applied sciences

MDPI

Article

A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks

Eddy Truyen *, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse and Wouter Joosen

imec-DistriNet, KU Leuven, 3001 Leuven, Belgium; dimitri.vanlanduyt@cs.kuleuven.be (D.V.L.); davy.preuveneers@cs.kuleuven.be (D.P.); bert.lagaisse@cs.kuleuven.be (B.L.); wouter.joosen@cs.kuleuven.be (W.J.)

* Correspondence: eddy.truyen@cs.kuleuven.be; Tel.: +32-163-735-85

Received: 23 November 2018; Accepted: 25 February 2019; Published: 5 March 2019

Featured Application: Practitioners and industry adopters can use the descriptive feature comparison as a decision structure for identifying the most suited container orchestration framework for a particular application with respect to different software quality properties such as genericity, maturity, and stability. Researchers and entrepreneurs can use it to check if their ideas for innovative products or future research are not already covered in the overall technological domain.

Abstract: (1) Background: Container orchestration frameworks provide support for management of complex distributed applications. Different frameworks have emerged only recently, and they have been in constant evolution as new features are being introduced. This reality makes it difficult for practitioners and researchers to maintain a clear view of the technology space. (2) Methods: we present a descriptive feature comparison study of the three most prominent orchestration frameworks: Docker Swarm, Kubernetes, and Mesos, which can be combined with Marathon, Aurora or DC/OS. This study aims at (i) identifying the common and unique features of all frameworks, (ii) comparing these frameworks qualitatively and quantitatively with respect to genericity in terms of supported features, and (iii) investigating the maturity and stability of the frameworks as well as the pioneering nature of each framework by studying the historical evolution of the frameworks on GitHub. (3) Results: (i) we have identified 124 common features and 54 unique features that we divided into a taxonomy of 9 functional aspects and 27 functional sub-aspects. (ii) Kubernetes supports the highest number of accumulated common and unique features for all 9 functional aspects; however, no evidence has been found for significant differences in genericity with Docker Swarm and DC/OS. (iii) Very little feature deprecations have been found and 15 out of 27 sub-aspects have been identified as mature and stable. These are pioneered in descending order by Kubernetes, Mesos, and Marathon. (4) Conclusion: there is a broad and mature foundation that underpins all container orchestration frameworks. Likely areas for further evolution and innovation include system support for improved cluster security and container security, performance isolation of GPU, disk and network resources, and network plugin architectures.

Keywords: container orchestration frameworks; middleware for cloud-native applications; commonality and variability analysis; maturity of features; feature deprecation risk; genericity

1. Introduction

In recent years, there has been a strong industry adoption of Docker containers due to its easy-to-use approach for distributing and bootstrapping container images. Moreover, in comparison to virtual machines, Linux containers have a lower memory footprint and allow for flexible resource

allocation to improve server consolidation [1]. The popularity of Docker has also changed the way in which application software can be packaged and deployed: container images are self-contained components that can be tagged with version numbers and are made available for download from private or public Docker registries. Moreover, container images are portable across different operating systems and different cloud provider stacks [2].

Container orchestration (CO) frameworks, such as Docker Swarm, Kubernetes, and the Mesos-based Marathon, provide support for deploying and managing a multi-tiered distributed application as a set of containers on a cluster of nodes [3]. Container orchestration frameworks have also increasingly been used to run production workloads as for example demonstrated in the annual OpenStack user survey [4–6].

However, there have been several high paces of feature additions among the most popular CO frameworks as illustrated by Figure 1, which shows the number of feature additions between June 2013 and June 2018. As shown, there was a first peak of feature additions between June 2014 and January 2015 because Mesos v0.20.0 [7] added support for Docker containers and Google open-sourced Kubernetes v0.4.0 [8] that from its inception offered support for Docker containers. Moreover, Kubernetes v0.6.0 included several innovating features such as container IP and service IP networking [9], pods [10], and persistent volumes [11]. This caused a ripple effect of feature additions across the other CO frameworks. For example, support for persistent volumes has been added to Docker v1.7 [12] in June 2015. By August 2016, Docker's architecture for persistent volumes has also been supported by Mesos v1.0.0 [13]. As another example, support for container IP networking has been added to Mesos v0.25.0 [14] and Docker Swarm stand-alone v1.0.0 [15] by January 2016.

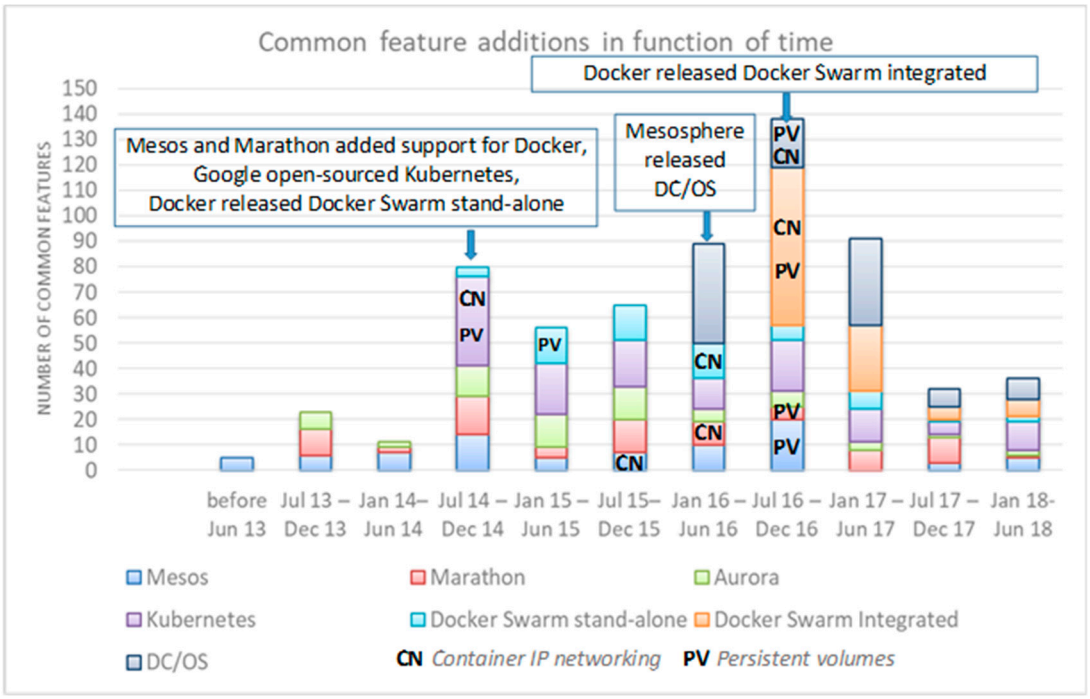

Figure 1. Density of feature additions over time (common features only).

This high pace of feature additions has been a challenge for companies to (a) keep an up-to-date understanding of what constitutes the conceptual foundation of the overall domain, to (b) determine which CO framework matches most closely with their requirements and to (c) determine which framework is most mature with respect to these requirements. This is both a risk for companies who start using container orchestration technology and companies who consider migrating from one CO framework to another framework. They are also faced with (d) feature deprecation risks, i.e., there is strong dependence on a feature that will not be supported anymore by future versions of the employed

CO framework. Finally, (e) academic researchers and entrepreneurs are also faced with the challenge that their innovative idea for a product or research prototype may become obsolete when a new version of the CO framework has been released.

An illustration of these challenges from the entrepreneur side is the story of ClusterHQ, a company that pioneered in 2014 with the container data management service Flocker [16]. Flocker initially gained a lot of traction and the company raised 12 $ million in 2015 [17] and there was a well-working integration [18] with Kubernetes, Mesos and Docker Swarm. However, by the end of 2016, the company stopped all its activities because of reportedly "self-inflicted wounds" [19]. Actually, by that time all major CO frameworks provided also built-in support for external persistent volumes.

A final challenge is to (f) keep track and interpret ongoing standardization efforts in this space. For example, the Cloud Native Computing Foundation has pushed Docker's container [20] architecture and the associated OCI specification [21] as the de-facto standard for container runtimes [22] and has pushed Kubernetes as the de-facto standard for container orchestration [23]. Indeed, Kubernetes has been the most popular framework for several years now [4,5,24] and has also the largest community on GitHub [25]. Moreover, Mesos [26] and Docker Enterprise Edition (Docker EE) [27] also provide support for Kubernetes as an alternative orchestrator. Even Amazon Web Services provides support for Kubernetes [28]. Nonetheless, the development of the other CO frameworks remains to continue and they also push other incompatible standards or architectures for networking and persistent volumes. This therefore raises the question what are the relevant standardization initiatives to which different CO frameworks align. Finally, although out-of-the-scope of this article, the expected convergence of CO frameworks with other important cloud application architectures such as micro-service architectures and server-less computing [29] will also trigger more standardization initiatives in specific vertical application segments.

1.1. Research Questions

To help address these challenges, we have performed a systematic assessment of the documentation of the aforementioned 7 CO frameworks on GitHub with respect to three main software qualities: genericity, maturity and stability. When the documentation appears inconclusive, we rely on experience drawn from earlier run-time experiments with CO frameworks or we have just tested out the specific feature.

A CO framework is defined as more generic than another when it supports more features than another framework. After all, the more features are supported, the more application and cluster configurations can be supported by a CO framework. The first aim of the systematic assessment is to determine a mapping from CO frameworks to commonly supported features and unique features. In order to provide an easy-to-navigate structure and draw higher-level insights from the results of this systematic assessment, we logically group the found features into 9 functional aspects and 27 sub-aspects that each cover a specific coherent set of related use cases (see Table 1). A functional aspect is defined as a set of related use cases that are of concern to the same type of stakeholder, whereas a functional sub-aspect is defined as an aspect of which the related use cases all represent interactions with the same architectural component or logical substrate of functionality of CO frameworks. We conduct not only a qualitative discussion of the identified aspects and CO frameworks, but also present a quantitative analysis of the number of supported features in each aspect and CO framework.

We also assess the maturity and stability of the different CO frameworks by studying the historical evolution of these CO frameworks in terms of subsequent releases on GitHub. More specifically, we have inspected all versions that are shown in Figure 2. The aim is to rank CO frameworks with respect to the time when they have released support for a particular feature for the first time. We also study the rate of feature deprecations in the development history to gather a more complete insight in the overall stability of the technological domain and we project this history of feature deprecation to an estimate of feature deprecation risks in the future.

This systematic assessment with respect to genericity, maturity and stability provides thus answers on the following 10 research questions:

With respect to genericity:

RQ1. What are the common features of CO frameworks and what are the different implementation strategies for realizing the common features?

RQ2. How can common features be organized in functional (sub)-aspects?

RQ3. What are the unique features of CO frameworks?

RQ4. How are functional (sub)-aspects ranked in terms of number of common and unique features?

RQ5. How are CO frameworks ranked in terms of number of common and unique features?

RQ6. (a) Which functional (sub)-aspects are best supported by a CO framework in terms of highest number of common features? (b) What if unique features are taken into account?

With respect to maturity:

RQ7. What is the maturity of a CO framework with respect to a common feature or a functional (sub)-aspect?

RQ8. Which functional sub-aspects are mature enough to consider them as part of the stable foundation of the overall domain? Which CO frameworks have pioneered a particular sub-aspect?

With respect to stability:

RQ9. What are the relevant standardization initiatives and which CO frameworks align with these initiatives?

RQ10. What is the risk that common or unique features might become deprecated in the future?

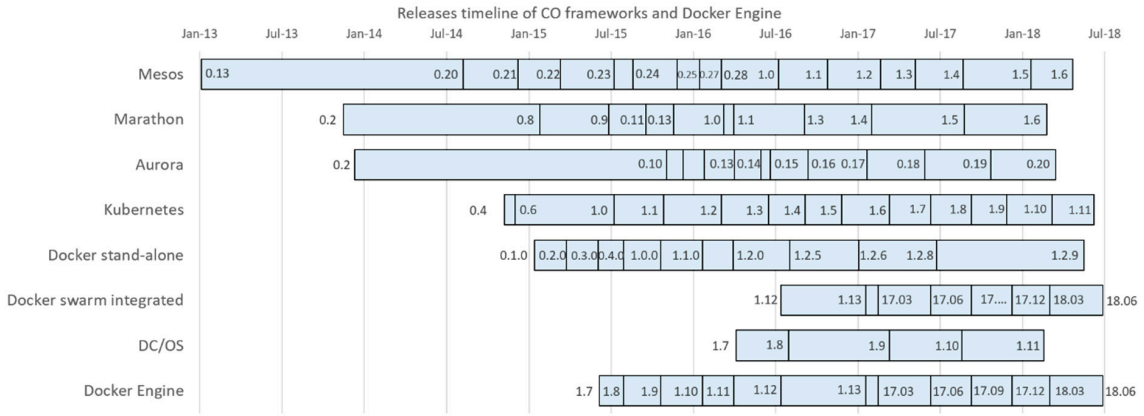

Figure 2. Timeline of when successive versions of container orchestration (CO) frameworks have been released (until September 2018).

1.2. Contribution Statement

The main contributions of this work are thus:

- A descriptive feature comparison overview of the three most prominent CO frameworks used in cloud-native application engineering: Docker Swarm, Kubernetes and Mesos.
- The study identifies 124 common and 54 unique features of all frameworks and groups it into nine functional and 27 sub-functional aspects.
- The study compares these features qualitatively and quantitatively concerning genericity, maturity, and stability.
- Furthermore, this study investigates the pioneering nature of each framework by studying the historical evolution of the frameworks on GitHub.

Table 1. Overview of functional aspects and sub-aspects and their number of common and unique features.

Functional Aspects	Functional Sub-Aspects	#Common Features	#Unique Features
Cluster architecture and setup		**13**	**2**
	Configuration management approach	1	0
	Architectural patterns	5	0
	Installation methods and deployment tools	7	2
CO system customization		**6**	**9**
	Unified container runtime architecture	3	0
	Framework design of orchestration engine	3	9
Container networks		**20**	**8**
	Services networking	8	2
	Host ports conflict management	2	0
	Plugin architecture for network services	4	0
	Service discovery and external access	6	6
Application configuration and deployment		**29**	**10**
	Supported workload types	7	1
	Persistent volumes	9	6
	Reusable container configuration	5	2
	Service upgrades	6	1
Resource quota management		**4**	**1**
Container QoS Management		**15**	**6**
	Container CPU and mem allocation with support for over-subscription	5	1
	Allocation of other resources	2	4
	Controlling scheduling behavior by means of placement constraints	3	0
	Controlling preemptive scheduling and re-scheduling behavior	5	1
Securing clusters		**9**	**4**
	User identity and access management	3	1
	Cluster network security	6	3
Securing containers		**7**	**3**
	Protection of sensitive data and proprietary software	2	0
	Improved security isolation between containers and OS	5	3
Application and cluster management		**21**	**10**
	Creation, management and inspection of cluster and applications	4	1
	Monitoring resource usage and health	4	3
	Logging and debugging of CO framework and containers	3	1
	Cluster maintenance	5	2
	Multi-cloud deployments	5	3
		124	**54**

More specifically with respect to genericity, this work will enable industry practitioners and researchers to

1. compare CO frameworks on a per feature-basis (thereby avoiding comparing apples with oranges),
2. quickly grasp what constitutes the overall functionality that is commonly supported by the CO frameworks by inspecting the 9 functional aspects and 27 sub-aspects,
3. understand what are the unique features of CO frameworks,
4. determine which functional aspects are most generic in terms of common features,
5. identify those CO frameworks that support the most common and unique features, and

6. determine those CO frameworks that support the most common and unique features for a specific functional (sub)-aspect.

