 2016. https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.

[9] R. Howard, “Orchestration With Kubernetes, Docker Swarm, and Mesos”, Dzone, July 2017. https://dzone.com/articles/orchestration-with-kubernetes-docker-swarm-and-mesos.

[10] “Rancher”, http://rancher.com/docs/rancher/v1.6/en/rancher-services/scheduler/.

[11] Yang Zhao et al., “Performance of Container Networking Technologies”, in Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet '17), pp.1-6.

[12] Z. Nikdel et al., “DockerSim: Full-stack simulation of container-based Software-as-a-Service (SaaS) cloud deployments and environments”, in *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2017, pp. 1-6.

[13] R. Morabito, “A performance evaluation of container technologies on Internet of Things devices”, in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 999-1000.

[14] Vincent Reniers, “The Prospects for Multi-Cloud Deployment of SaaS Applications with Container Orchestration Platforms”, *Middleware Doctoral Symposium'16,,* article 5, 2 pages.

[15] Wito Delnat et al., “K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters”, in *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '18)*, pp. 33-39.

[16] E Datsika et al. “Software defined network service chaining for OTT service providers in 5G networks”, IEEE Communications Magazine 55 Nov2017.