With respect to maturity and stability, it will enable industry practitioners and researchers to

1. identify and understand the impact of relevant standardization efforts,
2. compare the maturity of CO frameworks with respect to a specific common feature,
3. understand which features have a higher risk of being halted or deprecated, and
4. determine those (sub)-aspects that can be considered as mature and well-understood and therefore shape the stable foundation of the technological domain; moreover, academic researchers and entrepreneurs are guided to invest their time and energy in adding innovative functional or non-functional aspects that have not yet been well supported.

Although most insights that can be derived from answering the formulated research questions will stand the test of time, some quantitative results such as the exact number of common features and the number of unique features will naturally evolve. However, we have consciously started this research around the beginning of 2017 as the pace of feature additions has clearly slowed down after June 2017 (see Figure 1). As such, we believe that most statistical evidence about significant differences in genericity between the CO frameworks will not become obsolete in any near future due to the release of new versions of the studied CO frameworks.

1.3. Structure of the Article

The remainder of this article is structured as follows. First, Section 2 overviews related surveys and research articles that provide an overview of CO frameworks. Then, Section 3 presents our research method to perform the systematic assessment. Thereafter, Sections 4 and 5 present the qualitative assessment of the genericity of the CO frameworks: Section 4 presents the assessment of the common features and functional (sub)-aspects, i.e., research questions RQ1-RQ2, and Section 5 presents the assessment of unique features, i.e., research question RQ3. Subsequently, Section 6 presents the quantitative analysis with respect to the genericity property, i.e., research questions RQ4–RQ6. Thereafter, Section 7 summarizes the results of assessing the maturity and pioneering nature of the CO frameworks, i.e., research questions RQ7–RQ8, and the assessment of the stability of the CO frameworks, i.e., research questions RQ9–RQ10. Then, Section 8 presents a look-ahead into the future of CO frameworks and outlines important missing aspects and open research questions. Thereafter, Section 9 discusses the threats to validity and the limitations of the study. Subsequently, Section 10 presents a synthesis with respect to the findings for genericity, maturity and stability and summarizes the main distinguishing characteristics of the CO frameworks. Finally, Section 11 concludes.

Note, all the collected data including hyperlinks to relevant documentation pages of CO frameworks at GitHub is available at Zenodo [30]. Moreover, an extensive technical report with detailed comparisons between the CO frameworks and detailed assessments of all the data is also available [31].

2. Related Work

There are a number of papers that mainly focus on describing (and evaluating) the common features of the Linux container technology, i.e., system virtualization, and/or specific features of Docker [32–37]. However, these works provide little to no overview of the common functions of state-of-the-art container orchestration frameworks.

Heidari et al. [38] present a survey of seven container orchestration frameworks that were identified as most promising: Apache Mesos, Mesos Marathon, Apache Aurora, Kubernetes, Docker Swarm and Fleet. This survey concisely and clearly describes the architecture of these frameworks and zooms into a number of features of these platforms. However, it does not present a systematic

assessment of commonality and variability. Moreover, it does not study the maturity of these frameworks and the risks of feature deprecation.

Jennings et al. [39] and Costache et al. [40] present classifications of resource management techniques in cloud platforms. More specifically, Jennings et al. provides a review of the literature in cloud resource management, while Costache et al. focus on complete Platform-as-a-Service (PaaS) platforms, including commercial and research solutions. The latter work by Costache et al. studies commercial solutions include Mesos [41] and Borg [42], the predecessor of Kubernetes. Costache et al. also present a list of opportunities for further research, which includes the use of container orchestration frameworks to support (i) generic resource management for any type of workload and (ii) provisioning of cloud resources from multiple IaaS clouds. However, these works do not study the resource management concepts of container orchestration frameworks in detail, such as support for oversubscription and neither includes an assessment of other functional aspects such as cluster setup tools, virtual networking, customizability, security and multi-cloud support.

Pahl et al. [43] analyses required container orchestration functions for facilitating deployment and management of distributed applications across multiple clouds and how these functions can be integrated in PaaS platforms and relevant standards for portable orchestration of cloud applications. However, these functions are presented at a high level.

Kratzke et al. [3,44] define a reference model of container orchestration frameworks, i.e., these works identify common functionalities of existing container orchestration frameworks such as scheduling, networking, monitoring and securing clusters as well as their inter-dependencies. These common functionalities are similar to the found commonalities of our study but these functionalities are described shortly at a high-level while our work decomposes each functionality into a detailed set of individual features.

In another paper, Kratzke et al. also present a domain-specific language (DSL) for specifying portable, multi-cloud application descriptors that can be translated to application descriptors for multiple container orchestration frameworks such as Docker Swarm and Kubernetes [45]. This DSL is mainly concerned with expressing common concerns that are of interest to an application manager, i.e., specifying units of deployments and configuring their allocated resources and replica levels, customizing scheduling decisions, auto-scaling rules. Additionally, Kratzke et al. [46] study concerns that are of interest to a cluster administrator in order to build a middleware platform to transfer container clusters from one cloud provider to another cloud provider. As one of the requirements of the DSL and the middleware platform is to favour pragmatism over expressiveness [47], this DSL and middleware platform supports concepts that are supported by Kubernetes, Docker Swarm and Mesos.

We confirm by a large extent the common functionalities of container orchestration frameworks as presented by the work of Kratzke et al. However, we also extend the findings of this already extensive work in several dimensions. Firstly, we relax the definition of what is a common feature, i.e., a common feature is supported by at least two CO frameworks. Secondly, we also determine unique features that are only supported by one CO framework. Thirdly, we give a systematic and exhaustive overview of all common and unique features whereas the work of Kratzke et al. presents meta-models of configuration languages that encompass concepts to support expressing cluster or application configurations that are commonly supported by all CO frameworks; in other words, our work is complementary as it can be used to refine and update the meta-models with support for common features that have not been discovered by Kratzke et al. Finally, we do not only study common features, but we also study the maturity of these common features to distinguish between stable features and those features that are relatively immature and subject to change; additionally, we also discuss the risks of feature deprecation.

In summary, to our knowledge, this is the first work that presents a detailed and exhaustive commonality and variability analysis among popular container orchestration frameworks and that studies the maturity frameworks as well as the risks of feature deprecation. The systematic approach of processing the high-quality documentation of the frameworks ensures that no features of importance are overseen in this work.

3. Research Method

This section presents how we have been working towards studying the genericity, maturity, and stability of the CO frameworks. The reader can skip this section if she or he is not interested in these methodological aspects of our work.

Before starting the research for this article, we have already acquired plenty of experience with container orchestration frameworks in the context of the DeCOMAdS research project [48] that aims to design advanced deployment and configuration middleware for adaptive multi-tenant SaaS applications. At the beginning of this project, we performed a technical SWOT (strengths, weaknesses, opportunities and threats) analysis of containers and container orchestration framework that helps a SaaS provider to make a cost-benefit analysis to move their applications to a container orchestration framework [2]. Subsequently, we have also compared the performance of Docker Swarm and Kubernetes for NoSQL databases [49] and we have built a tool for comparing different auto-scalers for container-orchestrated services in Kubernetes [50].

We have processed the documentation of the CO frameworks on GitHub because this platform has been used to manage the editing of the documentation as well as the versioning of documentation. To manage the documentation of different versions of a CO framework, git tags are typically used. These git tags allow us to dynamically browse through different versions of the documentation. This was essential for us in order to discover the addition and removal of features across versions (see Section 3.4). Only DC/OS doesn't use git tags, therefore, we have used the official website of DC/OS.

Figure 3 presents a workflow diagram of how we have extracted all relevant data from the documentation of the CO frameworks on GitHub. This data involves information about the three aforementioned software quality properties of interest. After consulting with experts in mining software repositories, it became clear that no useful or robust tools for automating one or more of these 7 steps existed. Research in the area of processing software documentation [51,52] is in its infancy. We therefore have manually executed these steps using simple repetitive strategies and pseudo algorithms in order to reduce human error extract the relevant data from GitHub.

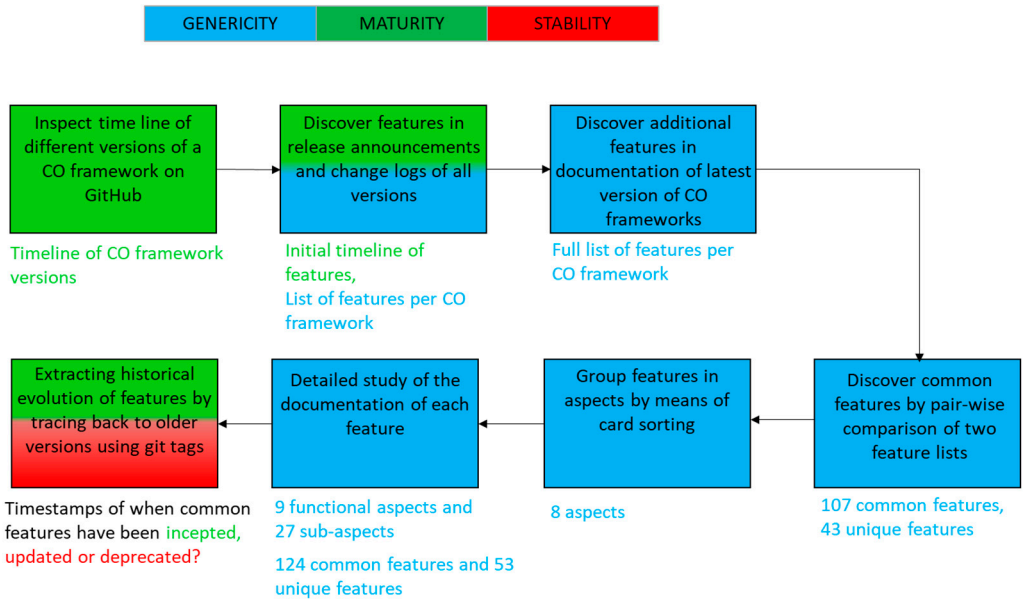

Figure 3. Overview of the method for collecting data from GitHub with respect to genericity, maturity and stability. The top of the diagram shows the 7 successive steps performed. Each step produced specific data that is related to one or more software quality properties. Each of these software qualities are presented by a colored bar. For each step, the extracted data items are specified directly below.

The following six subsections explain our method for (1) the qualitative assessment and (2) the quantitative analysis with respect to the genericity property—using the data collected in the blue bar of Figure 3, (3) the qualitative assessment of maturity using the data in the green bar, (4) the quality assessment of stability using the data in the red bar. We also explain (5) how we have gathered feedback from industry to improve the coherency and correctness of the collected data and (6) how we have dealt with the continuous evolution of the CO frameworks during the course of the research work.

3.1. Qualitative Assessment of Genericity

The following three subsections explain how (1) features of CO frameworks have been identified, (2) how common and unique features across CO frameworks have been discovered and modelled, (3) how features have been organized in functional aspects and sub-aspects.

3.1.1. Identifying Features in Documentation of CO Frameworks

Our method for identifying and modeling common and unique features among different CO framework is widely inspired on feature modeling [53,54] that is the commonly accepted method in product line engineering for modeling the commonalities and variabilities of a family of frameworks. A feature is defined as a characteristic of a framework that is visible to the end-user or as a distinguishable characteristic that is relevant to some stakeholder [54].

We have first derived an initial list of features for each CO framework separately by inspecting the release notes, change logs and feature planning documents of the latest version. We have then refined this initial list of features with additional features by reviewing the full documentation of the latest version of each CO framework. We also found multiple GitHub documentation pages that explained the same feature with different audiences or purposes in mind. We have grouped these pages so we could later study them together to fully understand the implementation strategy for the feature or discover additional related features.

3.1.2. Discovering Common and Unique Features

We have identified common features and unique features by comparing the feature lists of all CO frameworks pair-wise. We define a common feature as appearing in the documentation of two or more container orchestration frameworks (or related incubation projects) and having passed the beta stage in at least one of the frameworks.

We first identify all common features. The question of whether two documentation pages from different CO frameworks describe the same feature is concurred based on our previously acquired research experience in using and evaluating CO frameworks [2,49,50].

We then determine all unique features for each CO framework. We define a unique feature as a feature that has been documented by only one framework and other CO frameworks have no related incubation projects or design proposals on GitHub. By striking through all documentation pages of common features, we withhold documentation pages of possible unique features. This work resulted in one table with common features and one table with unique features.

3.1.3. Organizing Features in Functional Aspects and Sub-Aspects

We have organized the common and unique features in functional aspects because the number of discovered features was too large to be comprehendingly presented as a flat list. We have used the principles of card sorting [55] as the method for grouping features in usable aspects and naming these aspects. We decided that two features belong to the same aspect when they relate to similar use cases or requirements and have the same stakeholder in common.

Based on the feedback from industry (see Section 3.5), we concluded that it takes too much time to process the volume of the presented information in these tables. As such, we have refined the functional aspects into functional sub-aspects because the lists of features in some functional aspects were still too large in order to be comprehensively grasped from a helicopter view. We decided that two

features of a functional aspect belong to the same functional sub-aspect when they concern the same architectural component or logical substrate of functionality that is found in many CO frameworks.

We have then written an exhaustive inventory of common features by carefully reading the documentation pages of the CO frameworks. This helped us for a given common feature to (i) determine differences in feature implementation strategy among CO frameworks and (ii) to discover new features that are also distinguishable in other CO frameworks. Moreover, (iii) we discovered one new functional aspect and many sub-aspects; finally, we have identified 9 functional aspects and 27 functional sub-aspects (see Section 4 and Table 2).

We also classified the found unique features in the different found sub-aspects. It was possible to perform this task without introducing new functional sub-aspects (see Table A1). This increased our confidence that the set of identified sub-aspects covered the whole technological domain of container orchestration.

3.2. Quantitative Analysis with Respect to Genericity

The results of the qualitative assessment of genericity allowed us to quantify rankings between (sub)-aspects in terms of number of supported common and unique features (see Section 6, RQ4). Similarly, it possible to determine rankings between CO frameworks (see Section 6, RQ5).

To find evidence for overall significant differences between CO frameworks with respect to RQ6, we have used statistical tests for checking the overall ranking of multiple CO frameworks with respect to different sub-aspects (see Section 6, RQ6). The goal is to identify if there are significant differences in genericity between different CO frameworks, i.e., although a CO framework may support a higher number of features for several sub-aspects, the difference with other CO frameworks may still be just one or two features and therefore not significant. We have used the Friedman and Nemenyi tests that are designed with this goal in mind, but for un-replicated experimental designs [56]: un-replicated experiments take for each metric only one sample of the performance of a system, but many different metrics are evaluated; in the context of this study, metrics correspond with the 27 sub-aspects.

3.3. Study of Maturity

Initially, we have established a historical timeline of the versions of each CO framework by storing the date when each version of a CO framework has been released. We have extracted this information from official release notes (see Figure 2).

Then, for each Co framework, we have determined a historical timeline for each common feature. The historical timeline of a common feature starts with a feature addition event, then has zero or more feature update events and optionally ends with a feature removal/deprecation event. We annotate these events with the version of the CO framework during which the events have occurred. The pseudo-algorithm for defining the timelines is detailed in Section 3.3 of the technical report [31].

The obtained timelines of different CO frameworks are then merged per common feature in order to understand which CO framework pioneered in which feature and which functional sub-aspects. Detailed timelines to answer RQ7 for each common feature are available in Tables 18–26 of the technical report [31], Section 6, RQ7.

Finally, an overall assessment of the maturity of the sub-aspects has been conducted (see Section 7, RQ8). We define a sub-aspect as mature and well-understood if it meets the following three criteria: (i) the sub-aspect has been consolidated by the pioneering framework at least two traditional release cycles of 18 months [57] ago, (ii) the corresponding feature implementation strategies of the pioneering framework have at least reached beta-stage in the meantime and (iii) there are no deprecation or removal events of important features in the latest traditional release cycle.

3.4. Assessment of Stability

The existing standardization initiatives in the container orchestration space are an important indicator for the stability of the platform development artifacts of the leading CO frameworks. We have already identified the existing initiatives and the mapping towards adopting CO frameworks during the commonality analysis. As such, we could easily derive a compact table from this work to assess the overall state of these standardization initiatives (see Section 7, RQ9).

We have performed the assessment of feature deprecation risks during the last part of the writing. The risks of feature deprecation have only been assessed for the unique features because an analysis of the historical evolution of common features has shown that very few common feature implementation strategies have actually been deprecated by a CO framework. A detailed assessment of the feature deprecation risks is presented in Section 7 of the technical report [31]. A summary of the most important identified risks is shortly summarized in Section 7 of this article, as part of RQ10.

3.5. Involvement and Feedback from Industry

We have asked three senior platform developers to provide feedback on the aforementioned grouping of features into functional aspects as represented in a Google Docs document [58]. All three platform developers have worked or still work for companies who aim to create commercial platforms and tools for container orchestration in cloud computing environments. Moreover, they lead the development of installation tools and network plugins for setting up container clusters in Docker Swarm, Kubernetes and Mesos. They did not provide any substantial feedback however. This made us doubt about whether there is any interest in comparisons between CO frameworks from platform industry. When asked for the reasons of providing no feedback, it was because of lack of time.

We have also asked to review the technical report [31] by a senior developer from a software services company who has used DC/OS and Kubernetes for running their application services. Based on his feedback, we have been able to improve the clarity and correctness of the feature descriptions in the technical report.

3.6. Dealing with Continuous Evolution of CO Frameworks during the Research

We have performed the above research from April 2017 till December 2017. After that period, new versions of CO frameworks have of course been continuously released. We have kept the collected information up-to-date as follows. Each time a new version of a CO framework has been released, we reviewed the release notes and change logs of that new version in order to discover feature additions, feature updates and feature deprecations. As a result, new common features have been discovered when a unique feature of a CO framework becomes also supported by another framework; if so, timeline information was also updated.

As the article reached completion, we decide to take into account only versions released before 1 July 2018.

4. Genericity: Qualitative Assessment of Common Features

We present in this section answers to research questions RQ1–RQ2:

RQ1. What are the common features of CO frameworks and what are the different implementation strategies for realizing the common features?

RQ2. How can features be organized in functional (sub)-aspects?

We have used the OpenStack user survey as the main inspiration for selecting popular open-source CO frameworks as OpenStack itself is a cloud provider company that is fully rooted in the open-source culture and is a rather neutral with respect to promoting a specific CO framework. Figure 4 gives an overview of the most popular PaaS platforms in OpenStack deployments according to the last two surveys of October 2016 and November 2017. It shows that Kubernetes, OpenShift, Docker Swarm and

Mesos are the most used container orchestration frameworks for running production-grade services. Note that OpenShift 3.0 [59] has been completely built on top of Kubernetes and Cloud Foundry [60] also provides support for Kubernetes. Moreover, as OpenShift and Cloud Foundry are not pure container orchestration frameworks, but also offer additional PaaS development services, we choose to focus on Docker Swarm, Kubernetes and Mesos for deriving a base of common and unique features.

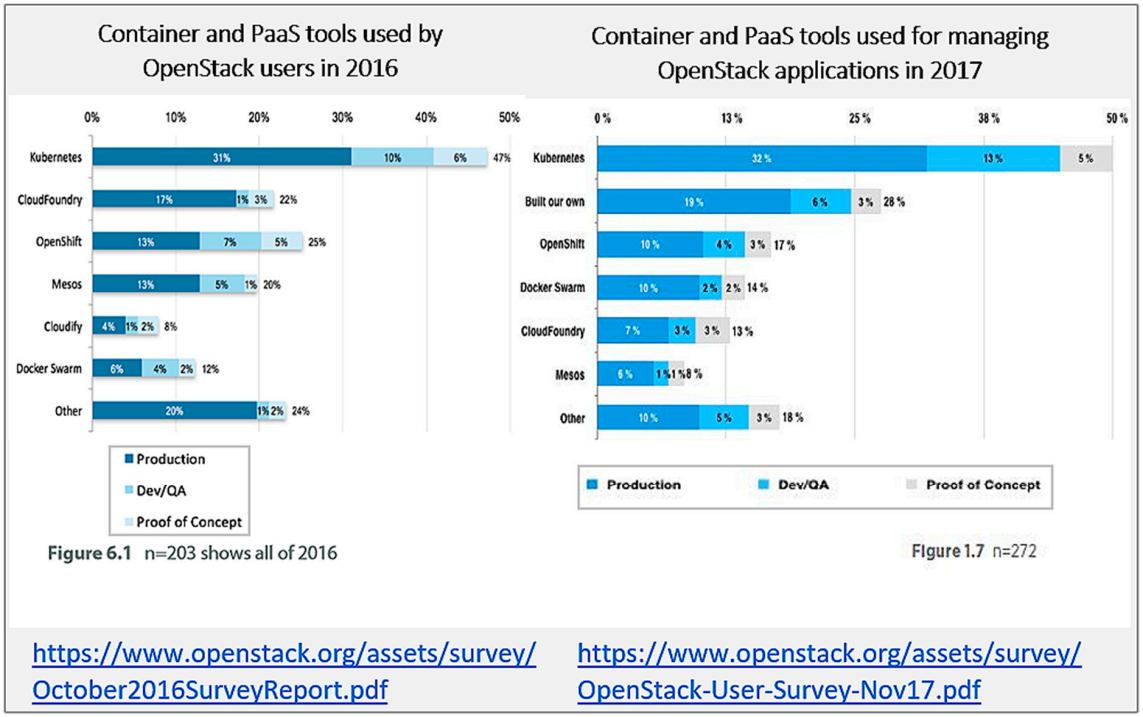

Figure 4. The two most recent annual OpenStack public user surveys show that Kubernetes, OpenShift, Docker Swarm and Mesos are the most popular container orchestration frameworks in OpenStack deployments. Cloud Foundry has decreased in popularity.

Note that Docker Swarm and Mesos actually cover different frameworks. As such, we compare in total 7 CO frameworks:

1. Kubernetes [61] supports deploying and managing both service- and job-oriented workloads as sets of containers.

Docker Swarm comes with two different distributions:

2. Docker Swarm stand-alone [62] manages a set of nodes as a single virtual host that serves the standard Docker Engine API. Any tool that already communicates with a Docker daemon can thus use this framework to transparently scale to multiple nodes. This framework is minimal but also the most flexible because almost the entire API of the Docker daemon is available. As such it is mostly relevant for platform developers that like to build a custom framework on top of Docker.
3. The newer Docker Swarm integrated mode [63] departs from the stand-alone model by re-positioning Docker as a complete container management platform that consists of several architectural components, one of which is Docker Swarm.
4. Apache Mesos [41,64] supports fine-grained allocation of resources of a cluster of physical or virtual machines to multiple higher-level *scheduler frameworks*. Such higher-level scheduler frameworks do not only include container orchestration frameworks but also more traditional non-containerized job schedulers such as Hadoop.

Currently, the following three Mesos-based CO frameworks are the most popular:

5. Aurora [65], initially developed by Twitter, supports deploying long-running jobs and services. These workloads can optionally be started inside containers.
6. Marathon [66] supports deploying groups of applications together and managing their mutual dependencies. Applications can optionally be composed and managed as a set of containers.
7. DC/OS [67] is an easy-to-install distribution of Mesos and Marathon that extends Mesos and Marathon with additional features.

We have identified in total 124 common features. A common feature is supported by at least two CO frameworks or related incubation projects and has not been released in the latest version of at least one of the frameworks. As stated above, the common features are grouped in 9 functional aspects that cover a set of related functionalities that are of concern to a single type of stakeholder. For reasons of simplicity we distinguish between two high-level stakeholders that each may subsume different user types:

- **Application Managers** develop, deploy, configure, control or monitor an application that runs in a container cluster. An application manager can be an application developer, application architect, release architect or site reliability engineer.
- **Cluster administrators** install, configure, control and monitor container clusters. A cluster administrator can be a framework administrator, a site reliability engineer, an application manager who manages a dedicated container cluster for his application, a framework developer who customizes the CO framework implementation to fit the requirements of his or her project.

A particular stakeholder, after reading the features associated to a particular functional aspect, will have a clear understanding of how CO frameworks work and how they must be operated with respect to that functional aspect. In total, we distinguish between 9 aspects. These are discussed in detail in terms of their common features throughout Sections 4.1–4.9 and presented visually in Table 2.

For each aspect, sub-aspects are indicated with a **bold paragraph heading**. For each sub-aspect, a common feature is indicated in an *italic paragraph heading*. Finally, for each common feature, different feature implementation strategies of the CO frameworks are identified based on relevant documentation webpages of the frameworks at GitHub. The URLs to these documentation pages are available as part of the bibliographic references (Direct hyperlinks are also available in table format as part of research data repository at Zenodo [30]).

Table 2 presents a summary of all common features, organized according to the 9 functional aspects and 27 sub-aspects. The first column presents the 27 sub-aspects, while the second column specifies the names of the common features. Table 2 also maps all identified common features to CO frameworks. The mapping includes structured information about (a) whether a common feature is fully or partially supported by that CO framework, (b) whether it is available in the open-source distribution or only in the commercial version of that CO framework, and (c) whether any standards related with the feature are implemented by that particular CO framework.

Table 2. Overview of the 124 common features and the mapping from features to CO frameworks.

Column Legend:

- **Sa**: Docker Swarm stand-alone
- **Si**: Docker Swarm integrated mode
- **Ku**: Kubernetes
- **Me**: Mesos
- **Au**: Mesos+Aurora
- **Ma**: Mesos+Marathon
- **Dc**: DC/OS

Cell Legend:

- ✓: The feature is fully supported by the open-source distribution of the platform.
- externalComponent: Support for the feature is not included in the open-source distribution of the CO framework, but the feature is supported by a third party component or platform. The name refers to the name of the component.
- future: The feature is not yet part of the open-source distribution of the CO framework. It has however been planned according to the documentation, or there is a separate incubation project.
- $..$: Support for the feature is not included in the open-source distribution of the CO framework, but is included in a commercial product or cloud service of the CO framework.
- *partially supported feature:* the CO framework offers partial support for the feature.
- tutorial: The feature is not directly supported by the framework, but a set of tutorials how to add auto-scaling capabilities using third-party components has been provided as part of documentation
- MsAz: Microsoft Azure, GCE: Google Cloud Engine, GKE: Google Kubernetes Engine, AWS: Amazon Web Services.
- DC/OS builds upon and extends Mesos+Marathon. Therefore, we characterize the nature of how DC/OS supports a feature as follow**s:**
  - Dlgt (Delegate): The feature is already implemented by Mesos+Marathon
  - Extnd (Extend): The feature is implemented by Mesos+Marathon, but DC/OS extends it
  - Sprsd (Supersede): The feature implementation by Mesos+Marathon is superseded by DC/OS
  - Add (Add): The feature is not supported by Mesos+Marathon, but DC/OS adds support for it.

Aspects		**Container Orchestration Frameworks**						
Sub-aspects	*Features*							
Cluster architecture and setup		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Configuration management	*Declarative configuration management*	✓	✓	✓	n/a	✓	✓	Dlgt
Architectural patterns	*Master-Worker architecture*	✓	✓	✓	✓	✓	✓	Dlgt
	Highly-available (HA) master design	✓	✓	✓	✓	✓	✓	Dlgt
	Generic, automated setup of HA masters	✓	✓	GCE, juju, tectonic		✓	✓	Dlgt
	Versioned HTTP API and client libraries	✓	✓	✓	✓		✓	Extnd
	Simple, policy-rich scheduling algorithm	✓	✓	✓	n/a	✓	✓	Dlgt
Installation methods and tools for setting up a cluster	*Dockerized CO software*	✓		✓	✓		✓	
	VM images with CO software for local dev			✓	✓	✓	✓	Extnd
	Linux packages + CLI for cluster setup		✓	✓	✓	✓	✓	Extnd
	Configuration management tools			✓	✓			
	Cloud-provider tool or platform	MsAz	MsAz	✓				MsAz
	Cloud-provider independent tools		✓	✓				Add
	Microsoft Windows or Windows Server		✓	✓	✓			
CO framework customization		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Unified container runtime architecture	*Unified container runtime architecture*	✓	✓	✓	✓	✓	✓	Dlgt
	Support for OCI specifications	✓	✓	✓	future			
	Other supported container runtimes	✓	✓	✓	✓	✓	✓	Dlgt
Framework design of orchestration engine	*External plugin architecture*	✓	✓	✓	✓		✓	Dlgt
	Plugin-architecture for schedulers			✓	✓	✓	✓	Dlgt
	Modular interceptors			✓	✓	✓		Dlgt

Table 2. *Cont.*

Aspects			**Container Orchestration Frameworks**						
Sub-aspects	*Features*								
Container networking			**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Services networking	*Routing mesh for global service ports*	*L4, ipvs-based LB distributed on all nodes*		✓	✓				
		central L4-L7 LB (without ipvs)		$Docker EE$	✓	*port mapping isolator*		✓	Extnd
	Virtual IP network for containers	*L4 distributed LB (with ipvs)*		✓	✓				Add
		with stable DNS name for services		✓	✓				Add
		IP per container	✓	✓	✓	✓		✓	Dlgt
	Host ports networking	*Mapping container port to host port*	✓	✓	✓	✓	✓	✓	Extnd
		with stable DNS name for service		✓		✓	✓	✓	Extnd
		Host mode networking	✓	✓		✓		✓	Dlgt
Host ports conflict management	*Dynamic allocation of host ports*		✓	✓		*port mapping isolator*	✓	✓	✓
	Management of host port conflicts			✓	✓				
Plugin architecture for network services	*Network plugin architecture*		✓	✓	✓	✓		✓	Dlgt
	Support for CNI specification				✓	✓	✓	✓	Dlgt
	Support for Docker's libnetwork		✓	✓		✓		✓	Dlgt
	Separation of data and control traffic		✓	✓	Multus plugin				
Service discovery and external access	*Internal DNS service*	*Distributed DNS server on all nodes*	✓	✓					Extnd
		Central DNS server			✓	✓	✓	✓	Dlgt
	DNS SRV records (only in central DNS)				✓	✓	✓	✓	Dlgt
	Bypassing the L4 service load balancer			✓	✓				
	External service access via routing mesh			✓	✓	✓		✓	Dlgt
	Co-existence of service IPs and global service ports for a single service		n/a	✓	✓	n/a	n/a	n/a	
Application configuration and deployment			**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Supported workload types	*Pods*				✓	✓		✓	Dlgt
	Container-based jobs				✓		✓		Add
	Container-based services			✓	✓		✓	✓	Dlgt
	Elastic scaling of services		✓	✓	✓		✓	✓	Dlgt
	Auto-scaling of services				✓			marathon-autoscale	
	Global containers			✓	✓				
	Composite applications		✓	✓	Helm Kom-pose			✓	Dlgt
Persistent volumes	*Local volumes*		✓	✓	✓	✓	✓	✓	Dlgt
	Automatic (re)scheduling				✓	✓	✓	✓	Dlgt
	Shareable volumes between containers		✓	✓	✓	✓			
	External volumes		✓	✓	✓	✓	✓	✓	Dlgt
	Volume plugin architecture		✓	✓	✓	✓		✓	Dlgt
	Run-time installation of volume plugins		✓	✓	*CSI*	✓		✓	Dlgt
	Docker volume plugin system support		✓	✓		✓		✓	Dlgt
	Common Storage Interface (CSI) support				✓	✓			Dlgt
	Dynamic provisioning of volumes		✓	✓	✓	*Supported for local volumes but not recommended for Docker volumes*			
Reusable container configuration	*Pass environment variable to container*		✓	✓	✓	✓	✓	✓	Dlgt
	Self-inspection API				✓			✓	Dlgt
	Separate configuration data from image			✓	✓				
	Custom ENTRYPOINT		✓	✓	✓	✓	✓	✓	Dlgt
	Custom CMD		✓	✓	✓	✓	✓	✓	Dlgt
Service upgrades	*Rolling upgrades of services*			✓	✓		✓	✓	Dlgt
	Monitoring of a rolling upgrade			✓	✓		✓	✓	Dlgt
	Rollback			✓	✓		✓		Add
	Configuration of custom readiness checks				✓			✓	Dlgt
	Customizing the rolling upgrade process			✓	✓		✓	✓	Dlgt
	Canary deployments				✓		✓		Add
	In-place updates of app configurations		✓	✓	✓				Add
	Non-disruptive, in-place updates		✓	✓	✓				

Table 2. *Cont.*

Aspects		**Container Orchestration Frameworks**						
Sub-aspects	*Features*							
Resource quota management		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Resource quota management	*Partitioning API objects in user groups*		$Docker EE$	✓	✓	✓		Add
	CPU, mem and disk quota per user group			✓	✓	✓		
	Object count quota limits per user group			✓	*ports*			
	Reserving resources for the CO framework			✓			✓	Dlgt
Container QoS management		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Container CPU and memory allocation with support for oversubscript.	*Minimum guarantees for CPU*	✓	✓	✓	✓	✓	✓	Dlgt
	Abstraction of CPU-shares		✓	✓				
	Minimum guarantees for memory	✓	✓	✓				
	Maximum limits for CPU	✓	✓	✓	✓			
	Maximum limits for memory	✓	✓	✓	✓	✓	✓	Dlgt
Allocation of other resources	*Limits for NVIDIA GPU*			*no gpu sharing*	✓	✓	✓	Dlgt
	Limits for disk resources			*local storage*	✓	✓	✓	Dlgt
Controlling scheduling behavior	*Evaluate over node labels/attributes*	✓	✓	✓	n/a	✓	✓	Dlgt
	Define custom node labels/attributes	✓	✓	✓	✓	✓	✓	Dlgt
	More expressive constraints	✓	✓	✓	n/a	✓	✓	Dlgt
Controlling preemptive scheduling and re-scheduling behavior	*Preemptive scheduling*			✓		✓		
	Container eviction when out-of-resource			✓		✓		
	Container eviction on node failure	✓	✓	✓	✓	✓	✓	Dlgt
	Container lifecycle handling		✓	✓	✓	✓	✓	Dlgt
	Re-distributing unbalanced services		✓	future				
Securing clusters		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
User identity and access management	*Authentication of users with master API*	✓	✓	✓	✓	✓	✓	Extnd
	Authorization of users with master API		✓	✓	✓	✓	✓	$Extnd$
	Tenant- aware ACLs		$Docker EE$	✓	✓	✓		Add
	Authent. of worker nodes with master API	✓	✓	✓	✓	✓		Dlgt
	Automated bootstrap of worker tokens		✓	✓	✓			
Cluster network security	*Authorization of CO agents on workers*			✓	✓	✓		
	Encryption of control messages		✓	GKE				Add
	Restricting external access to service ports		✓	✓				Add
	Encryption of application messages		✓	Weave Net				Add
Securing containers		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Protection of data and software	*Storage of sensitive-data as secrets*		✓	✓	✓		✓	Extnd
	Pull image from a private Docker registry		✓	✓	✓		✓	Extnd
Improved security isolation	*Setting Linux capabilities per container*	✓	future	✓	✓			future
	Setting SELinux labels per container	Red Hat	future	✓				
	Setting AppArmor profiles per container	✓	future	✓				
	Setting seccomp profiles per container	✓	future	✓				
	Higher-level aggregate objects	future	future	✓				
Application and cluster management		**Sa**	**Si**	**Ku**	**Me**	**Au**	**Ma**	**Dc**
Creation, management and inspection of cluster and applications	*Command-line interface (CLI)*	✓	✓	✓	✓	✓	✓	Sprsd
	Web UI	$Docker EE$		✓	✓	✓	✓	Sprsd
	Labels for organizing API objects	✓	✓	✓	✓		✓	Dlgt
	Inspection of resource usage graphs	$Docker EE$		✓		*disk usage*		Add
Monitoring resource usage and health	*Monitoring container resource usage*			✓	✓			Extnd
	Monitoring CO framework resource usage	Prometheus		✓	✓	✓	✓	Dlgt
	Framework for container health checks	✓	✓	✓	✓	✓	✓	Extnd
	Distributed events monitoring		✓	✓	✓	✓	✓	Dlgt
Logging and debugging of CO framework and containers	*Logging of containers*	✓	✓	✓	✓			Extnd
	Logging of CO framework components	✓	✓	✓	✓	✓	✓	Extnd
	Integration with log aggregator systems	$Docker EE$		✓				Add
Cluster maintenance	*Cluster state backup and recovery*		✓	future	✓	✓	✓	Dlgt
	Official cluster upgrade documentation			✓	✓	✓	✓	Extnd
	Upgrade does not affect active containers	✓	✓	kube adm	✓	✓	✓	Dlgt
	Draining a node for maintenance		✓	✓	✓		✓	Dlgt
	Garbage collection of containers/images	*images*	*images*	✓	*images*			Extnd
Multi-cloud support	*A cluster across availability zones/regions*	✓	✓	$GKE$$A	✓	✓	✓	Extnd
	Recovering from network partitions				✓	✓		
	Management of multiple clusters		✓	✓				Add
	Federated authentication across clusters			✓				Add
	Multi-zone/multi-region workloads		✓	✓	✓	✓	✓	Extnd

4.1. Cluster Architecture and Setup

This aspect represents common architectural patterns and features of CO frameworks that a cluster administrator must understand in order to be able to set up a running container cluster on top of a particular operating system and/or cloud provider infrastructure.

Configuration management approach. All container orchestration (CO) frameworks follow a declarative configuration management approach instead of an imperative configuration management approach [68].

Declarative configuration management implies that an application manager describes or generates a declarative specification of the desired state of the distributed application. The CO framework then continuously adapts the deployment and configuration of containers until the actual state of the distributed application matches the described desired state. The configuration language that is used for describing the desired state varies among CO frameworks. Docker Swarm stand-alone [69], Docker Swarm integrated mode [70] and Kubernetes [71] support the YAML mark-up language. Kubernetes also support the JSON mark-up language but recommends YAML. Aurora [72] uses the Python programming. Marathon [73] and DC/OS [74] use the JSON mark-up language.

Architectural patterns.

Master-work architecture. The core architectural pattern underlying a container cluster is very similar between all frameworks: it is based on the Master-Workers architecture where a Master node controls that running applications are always in their desired state by scheduling containers to the Worker nodes and by monitoring the actual run-time state of nodes and containers. Masters use a distributed data store (e.g., etcd, Consul, or Zookeeper) for storing the actual configuration state about all deployed containers and services.

In opposition to Kubernetes and Docker Swarm, Mesos supports a two-level scheduler architecture [75]:

1. To deal with the differences between frameworks (e.g., some frameworks execute applications in containers, while other frameworks do not), Mesos uses the generic concept of Task for launching both containerized and non-containerized processes.
2. Mesos consists of a two-level scheduler architecture, i.e., the Mesos master and multiple framework schedulers. The Mesos master offers free resources to a particular framework based on the Dominant Resource Fairness [76] algorithm. The selected framework can then accept or reject the offer, for example based on data locality constraints [41]. In order to accept an offer, the framework must explicitly reserve the offered resources [75]. Once a subset of resources is reserved by a framework, the scheduler of that framework can schedule tasks using these resources by sending the tasks to the Mesos master [77]. The Mesos master continues to offer the reserved resources to the framework that has performed the reservation. This is because the framework can respond by unreserving [78] the resources.
3. Since the task life cycle management is distributed across the Mesos master and the framework scheduler, task state needs to be kept synchronized. Mesos supports at-most-once [79], unreliable message delivery between the Mesos master and the frameworks. Therefore, when a framework's scheduler has requested the master to start a task, but doesn't receive an update from the Mesos master, the framework scheduler needs to perform task reconciliation [80].

Highly-Available (HA) master design. To ensure high-availability of the cluster, Masters can be replicated in all CO frameworks (see Table 2).

Generic and automated setup of HA masters. A fully automated and portable framework for setting up replicated Masters in different execution infrastructures is supported in Docker Swarm integrated mode [81], Aurora [82], Marathon [79] and DC/OS [83]. A fully automated HA framework for Kubernetes does not exist in the open-source distribution. However, a large number of public cloud provider services (e.g., Google Compute Engine (GCE) [84], Amazon Elastic Container Service for Kubernetes (EKS) [28] and Google Kubernetes Engine (GKE) [85]) and a number of commercial tools for installing and managing Kubernetes clusters (e.g., juju [86] and tectonic [87]) include support for an automated HA setup procedure.

Versioned HTTP API and Client API libraries. All CO frameworks except Aurora offer a versioned API that defines the concepts for specifying the desired state of the cluster and distributed applications

(see Table 2). In the remainder of this article, we refer to an atomic element in such desired state specification as an API object. For example, a request to the Master API for registering a new worker node will lead to the creation of Node object, which is specified in the aforementioned configuration languages and stored in the distributed data store of Master nodes.

To support evolution of the API, a specific versioning schema is devised for each CO framework. In general, a specific version of the API corresponds with a certain version of the CO framework. The version schema also allows demarcating stable parts of the API from those parts that are still beta. An HTTP API becomes only usable if there are client libraries available for one or more programming languages.

Simple and policy-rich scheduling algorithm. An important element of every CO framework is the scheduling algorithm used for computing on which node a container should be placed. All CO frameworks have a simple yet highly customizable scheduling algorithm. For example, the scheduling algorithms of Mesos-based frameworks [88,89] randomly select the first Mesos agent with a reserved resource offer that fits the task, but the placement decision can be restricted by means of different kinds of constraints (see Section 4.6). This simple, yet highly customizable scheduling algorithm is an interesting difference with schedulers for traditional clusters like Hadoop which must compute job placements at massive scale in a time-efficient manner such that node resources are fairly distributed across different users [76,90–93]. Container clusters, on the other hand, need to run dozens of small services that need to be organized and networked to optimize how they share data and computational power [94]. A detailed overview of the specific scheduling algorithms of Docker Swarm and Kubernetes is presented in Section 4.1.1 of the technical report [31].

Installation methods and tools for setting up a cluster. In order to simplify the installation procedure, a number of deployment methods and associated tools or platforms exist (see Table 2 for a detailed mapping towards CO frameworks):

- *Methods that install the CO software itself as a set of Docker containers.*
- *Methods that use VM images with the CO software installed for local development.*
- *Methods that install the CO software from a traditional Linux package.*
- *Methods that use configuration management tools such as Puppet or Chef.*
- *Cloud provider owned tools and APIs*
- *Cloud provider independent orchestration tools that come with specific deployment bundles for installing a container cluster on one or multiple public cloud providers.*
- *Container orchestration-as-a-Service platforms*
- *Setup-tools for Microsoft Windows or Windows Server*

4.2. CO framework Customization

This aspect corresponds with features of CO frameworks that a cluster administrator must understand in order to create a customized version of the CO framework.

Unified container runtime architecture. All CO frameworks support a *unified container runtime architecture such that multiple container runtimes can be plugged in*, and optionally different container image formats can be supported.

Support for Open Container Initiative specifications. The Open Container Initiative (OCI) [95] defines a specification for container runtimes and a specification for container images. Docker's container architecture [20] supports both specifications. Kubernetes [96] has an OCI-based implementation of its Container Runtime Interface. Mesos-based frameworks will provide support for the OCI image specification in the future [97].

Other supported container runtimes. As a consequence of the unified container runtime architecture, each CO framework also supports other container runtimes besides Docker Engine: Docker Swarm supports runC [98] that runs containers according to the OCI specification. Kubernetes supports the rkt

container runtime, runC and any other OCI-based container runtime [99]. Mesos-based frameworks support besides the Docker containerizer also the Mesos containerizer [100].

Framework design of core orchestration engine.

External plugin architectures for customizing multiple cluster operations are supported by all CO frameworks. The following cluster operations can be typically customized by means of a plugin: container networking, persistent volume operations, and Identity and Access Management (IAM). Network plugin architectures are presented in Section 4.3, volume plugin architectures are discussed in Section 4.4, and security plugin architectures are discussed in Section 4.7.

Plugin-architecture for schedulers. It is also possible to plug-in a custom scheduler in Kubernetes [101], Mesos [102], Aurora [103] (see scheduler configuration [104], parameter—offer_set_module), Marathon [105] (and by inclusion DC/OS). In Kubernetes it is even possible to plug-in multiple schedulers in parallel [106].

Modular interceptors for functional extension of the orchestration engine. Modular interceptors encapsulate specific extensions of existing CO components. For example, they are used for implementing resource quota management (see Section 4.5), authentication and authorization of users and worker nodes with respect to the Master API (see Section 4.7) and container security isolation (see Section 4.8) and even customizations to container runtimes. Multiple kinds of modular interceptors are supported by Kubernetes [107–109], Mesos [100,110] and Aurora [111,112]. These different kinds of interceptors vary according to (i) whether they support black-box or white-box specialization of the Master API and (ii) whether they can be deployed statically at compile-time or dynamically at run-time. A detailed overview of these different kinds of modular interceptors is presented in Section 4.2.1 of the technical report [31].

4.3. Container Networking

This aspect corresponds with features of CO frameworks that a cluster administrator must understand in order to customize how containers are networked, load balanced and discovered.

Services networking. A container exposes a certain service at a well-defined container port. In order to support availability and fault-tolerance, multiple replicas of the container have to be started across multiple nodes and health checked. In order to support connectivity to such container-based, replicated services the following elements are necessary: (i) a stable service name or IP address that is unique to this service irrespective of the state of the pool of containers of that service, (ii) a network with a unique network address for each container, and (iii) a service proxy that enables to lookup service network addresses and translate them to container replica network addresses; the service proxy may also encompass a load balancer to spread the workload of a service across the different replica's.

There are three different approaches to enable these three elements of services networking. We consider them as parent features that can be decomposed into a number of child features:

Routing mesh for global service ports. Here, (i) every service is identified by means of a unique port that is opened at each node of the cluster where a container replica runs, (ii) a container is thus addressed using the IP address of its local cluster node and the unique service port, (iii) at one or more nodes of the cluster a load balancer serves requests to a service port by forwarding the requests to the cluster nodes where the containers of that service are running. Load balancers (LBs) can be classified according to the following sub-features:

1. Whether the LB is *automatically distributed* on every node of the cluster vs. *centrally installed* on a few nodes *by the cluster administrator*. In the latter case, sending a request to a service port requires a multi-hop network routing to an instance of the LB.
2. Whether the LB supports *Layer 4* (i.e., TCP/UDP) vs *Layer 7* (i.e., HTTPS) load balancing. Layer 7 load balancing allows implementing application-specific load-balancing policies.
3. Whether the Layer 4 LB implementation is *based on the ipvs load balancing module of the Linux kernel*. This ipvs module is known as a highly-performing load balancer implementation [113].

4. Whether containers can run in *bridged* or in *virtual network* mode. In the former mode containers can only be accessed via a host port of the local node; the host port is mapped to the container port via a local virtual bridge. In the latter case, remote network connections to a container can be served.

Docker Swarm integrated mode [114], Kubernetes [115], Marathon [116] and DC/OS [117,118] provide support for a routing mesh. An overview of how these CO frameworks can be classified according to the above sub-features is provided in Table 2 and a detailed account is provided in (Section 4.3.1 of the technical report [31]).

Virtual IP network for containers. Here, (i) each service is either identified by means of a stable DNS name or a stable service IP address, (ii) containers run in virtual network mode, i.e., each container has a unique IP address to which can be remotely connected by means of a virtual network; this virtual network is supported by overlay network software that preferably supports IPv6 network addresses in order to allow for a massive amount of containers in a single cluster, (iii) Service IPs are load balanced by an automatically distributed Layer 4, ipvs-based load balancer, (iv) DNS names are served by an internal DNS service that is automatically installed at one or multiple nodes of the cluster. For load-balanced services with a Service IP, the DNS service resolves to the Service IP by default; otherwise, the DNS services resolves to the list of IP addresses of the containers behind that service. The DNS service of different CO frameworks can be classified according to several features which are described in the "service discovery and external access" sub-aspect.

All CO frameworks, except Aurora, support this approach (see Table 2), but Docker Swarm alone and Marathon provide only container IP networking but no service IP load balancing [31].

Host port networking. Here, (i) services are identified by means of *a stable DNS name*; (ii) container ports are exposed via a host port—here containers can run in *host mode* (i.e., share the network stack of the underlying host) or *bridge mode* (which is less performant but more secure than host mode because of the intermediate virtual bridge; (iii) in the internal DNS service, the IP addresses of the nodes on which a container of the service is deployed are registered as a list of A records and these records are returned according to the *DNS round robin* scheme; (iv) it is also possible to register exposed host ports as SRV records.

Docker Swarm integrated mode [119], Mesos+Aurora [120], Mesos+Marathon [121] and DC/OS [117] fully support this third approach.

Host ports conflict management. A common problem with host ports networking is that containers with the same host port cannot be scheduled on the same node and therefore the number of scheduled containers with the same host port is limited to the number of nodes in the cluster. For these reasons, host ports are not recommended by Kubernetes [122] except for very specific use cases such as running CO plugins as global containers (see Section 4.4) on every node of the cluster.

Dynamic allocation of host ports. To deal with host port conflicts at the same node, host ports for a container are preferably dynamically allocated so that every allocated host port is guaranteed to be unique within the cluster. Such dynamic allocation can be requested in the API object specification of a container in Docker Swarm integrated mode [123], Aurora [124] and Marathon [125].

Note that dynamic allocation of host ports also requires that the containerized application is reconfigured via a custom Docker ENTRYPOINT so that the default port of the application is changed to the dynamically allocated host port (see Section 4.4).

Management of statically specified host port conflicts on the same node. For those applications where dynamically changing the default port is not possible or too cumbersome, or those CO frameworks that do not support dynamic host port allocation, it is still possible for a container to statically reserve a particular port:

- in Docker Swarm integrated mode [126], host ports are centrally managed at the service level such that requests for creating a new service with an already allocated host port is a priori rejected
- in Kubernetes [127], the default scheduler policy (see Section 4.1) ensures that containers are automatically scheduled on nodes where the requested host port is still available.

There is no specific support for reservation of host ports in the other CO frameworks. As a workaround, scheduling constraints (see Section 4.6) can be specified per container in order to ensure that containers with the same host port are scheduled on different nodes.

Plugin architecture for network services.

Network plugin architecture. In order to support different network implementations, all CO frameworks support a common interface and composition framework for network plugins. What network plugin is preferred by an application depends on various contextual parameters such as the underlying cloud provider network, the desired performance, desired routing topology, etc. The implementation of routing mesh and/or virtual network can be customized to accommodate performance requirements of the containerized applications. The involved customizations include the implementation of the local virtual bridge, the virtual overlay network software, and the distributed load balancer. For more details, we refer to the technical report [31], Section 4.3.1.

Support for the Container Network Interface specification. A noteworthy standardization initiative for network plugins is the Container Network interface (CNI) project [128], which consists of a specification, a library for writing network plugins and a set of helper plugins. Currently, Kubernetes [129], Mesos [130] and DC/OS [131] support CNI. The CNI specification also allows for multiple networks to exist simultaneously. However, mainstream Kubernetes installation tools currently do not support the creation of multiple co-existing networks. Instead, a single network must be installed for the entire cluster when bootstrapping the master node. As such, exhaustion of the number of available subnet IP addresses is an issue. Another limitation is that most CNI plugins do not yet support hairpin mode which allows containers to reach themselves via their Service IPs [132].

Support for Docker Swarm's libnetwork. Docker Swarm uses its own networking plugin architecture, libnetwork [133]. The advantage of this architecture is that multiple networking plugins can be dynamically installed/removed and co-exist in an already running cluster. Mesos v1.0.0+ [134] and DC/OS v1.9+ [135] also support Docker's libnetwork architecture. Due to Mesos' architecture, it is however not possible to add or remove virtual networks at run-time [136], nor is it possible to connect a container to multiple Docker networks [137].

Separation of data and control traffic. Docker v17.12 [138] can be configured to use separate network interfaces for handling data traffic and swarm control traffic. For CNI-based networks, a specific Kubernetes network plugin, named Multus [139], also supports separating data and control traffic by means of distinct container network interfaces.

Service discovery and external access.

Internal DNS service. All CO frameworks support an internal DNS service for mapping service DNS names to IP addresses. A DNS service can be either deployed *centrally*, *distributed* across all nodes of the cluster, or in a *hybrid* fashion where a central and a distributed DNS service co-exist.

DNS SRV records. It is also possible to lookup named ports as SRV records [140] in Kubernetes [141], Aurora [120] and DC/OS [142].

Bypassing the Layer 4 load balancer. Kubernetes and Docker Swarm allow to bypass the built-in Layer 4 load balancer of respectively the virtual network layer and routing mesh by means of round-robin DNS. In Kubernetes [143], this feature is supported as Headless Services, which don't have a Service IP address. Instead, the IP addresses of the containers are stored as a DNS record in the internal DNS service. Of course, clients need to implement the load-balancing themselves. In Docker Swarm integrated mode [119] it is only possible to bypass the L4 load balancer if the service is exposed as a global service port via the routing mesh. A DNS lookup for the service name returns a list of IP addresses for the nodes running the containers behind that service.

Support for access to services *from outside the cluster via the routing mesh.* In order to support access to services from external clients that run 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 and the global service port of the service are reachable for external clients. DC/OS' Edge-LB load balancer is

specifically designed for this purpose [144] and also allows to load balance non-container orchestrated services of DC/OS [145].

It is also possible to let a cloud provider's load balancing service forward client requests to the L4 load balancer. However, only Kubernetes [146] supports automated configuration for this feature.

Co-existence of service IPs and service ports for a single service. Docker Swarm integrated mode [147] and Kubernetes [115] allow assigning to a single service both a global service port and a service IP. After all, both network addresses are served by the same distributed L4 load balancer.

Note that this is not possible in Marathon or DC/OS [116]: global service ports and virtual container networking cannot be combined for the same application: global service ports can only be assigned to Marathon applications of which the containers do not run in container network mode and thus these containers cannot be reached via a virtual Service IP address (which is served by DC/OS' distributed L4 load balancer [148]). As a consequence, internal clients are required to send their requests to the centrally deployed L4-L7 marathon load balancer. Vice versa, Marathon applications that do run in container network mode cannot be accessed via the marathon-lb and thus are not externally accessible. As a work-around, DC/OS containers that run in host or bridged mode can be assigned a global service port and an internal DNS name (which is resolved to a cluster node IP address [149]).

4.4. Application Configuration and Deployment

This aspect covers features of CO frameworks that an application manager must understand in order to configure, compose, deploy, scale and upgrade containerized software services and applications.

Supported workload types. All CO framework offer support for running different types of workloads: (i) user-facing latency-sensitive, elastically scalable, stateless services; (ii) throughput-sensitive job processing; and (iii) stateful applications. In this sub-aspect we zoom into the former two types of workloads while the next sub-aspect focusses on the support for stateful applications.

Smallest unit of deployment. Docker Swarm integrated mode [150] and all Mesos-based frameworks [77] propose the concept of Task, which is the smallest atomic unit of deployment. In Docker Swarm, a task encapsulates always a single container. In Mesos-based frameworks, a task encapsulates at most one container, but a task can also run non-containerized processes.

In opposition, in Kubernetes, the smallest unit of deployment is a Pod [151], which is a set of co-located containers that logically belong together and therefore are always deployed together on the same node.

Pods. The abovementioned Kubernetes concept of Pod has also be adopted in Mesos [152], Marathon [153] and DC/OS [154]. Here multiple containers can be launched atomically as part of a task group and these containers are all started inside an underlying Executor container. Such nested containers are only supported in the Mesos containerizer runtime [155].

Container-based jobs. Job-oriented workloads where jobs can be configured to execute either (i) *sequentially* or (ii) *in parallel* are supported by Kubernetes [156] and Aurora [157]. (iii) *Cron jobs*, which run at predetermined time intervals, are also supported by Kubernetes [158] and Aurora [159].

Container-based services. As already stated in Section 4.3, all CO frameworks, except Docker Swarm stand-alone, offer a Service concept for exposing a replicated set of containers to customers as a stable service endpoint that is served by the distributed load balancer or the internal DNS service. Such stable service endpoint can be represented by one or more forms: a virtual IP address, a global service port, a fully qualified DNS name, or just a unique service name. To deploy the container image that implements the service, the application manager declares, besides properties for bootstrapping containers from that image, also information related to the type of load balancing that must be used (internal to the cluster, external, DNS round-robin). A detailed account of the specific approach for each CO framework is presented in the technical report [31].

Kubernetes additionally introduces the concept of ReplicaSet [160] for managing Pod replicas of services and for restarting Pods after a node failure. A ReplicaSet attaches labels [161] to Pods.

A service configuration file then only contains a so-called label selector [162] that selects Pods based on their attached labels.

Elastic scaling of services. In all CO frameworks, the containers behind a service can be horizontally scaled in a dynamic fashion. The number of container replicas can be increased or decreased and for those CO frameworks with a built-in load balancer, the load balancer will automatically reconfigure itself to take into account newly added or removed container replicas.

Auto-scaling of services. Kubernetes [163] also supports auto-scaler functionality that automatically adapts the number of Pod replicas depending on one or more threshold values with respect to a performance or resource consumption metric. In order for the auto-scaling to work, resource monitoring features (see Section 4.9) must be enabled. The DC/OS [164] distribution of Marathon also supports auto-scaling of Marathon applications by running a Python implementation inside a separate Docker container. It also includes third-party documentation other approaches for auto-scaling services.

Global containers. For some applications or framework support services, it is necessary that a particular container image is running at every node of the cluster. This concept is supported in Docker Swarm integrated mode [165] and Kubernetes [166] as respectively global services and daemon sets.

Composite applications. Docker Swarm integrated mode and Marathon provide support for deploying multiple tiers of a distributed application in such an order that the dependencies between the tiers are respected. In Docker Swarm integrated mode [167], different service configurations and their mutual dependencies can be specified as part of a ComposeV3 file [70]. The docker stack deploy command takes as input such ComposeV3 file and deploys all the specified services together as one group while respecting the interdependencies between the services. Marathon supports a similar concept called Application groups [168].

Kubernetes does not natively support a similar concept, but several tools such as Helm [169] and Kompose [170] can be employed.

Persistent volumes. In all container orchestration frameworks, containers are stateless; when a container dies, any internal state is lost. Therefore, so-called persistent volumes, which store the persistent state, can be attached to containers. Persistent volumes can be implemented by various mechanisms: services for attaching block storage devices to virtual machines such as Cinder, distributed file frameworks such as NFS, cloud services such as Google Persistent Disk, or local volumes that reserve a subset of the local disk resources. Persistent volume mechanisms can be categorized according to the following 9 features:

Local volumes that are comprised of disk resources of a container's local host node are supported by all CO frameworks.

Automatic (re)scheduling of containers to local volumes. Containers that are configured to use a specific local volume are automatically (re)scheduled to the node where that local volume resides.

Shareable volumes between containers can be used as an asynchronous data communication channel between containers. Note however, that in general, not all types of persistent volumes support sharing.

External persistent volumes are supported by all CO frameworks. Such external volumes support managing data sizes that exceed a node's disk capacity and also allow for state recovery in case of node failures.

Volume plugin architecture. All CO frameworks except Aurora support a unified interface for different volume implementations. Overall there are two different architecture that are adopted by multiple CO frameworks: the Docker Engine plugin framework and the CSI-based plugins:

- *Support for Docker volume plugin architecture.* The Docker Engine plugin framework [171], which offers a unified interface between the container runtime and various volume plugins, is adopted by Mesos [172], Marathon [173] and DC/OS [174] in order to support external persistent volumes. In Mesos-based frameworks, Docker volume plugins must be integrated via a separate Mesos module, named dvdi [175], which requires writing plugin-specific glue code. As such a limited number of Docker volume plugins are currently supported in Mesos.

- *Support for the Common Storage Interface (CSI) specification.* The Common Storage Interface specification [176] aims to provide a common interface for volume plugins so that each volume plugin needs to written only once and can be used in any container orchestration framework. The specification also supports run-time installation of volume plugins. Typically, CSI can be implemented in any CO framework as a normal volume plugin, which itself is capable interacting with multiple external CSI-based volume plugins. Currently, CSI has been adopted by Kubernetes [177], Mesos [178] and DC/OS [179].

Support for run-time installation of volume plugins has been supported by the Docker Engine plugin framework [180] since Docker engine v1.12 and therefore also supported by Docker Swarm, Mesos, Marathon and DC/OS. Kubernetes v1.9+ [177] and Mesos v1.5+ [178] support the CSI specification [176] that allows run-time installation of external volume plugins.

Dynamic provisioning of persistent volumes is supported by most CO frameworks. This feature entails that volumes must not be manually created by the application manager in advance, but instead, volumes are automatically created when a new container is started.

The technical report [31] presents a more in-depth comparison of the CO frameworks with respect to the above features.

Reusable container configuration. There are a number of commonly supported features related to supporting generic yet configurable container images.

Passing environment variables to a container. First, all CO frameworks allow passing of environment variables to a container, which is a common way for configuring the software that is running inside the containers.

Self-inspection API. Kubernetes [181] and Marathon [182] enable a container to retrieve information about itself via a so-called downward API. Therefore, this information must not be specified as part of the container configuration or container image.

Storing non-sensitive information (such as configuration files) outside a container's image. Docker Swarm integrated mode [183] and Kubernetes [184] additionally support separating configuration data from images in order to keep containerized applications portable.

Configuring a custom ENTRYPOINT and CMD. All CO frameworks allow customizing the default ENTRYPOINT and CMD entries of a Docker image at run-time. ENTRYPOINT specifies the command that must be run when starting the container (e.g., /bin/sh—c opens a shell), while CMD specifies the arguments for the entrypoint's command (e.g., cassandra –f starts the Cassandra program of the official Cassandra container image) [31].

Service upgrades. *Rolling upgrades of services* are supported by all CO frameworks by means of restarting the containers of the service with a new image. In this way, the old version of the service gets gradually replaced with a new version. Note Aurora supports both rolling upgrades of jobs and services [185].

Monitoring the progress of a rolling upgrade. The status of the rolling upgrade can be monitored. Default health and readiness checks (see Section 4.9) are used in order to monitor the health of container replicas and the readiness of new container replicas to start processing requests. In case of failures, the upgrade can be paused, resumed or rolled back.

Configuration of custom readiness checks. It is also possible to configure custom readiness checks in Kubernetes [186] and Marathon [187].

Customizing the enactment of the rolling upgrade. Docker Swarm integrated mode [188], Kubernetes [189], Marathon and DC/OS [190] offer various options to customize how the rolling upgrade process is executed/enacted. A common enactment customization is controlling how many instances of the old and new version of the service should always be running during the upgrade.

Rollback. Docker Swarm integrated mode [191], Kubernetes [192], Aurora [193] and the DC/OS [194] distribution of Marathon support rolling back an upgrade. Aurora does not offer a command for rolling back an upgrade but can be configured to automatically rollback in case of a

failure. Note that recovering from a failed upgrade is a more complicated problem than what a rollback can resolve. In most case, it is better to roll forward by upgrading to a resolved application state [195].

Canary deployments. A variant of rolling upgrades, named blue-green deployments or canary deployments [196], intents the same effect as a rolling upgrade but allows for more manual control over the upgrade. The application manager will deploy a completely new service next to the existing service and the application manager can manually control when to redirect users from the old to the new service. Typically, this redirection is only performed after testing the health and readiness of the new service. Moreover, users are redirected in a gradual way so the old service is gradually scaled down while the new service is gradually scaled up. Kubernetes [197], Aurora [193] and DC/OS [198] support performing such canary deployments.

In-place updates of application configurations. Several CO frameworks allow narrow updates to application configuration files such as changing the value of property. This can be supported by either an update command that allows setting the value of a specific property or by means of replacing the entire configuration file of a particular API object.

Non-disruptive, in-place updates. Some CO frameworks allow one to perform the aforementioned in-place updates without stopping and restarting the affected applications. When the set of possible properties than can be updated is extensive, application managers should be aware that some properties such as resource limits can only be updated safely if a set of desired conditions holds.

4.5. Resource Quota Management

This aspect covers features of CO frameworks that a cluster administrator must understand in order to organize the hardware resources of a cluster among different teams or organizations.

Concept for partitioning API objects into logically named user groups. All container orchestration frameworks offer a concept 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 user groups is to support multi-tenancy such that the resources and services of a cluster can be divided among different tenant of the cluster.

Mesos does not offer the concept of user groups. Instead, it offers a similar concept, named framework roles [199], for dividing hardware resources across multiple scheduler frameworks.

Declaring a minimum guarantee and/or maximum limit on CPU, memory and disk quota per user group. Kubernetes, Mesos and Aurora provide support for declaring a minimum and a maximum quota of CPU, memory and disk resources per user group. More specifically:

- Kubernetes supports attaching to user groups minimal guarantees and maximum limits for CPU and memory quota [200] and maximum limits for disk quota per storage class [201].
- Mesos supports attaching to framework roles minimal guarantees [202] for CPU, mem and disk quota [203] for local volumes as well as weights [204] for dividing resources across roles.
- Apache Aurora allows attaching to user groups quota for memory and disk [205] via the aurora_admin set_quota command.

Declaring an object count quota limit for the number of API objects per user group. Kubernetes and Mesos allow assigning to user groups/framework roles a maximum number of API objects such as the number of nodes, containers, services, etc. More specifically, in Kubernetes [206], object count quota can be declared by expressing a maximum quantity for different kinds of Kubernetes API objects. In Mesos [203], port ranges can be associated with framework roles.

In Docker EE [207] high-level resources such as nodes [208], volumes [209] and services can be organized in collections, but there is no declaration of a maximum limit. The DC/OS distribution of Marathon [210] also allows organizing services into service groups without enforcing a limit on the number of services for a service group.

Reserving resources for the CO framework. The available set of resources on a node is automatically computed via the operating system in all CO frameworks. Additionally, Kubernetes [211] and

Marathon [212] can be configured to reserve a subset of the node resources for the framework's operation and local daemons.

4.6. Container QoS Management

This aspect covers features of CO frameworks that an application manager must understand in order to efficiently use the resources of a user group while also achieving the intended QoS level of its applications.

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. As such, CO frameworks are designed with the following two goals in mind:

- Resource allocation models have been developed that support QoS differentiation between containers while also allowing for over-subscription of resources to improve server consolidation.
- CO frameworks offer various mechanisms to application managers for controlling scheduling decisions that influence the performance of the application. These decisions include the placement of inter-dependent containers and data, and prioritization of containers during resource contention.

Note that the offered features do not provide strong SLA guarantees at the level of application-specific metrics (e.g., latency or throughput) but include general mechanisms that can be used to balance the competing goals of improved resource utilization and controllable performance of the application.

Container CPU and memory allocation with support for oversubscription. This sub-aspect covers common features of CO frameworks that an application manager must understand how to (i) allocate sufficient resources to a container to achieve its intended performance level, but also to (ii) allow flexible reallocation of idle resources to improve resource utilization.

In general, the allocation of computational resources to a container is governed by means of resource allocation policies. Container orchestration frameworks differ in their support for resource allocation policies and also differ in the type of resources that can be limited. In the following, we set out the available support for the different types of resources.

Minimum guarantees and maximum limits for CPU and memory. Kubernetes and Docker Swarm provide support for minimum guarantees and maximum limits for CPU and memory, while Mesos-based frameworks support minimum guarantees for CPU and maximum limits for both CPU and memory [31]. For example, Kubernetes manages a <request, limit> [213] pair per container and per Pod. A request defines the resource quantity that is always guaranteed to the container (e.g., a requests of 1.5 CPU imply that 1 CPU core and 50% of another CPU core is fully assigned to the container), while a limit specifies the maximum resource quantity that can be used by this container (e.g., a request of 1.5 CPU and a limit of 2 CPU specifies that the container is guaranteed 1.5 CPU cores, but it can take up until 2 full CPU cores if the processing power is not used by other containers). When a CPU limit is exceeded by a container, the container will be throttled [214]. When a memory limit is exceeded, the process using the most memory in the container is killed [215]. Note that when the request is set lower than the limit, the container is guaranteed the request but can opportunistically consume the difference between request and limit if some resources are not being used by other containers. It has been shown in Borg, the predecessor of Kubernetes, that setting requests and limits in the above ways increases resource utilization [42].

Abstraction of CPU-shares for enforcing CPU guarantees. Note that Mesos, Aurora, Marathon and Docker Swarm stand-alone rely on CPU-shares of the CFS Scheduler for implementing minimal guarantees. CPU-shares are however difficult to configure because CPU-shares are always defined as weights that are relative to the CPU-shares of other co-located containers: for example, if a new container is started, then the CPU-shares declared by that new container reduce the weights of the already running containers. Kubernetes [213] and Docker Swarm integrated mode [216], on the other

hand, offer higher-level abstractions for expressing minimal guarantees that hide the complexity of CPU-shares.

Allocation of other resources.

Limits for NVIDIA GPU are supported by Mesos [217], Aurora [218] and Marathon [219] (and DC/OS [220]). Kubernetes [221] offers partial support for GPU allocation because containers cannot requests fractions of a GPU, only a whole GPU, and a single GPU device can neither be shared between containers.

Limits for disk resources. Mesos offers support for hard [155] and soft limits for disk usage. Hard limits for disk usage are adopted by Aurora [222], Marathon [219] (and DC/OS). Kubernetes [223] offers support for setting a <request, limit> pair for usage of a node's local root partition (ephemeral storage).

Controlling scheduling behavior by means of placement constraints. All CO frameworks allow restricting the placement decision of the default scheduling algorithm by means of various user-specified constraints in order to improve the QoS level of applications. These user-specified constraints support placing inter-dependent application containers and data close or far from each other in the network topology. Different types of constraints are supported:

Restrict the set of nodes by evaluating over node labels. CO frameworks allow restricting the set of nodes on which a specific container can be scheduled by means of evaluating over node labels or attributes. A label is defined as a <key, value> pair. A number of such labels are predefined like the hostname of the node.

Evaluate over custom labels. Custom labels can also be defined: in Docker Swarm integrated mode [224] and Kubernetes [225], custom labels can be dynamically added or removed, whereas in Marathon [226] and DC/OS [227] custom attributes [228] can only be changed by (re)starting the Mesos agent with the desired list of attributes.

More expressive constraints. The CO frameworks differ in the expressiveness of the constraints. Docker Swarm integrated mode [229] offers *set-based inclusion operators* for both label keys and label values. Kubernetes does not only offer the same set-based inclusion operator [230], but also more expressive affinity and anti-affinity constraints [231] and constraints for restricting placement of containers to nodes with specific hardware features such as GPUs [232]. Mesos-based frameworks support SQL-alike queries such as GROUP BY that evenly divides containers across aggregate units such as racks or datacenters [89].

Controlling preemptive and re-scheduling behavior. This sub-aspect covers common features of CO frameworks that an application manager must understand in order to customize the pre-emptive scheduling and rescheduling logic of CO frameworks such that an intended QoS level for a containerized application is achieved during several exceptional conditions: (i) resource contention at the scheduler level, (ii) out-of-resource node conditions, (iii) node failures, (iv) container start failures and (v) unbalanced services of which the containers are not spread across different nodes.

Pre-emptive scheduling. Kubernetes [233] and Aurora [234] use priorities between containers for killing low-priority containers in case the scheduler cannot find a node with enough available resources for scheduling a new container.

Container eviction when a node runs out of resources. A fully packed node will likely run out of resources in Kubernetes and Docker Swarm when the maximum resource limits of multiple containers on that node are set higher than their minimum guarantees. After all, the default scheduling algorithm of Kubernetes [235] and Docker Swarm will allocate containers to a node so that, for each resource type, the sum of the containers' minimum guaranteed resources does not exceed the capacity of that node. To handle such out-of-resource conditions, Kubernetes and Aurora distinguish between different QoS classes of containers. In case a node is about to run out of resources, Pods of the lowest QoS class are evicted first.

Container eviction on node failures. Node failure detection is performed by means of different kinds of health checks by the master. When a node is considered failed, the master reschedules containers on that node to healthy nodes. Mesos [236] distinguishes between multiple failure scenarios (failed or

partitioned agents) and multiple recovery tactics depending on whether the frameworks on the failed agent have enabled checkpointing [237].

Container lifecycle handling. All CO frameworks, except Docker Swarm stand-alone, manage the life cycle of a container as a state machine involving states such as staging, pending, running, failed, completed, etc.

Re-distributing unbalanced services. As container clusters can be very dynamic, the distribution of containers over nodes may become unbalanced over time, for example when adding new nodes to the cluster. CO frameworks, by default, do not automatically re-distribute unbalanced services in order to avoid temporary service disruptions. Instead, the application manager can control by means of a direct command or higher-level policy when the containers of a particular service must be re-distributed.

4.7. Securing Clusters

This aspect covers features that a cluster administrator must understand in order to setup a secure cluster. Note that this aspect only focuses on the security provisions at the level of the container orchestration framework as it does not entail features related to the security of the applications running inside containers.

User identity and access management.

Secure access to the Master API by means of authentication and authorization of users. Different CO frameworks differ in the range of supported authentication and access control models, as well as the plug-ability of the solutions) (see technical report [31] for a detailed comparison).

Tenant-aware access control. All CO frameworks, except Docker Swarm stand-alone, support *tenant-aware access-control* that grants users, teams or organizations specific access permissions to a particular user group (see user groups and resource quota management in Section 4.5).

Cluster network security.

Authentication of worker nodes with the master API is supported by Docker Swarm stand-alone [238], Docker Swarm integrated mode [239], Kubernetes [240], Mesos [241] and Aurora [242].

Automated bootstrap of authentication tokens for worker nodes is also supported by Docker Swarm integrated mode [243] and Kubernetes [244]. An authentication token is a symmetric key that enables worker nodes to more easily register with the master node to join the cluster.

Authorization of CO agents on worker nodes towards the master API is additionally supported by Kubernetes [245], Mesos [246], and Aurora [242]. In Kubernetes, this feature allows one to grant API access at the Master node to the Kubelet agent of any node based on the containers that are currently running on that node. Mesos can be configured with an ACL to allow or deny worker nodes to (re-)register with the master. Aurora relies on Zookeeper's ACL mechanisms [247] for controlling Aurora-specific actions of the worker nodes.

Encryption of control messages between masters and workers is supported by Docker Swarm integrated mode [248] and Kubernetes Container-as-a-Service offering Google Container Engine [249]. Moreover, DC/OS [250] can be configured to startup in a strict or permissive security mode that respectively enables or enforces TLS encryption of communications between masters and agents.

Encryption of application messages is supported by Docker Swarm integrated mode [251] and DC/OS [250] as an optional feature. Finally, the Weave NET plugin of Kubernetes [252] also supports encryption of application messages.

Restricting external access to service ports. As stated in Section 4.3, containers of which the services are exposed via a service port can be accessed from outside the cluster if there exists a cluster node with a load balancer which has an IP address that is routable from outside the cluster. However, a security risk ensues that any container with a service port is susceptible to outside malicious attacks, especially if the load balancer is deployed distributed on every node. In order to manage this security risk, one needs a way to segregate the nodes of a cluster into those attached to a private network only and those attached to a private and public network.

With this end in view, Docker Swarm integrated mode [253] allows exposing master and worker endpoints at a specific IP address or network interface so that service ports on that node are only accessible from the subnet to which the endpoints are attached.

Kubernetes v1.10+ [254] added a similar feature but also allows specifying a range of IP addresses instead of a single IP address for a master or worker node.

Finally, DC/OS [255] distinguishes directly between private and public node types. Public nodes support inbound connections from outside the cluster and are thus primarily meant for externally facing load balancers such as marathon-lb or edge-lb. Private nodes cannot be directly accessed from outside the cluster.

4.8. Securing Containers

Improved support for container security is needed to deal with a large array of known security vulnerabilities at the level of container images [256] and container runtimes [257]. This aspect, therefore, covers features that an application manager must understand in order to manage sensitive information, manage passwords for getting access to private Docker repositories, and limiting the security attack interface of containers by limiting the access of containers towards the underlying Linux kernel.

Protection of sensitive data and proprietary software.

Secrets. Docker Swarm integrated mode [258], Kubernetes [259], Mesos [260], Marathon [261] and the Enterprise distribution of DC/OS [262] offer concepts for storing sensitive information such as private keys in Secret API objects which encompass one or more encrypted data fields.

Pulling images from a private Docker registry. Docker Swarm, Kubernetes, Mesos and Marathon offer support for automated login to a private Docker registry so that private images can also be pulled (see the technical report [31] for a detailed comparison).

Improved security isolation. One of the weaknesses of containers is that they have a broader security attack surface than virtual machines: containers run on the same host operating system and thereby enlarge the attack surface in comparison to virtual machines that run on a more compact hypervisor. For this reason, all major cloud providers continue to use a virtual machine as a key abstraction for representing a node in order to protect their assets.

Therefore, besides the basic isolation mechanisms at the level of linux containers, i.e., cgroups and namespaces [263], container orchestration frameworks additionally leverage existing security modules in the Linux kernel in order to configure on a per container basis what a container is allowed to do in terms of linux system calls, file permissions, etc.

These modules include SELinux, AppArmor, seccomp and Linux capabilities. SELinux [264] and AppArmor [265] are security modules that can limit by means of access control policies what a process can do. Seccomp-bpf [266] allows filtering of framework calls using a configurable policy implemented using Berkeley Packet Filter [267] rules. Linux capabilities [268] allows one to give a user-level process specific root-level permissions. As such a process can be granted access to what it needs without running the process as root.

All container orchestration frameworks have just begun to integrate with these different security modules. Note that the following security features are supported by Docker engine and therefore also for Docker Swarm stand-alone. However, these security features are not yet available for Services in Docker Swarm integrated mode.

Setting Linux capabilities per container is supported by Docker Swarm stand-alone [269], Kubernetes [270] and Mesos.

Setting SELinux labels per container is supported by the Fedora Atomic distributions of Docker-engine [271] and by Kubernetes [272].

Setting custom AppArmor profiles per container is supported by Docker Swarm stand-alone [273] and is a beta feature of Kubernetes [274].

Setting custom seccomp profiles per container is supported by Docker Swarm stand-alone [275] and there is also alpha support for seccomp profiles in Kubernetes [276].

Higher-level aggregate objects for storing multiple security profiles. Kubernetes offers a *generic aggregate object* in the Kubernetes API, named SecurityContext [277], which manages per container and per Pod which Linux capabilities, custom profiles, SELinux labels and other privileges must be applied. Docker [278] has launched a design proposal and work-in-progress library for supporting a similar generic object, called entitlements [279]. Such entitlements are actually envisioned as higher-level abstractions for encompassing security profiles of Services in Docker Swarm integrated mode as well as Pods in Kubernetes.

4.9. Application and Cluster Management

This aspect covers features of CO framework that a cluster administrator or application manager must understand in order to manage various non-functional requirements of respectively the cluster or the containerized applications. These management services rely on the Identity and Access Management functionality (see Section 4.7) in order to support customized instances of their functionality to cluster administrators and application managers.

Creation, management and inspection of cluster and applications.

Command Line Interfaces (CLIs) with a well-defined command structure are provided in all CO frameworks in order to offer user-friendly usage of the Master API.

Web UI. Docker [280], Kubernetes [281] and DC/OS [282] offer beside their HTTP-based Master API and Command-Line Interface (CLI) also a graphical user interface for inspecting and managing the state of all objects that can be managed via the Master API, e.g., nodes, services, containers, replication levels of containers, volumes, user groups, multi-tenant authorization controls etc. Erroneous states such as unhealthy containers or failed nodes can also be inspected. See the technical report [31] for a detailed comparison.

Labels for organizing and selecting subsets of API objects. The CLI and/or dashboard of Docker, Kubernetes, Mesos and DC/OS also use labels for organizing and selecting subsets of containers, services and nodes according to multiple dimensions (e.g., development vs production environments, front-end vs database tiers). Docker Swarm supports service labels [283] and node labels [284]. In Kubernetes, labels can be attached to various API objects [161] and the Kubernetes CLI and dashboard allows to select objects by their labels. Mesos supports task labels [285], while the DC/OS distribution of Marathon allows one to attach labels to Marathon applications and Mesos tasks [286].

Inspection of cluster-wide resource usage. The GUIs and associated CLIs also support inspection of (aggregate statistics of) cluster-wide resource usage in order to inspect global cluster state and health. Docker EE's Universal Control Plane [280] shows CPU and memory resource consumption of nodes. Kubernetes' dashboard [281] shows CPU and memory usage at different levels: cluster-wide, per node, and for each separate pod. Aurora's Observer component [287] enables browser-based access to disk usage metrics per task. DC/OS [282]'s dashboard shows CPU, memory and disk usage at similar levels: cluster-wide, per node, per service.

Monitoring resource usage and health.

Central monitoring of resource usage of services and containers. Kubernetes [288] supports two kinds of resource metrics pipelines: (i) a core Metrics API [289] that supports monitoring Pods for auto-scaling purposes and (ii) several independent full metrics pipelines [290] of which the most prominent are the Prometheus open-source project with built-in support for Kubernetes and Google Cloud Monitoring.

Mesos [291] exposes at every agent an HTTP endpoint with aggregate resource consumption metrics for containers running under that agent. When using marathon-lb [292] for exposing the service of a container, statistics for the network interface of that container [293] can be monitored at the same HTTP endpoint. When using a CNI network [294], network statistics of a container can also be queried.

DC/OS supports a central Metrics API [295] for monitoring containers and Marathon applications. This also involves monitoring network usage per container.

Central monitoring *of resource usage by CO framework components.* Besides monitoring the resource usage of services and containers, Docker Swarm [296], Kubernetes [297], Mesos [298], Aurora [299], Marathon [300] and DC/OS [301] also support central monitoring of (aggregated statistics) of resource usage by CO framework components. Kubernetes [302] and Mesos [303]-based frameworks also support monitoring GPU usage.

Reusable and configurable framework for checking the health of containers. All CO frameworks also offer a framework for developing custom health checks per container. Different health check methods are possible including HTTP checks and checking via a shell command. Relevant configuration parameters include the timeout period, the interval between two checks and the minimum number of consecutive failed checks for the health check to be considered failed [31].

Central monitoring of distributed events. Docker Swarm integrated mode, Kubernetes, Mesos, Aurora and Marathon also support an API for monitoring of events about new requests for creating services, container, container state changes and errors [31].

Logging and debugging of CO framework and containers.

Logging of containers is supported via the CLI and/or dashboard of Docker Swarm [304], Kubernetes [305], Mesos [306] and DC/OS' Marathon [307].

Internal logging of CO framework components is supported by all CO frameworks. Which specific logging tool is used, depends on the used deployment method: when the CO framework is deployed as a set of containers, container logging can be used; when the CO framework is installed as Linux package, the journald [308] service is used [31].

Integration with external log aggregation frameworks is documented in Docker Swarm [309], Kubernetes [310] and DC/OS [311].

Cluster maintenance.

Cluster state backup and recovery is a built-in feature of Docker Swarm integrated mode [312], Mesos [313], Aurora [314] and Marathon [315]. For Kubernetes an external project for cluster state management operations [316] such as backup and restore exists. Note that Mesos uses state machine replication (SMR) for storing the state of the entire cluster, including the state of the running frameworks. Aurora uses Mesos' SMR while Marathon does not.

Documentation about how to upgrade a running cluster to a next release is provided by Kubernetes [317], DC/OS distribution of Mesos [318], Aurora [319] and Marathon [320].

Upgrades do not affect active containers. Docker, the kubeadm deployment tool of Kubernetes and all Mesos-based frameworks support that when the Docker daemon is shut down for upgrade on a node, the containers on that node can continue running [31].

A CLI command for draining all containers from a node for maintenance is supported by Docker Swarm integrated mode [321], Kubernetes [322], Mesos [323], Marathon [324] and DC/OS [325].

Garbage collection of containers and/or images is differently supported by different CO frameworks. Docker [326] supports manual garbage collection of images only at the level of the local registry; Kubernetes' kubelet agent [327] supports automated garbage collection of container images as well as containers. Mesos v1.5 [328] supports automated garbage collection of only Docker images for the Unified Container Runtime. Finally, DC/OS extends Mesos with support for garbage collection of container images for both the Unified Container Runtime as well as the Docker containerizer. Moreover, the architecture of DC/OS [329] also includes support for garbage collection of Docker containers.

Multi-cloud support.

One cluster across multiple availability zones or regions. Docker Swarm stand-alone [330] as well as integrated mode [331] allow deploying multiple master nodes across multiple availability zones. Kubernetes [332] provides limited support for multi-zone deployments as generic support for automated HA master setups is not provided. However, Kubernetes-as-a-Service platforms such as Google Kubernetes Engine (GKE) and Amazon Elastic Container Service for Kubernetes (Amazon EKS) [28] offer scalable and highly-available Kubernetes clusters where multiple masters can be deployed across different availability zones. The design of Mesos [333]-based frameworks, in

particular DC/OS, allows that one cluster can be more easily deployed across multiple availability zones or regions because these CO frameworks have generic and automated support for setting up replicated masters (see Highly-Available Master/Manager architecture in Section 4.1).

Recovering from network partitions. Mesos [236] provides good support for dealing and recovering from network partitions. Aurora v0.20.0 [103] has added an optional and experimental feature for using the Mesos partition-aware APIs in order to customize the job or service recovery strategy. Users of Aurora can set partition policies [334] per job whether or not to reschedule and how long to wait for the partition to heal.

Management of multiple clusters across multiple clouds. Docker's Docker Cloud [335], Kubernetes' kubefed [336], and DC/OS' multi-cluster CLI [337] also offer CLI commands for managing multiple clusters across one or more cloud providers.

Federated authentication: Kubernetes's federated API [338] and DC/OS' single-sign-on across clusters [339] capability support federated authentication of users.

Multi-zone/multi-region workloads: All CO frameworks, except Docker Swarm stand-alone, allow controlling the availability of a service by spreading its containers across multiple fault domains (i.e., availability zones, regions or datacenters). Docker Swarm integrated mode [229], Mesos [340], Aurora [341], Marathon and DC/OS [342] require that nodes are in advance labeled with their zone, region or datacenter and offer a placement preference operator that ensures that containers of a service are spread across these different fault domains. Kubernetes [343] uses another approach: It uses its extensive support for federating multiple container clusters across different fault domains (see Section 5.1).

5. Genericity: Qualitative Assessment of Unique Features

This section answers RQ3:

RQ3. What are the unique features of CO frameworks?

We define a unique feature as being supported by only one CO framework and not having been released in the latest version of the framework. According to this definition, we have identified 54 unique features. We have been able to organize these 54 unique features according to the 27 functional sub-aspects (see Table A1 in Appendix A), therefore increasing our confidence that these 27 functional sub-aspects comprehensively cover the overall technical domain.

A quantitative analysis of Table A1 is presented in Section 6. A first look at this table clearly shows that Kubernetes has much more unique features than the other frameworks. The following three subsections present a brief overview of the most prevalent unique features of respectively Kubernetes, Mesos-based frameworks and Docker Swarm. A detailed overview of all unique features is presented in the technical report [31], Section 4.

5.1. Kubernetes

With respect to the sub-aspect "installation methods and tools" multiple public cloud providers exploit commercial Kubernetes-as-a-Service offerings [344].

This wide support from public cloud providers has also ensured various unique features in the open-source distribution of Kubernetes to improve the integration with cloud platforms. Most of these features belong to the sub-aspect "service discovery and external access" and include *automated integration with load balancing services of cloud providers* [146], *synchronization with external DNS providers* [345] and *IP masquerading* when Pods send messages to IP addresses outside of the Pod CIDR range [346].

With respect to CO framework customization, Kubernetes is much more extensible than the other CO frameworks [347]; more specifically, the following types of customizations to Kubernetes are supported: *cloud-provider specific functionality* [348], *API object annotations* [349], *API extension* [350] and *API aggregation* [351] and *hardware-specific device plugins* [352].

With respect to application configuration and deployment, Kubernetes offers support for *vertical Pod auto-scaling* [353] and the concept of *Podpresets* that helps developers to reuse the same piece of configuration code across multiple Pod configuration files [354]. Moreover, with respect to the sub-aspect "persistent volumes", Kubernetes offers *automated support for deploying and managing database clusters* [355], *raw block storage* [356], *on-line re-sizing of persistent volumes* without having to restart Pods [357] and support for *dynamically limiting the maximum number of volumes that can be attached to a node* [358].

With respect to container QoS management, Kubernetes offers concepts for *stronger performance isolation guarantees* for memory [359] and CPU resources [360]. Moreover, it allows cluster administrators to define *custom node resources* of random kind with the limitation that resource quantities must be integers and oversubscription is not support [361].

With respect to cluster security, Kubernetes provides support for *auditing* [362], *authentication and authorization for access to the HTTP API of worker agents* [363] and *network policies* [364] that are specifications of how groups of Pods are allowed to communicate with each other.

With respect to container security, Kubernetes offers *Pod security policies* for admission control of Pods to validate that Pods have declared the appropriate access control profiles, Linux capabilities, and privileges [365]. Kubernetes also offers a *Linux sysctl interface* [366] that enables cluster administrators to modify Linux kernel parameters at runtime and in a safe manner.

Finally, Kubernetes exhibits unique features in several applications and cluster management sub-aspects: (i) with respect to the sub-aspect "monitoring resource usage and health", it supports a *cluster autoscaler* [367] for automatically adding or removing nodes to a cluster, (ii) with respect to the sub-aspect "logging and debugging of CO framework and containers", Kubernetes supports *port forwarding* [368] that allows a developer to connect his local workstation to a running Pod, (iii) with respect to the sub-aspect "cluster maintenance", Kubernetes supports a *disruption budget* [369] enabling an application manager to limit the number of concurrent voluntary disruptions that his application experiences due to cluster maintenance operations, and (iv) with respect to the sub-aspect "multi-cloud support", Kubernetes offers a separate *federated API* with federated instantiations of several single-cluster API objects [370] such as deployments. This API is implemented by a separate policy-based controller plane that additionally supports *federated management of service shards* that are running across multiple clusters in different availability zones [371].

5.2. Mesos-Based Frameworks

Mesos-based frameworks have various unique features that logically ensue from the two-level scheduler architecture with a central Mesos master and multiple framework schedulers:

- Mesos is better suited for managing datacenter-scale deployments given the *inherent improved scalability properties* of its two-level scheduler architecture with multiple distributed scheduler frameworks [41].
- Mesos offers a *resource provider abstraction* [372] for customizing Mesos worker agents to framework-specific needs.
- DC/OS offers *integrated support for load-balancing Marathon-based services as well as load-balancing of workloads that are managed by other, non-container-based frameworks* [373].
- *Local volumes can be shared by tasks from different frameworks* [374].
- *Framework rate limiting* [375] aims to ensure performance isolation between frameworks with respect to request rate quota towards the Mesos Master.

Mesos has also various unique features to support high-performance computing:

- DC/OS has made the conscious design choice to enable *database-as-a-service offerings without using containers for running database instances* [376], presumably because of the non-negligible performance overhead of container orchestration frameworks for managing database clusters [49].

- With respect to container QoS management, Mesos contributes to improved network performance isolation between containers for both routing mesh networking [377] and virtual networks [378]. Unfortunately, little of these features are currently used by the CO frameworks Aurora and Marathon. Only container port ranges in routing mesh networks can be isolated in Marathon.

5.3. Docker Swarm

With respect to the sub-aspect "reusable container configuration", in both Docker Swarm stand-alone [379] and Docker Swarm integrated mode [380], *an option can be set for automatically running a simple service initialization system inside containers*.

With respect to the "service upgrades" sub-aspect, Docker Swarm integrated mode allows *customizing the enactment of a rollback* [191] of a service.

With respect to container QoS management, Docker Swarm stand-alone supports *run-time updating resource reservations and limits of a container* without needing a restart of the container [381]. These operations should be managed with care and preferably as automated as possible to avoid human-errors by application managers [382].

Finally, with respect to cluster and application management, Docker's CLI comes with a very-handy *command-line completion for Docker Swarm integrated mode* [383].

6. Genericity: Quantitative Analysis

This section presents the results of the quantitative analysis of the collected data in Sections 4 and 5 to determine evidence of significant differences in genericity between aspects and CO frameworks. We structure the presentation of these results in accordance with the research questions RQ4–RQ6 (see Section 1.1).

A CO framework is more generic than another CO framework when it supports a higher number of common features. After all, the more features are supported, the broader the set of application and cluster configurations that can be supported and managed by a CO framework. The same measure can also be used to quantify differences in genericity between (sub)-aspects.

We also take into account the number of unique features for quantifying the differences in genericity because Kubernetes has a relatively large number of unique features. Since Kubernetes is already supported by many public cloud providers and Docker EE and DC/OS also offer support for Kubernetes as an alternative orchestrator, these unique features are widely available at a large set of private and public cloud platforms.

RQ4: How are functional (sub)-aspects ranked in terms of number of common and unique features?

Table 3 presents an overview of the number of common and unique features found for the 9 aspects of container orchestrations. The table ranks the aspects according to the number of common feature implementation strategies by CO frameworks. We see that the functional aspects of "application configuration and deployment", "application and cluster management", "container networking" and "container QoS management" count the most common feature implementation strategies. On the other hand, the aspects of "securing containers", and "resource quota management" count the lowest number of common feature implementation strategies.

Table 3. Functional aspects ranked according to the number of common feature implementation strategies by CO frameworks. If a common feature is partially supported by or only supported in the commercial version of a CO framework, the implementation strategy is counted as $\frac{1}{2}$. Finally, the number of common and unique features of each functional aspect are also presented.

Aspects	#Common Features	#Implementation Strategies	#Unique Features
Application configuration and deployment	29	130.5	10
App and cluster management	21	104	10
Container networking	20	82	8
Container QoS Management	15	69	6
Cluster architecture and setup	13	63	2
Securing clusters	9	36	4
CO framework customization	6	32	9
Securing containers	7	19.5	3
Resource quota management	4	12.5	1
Total	124	548.5	53

Table 4 ranks the functional sub-aspects according to the number of common feature implementation strategies. Again, this metric is a measure for ranking sub-aspects in terms of genericity.

Table 4. Functional sub-aspects ranked according to the number of common feature implementation strategies by CO frameworks. Features that are only partially supported by CO framework or that are only offered by the commercial version are counted as a $\frac{1}{2}$.

Sub-aspects	#Common Features	#Implement. Strategies	#Unique Features
Persistent volumes	9	47	6
Services networking	8	35	2
Service upgrades	8	32	1
Architectural patterns	5	31.5	0
Reusable container configuration	5	26	2
Installation methods and deployment tools	7	25.5	2
Supported workload types	7	25.5	1
Cluster maintenance	5	25	2
CPU and mem allocation with support for over-subscription	5	23	1
Creation, management and inspection of cluster/applications	4	22.5	1
Service discovery and external access	6	22	6
Monitoring resource usage and health	4	22	3
Multi-cloud deployments	5	19.5	3
Controlling scheduling behavior	3	19	0
Cluster network security	6	18.5	3
Controlling preemptive (re)-scheduling behavior	5	18	1
Plugin architecture for network services	4	17.5	0
User identity and access management	3	17.5	1
Unified container runtime architecture	3	17	0
Framework design of orchestration engine	3	15	9
Logging and debugging of CO framework and containers	3	15	1
Resource quota management	4	12.5	1
Protection of sensitive data and proprietary software	2	10	0
Improved security isolation	5	9.5	3
Allocation of other resources	2	9	4
Host ports conflict management	2	7.5	0
Configuration management approach	1	6	0
Total	124	548.5	53

A first finding from Table 4 is that the sub-aspect "persistent volumes" counts the most common features and the most common feature implementation strategies. This is because of two reasons:

- Besides the main functional requirement of persistent storage, various orthogonal orchestration features for management of persistent volumes can be distinguished. Moreover, most of these features are supported by almost all CO frameworks.
- The adoption of the Docker volume plugin architecture by Mesos-based systems as well as the CSI specification by Kubernetes and Mesos has also been recorded as an additional feature.

Secondly the sub-aspect "services networking" counts also a high number of common features because of two similar reasons:

- No less than 3 alternative approaches to services networking can be distinguished that are all supported by multiple CO frameworks and within each alternative approach one can distinguish at a lower nested level between different alternative load balancing strategies.
- There are again two standardization initiatives related to this sub-aspect: Docker's libnetwork architecture and the CNI specification.

RQ5: How are CO frameworks ranked in terms of number of supported common features?

As shown in Figure 5, Kubernetes implements the highest number of common features, but also supports the highest number of unique features.

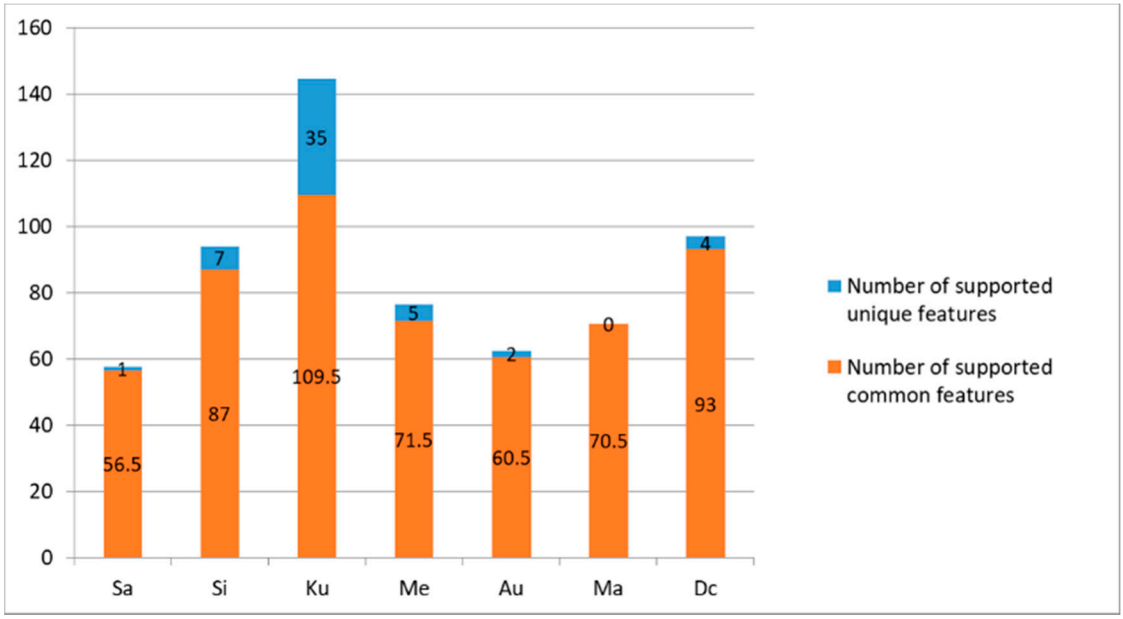

Figure 5. Comparison of CO frameworks according to the total number of supported features. Features that are partially supported by a CO framework or that are only offered by the commercial version of the framework are counted as a $\frac{1}{2}$.

RQ6a. Which functional (sub)-aspects are best supported by a CO framework in terms of common features?

As shown in Figure 6, Kubernetes implements the highest number of common features for 6 aspects. Docker Swarm integrated mode supports the most common features for the aspects "container networking" and "securing clusters". Finally, DC/OS supports the most common features for the aspect "application and cluster management".

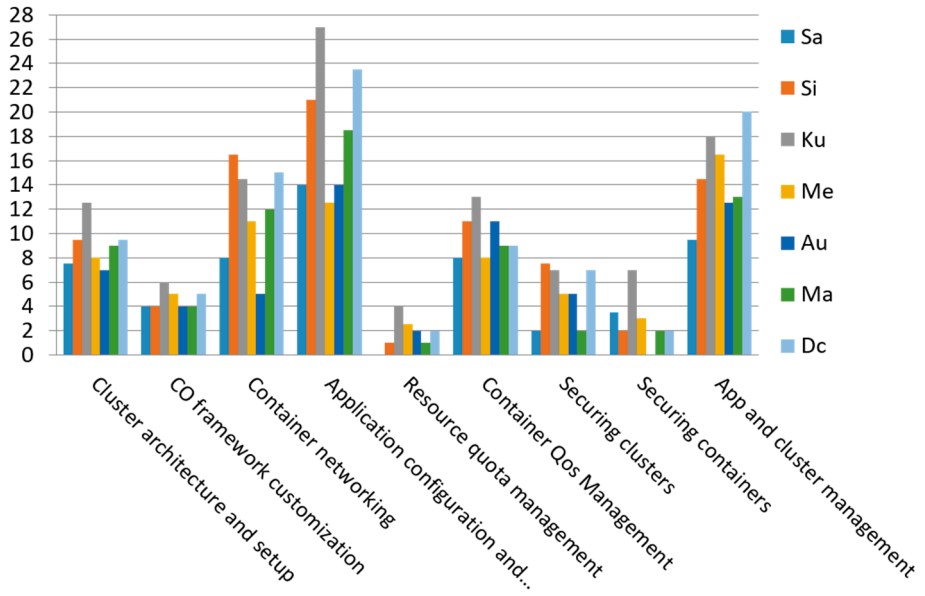

Figure 6. The number of common feature implementation strategies supported by each CO framework is shown for each of the 7 CO frameworks.

Table 5 presents an overview of the number of common feature implementation strategies per CO framework and per (sub)-aspect. We find significant differences in ranking between the frameworks when applying the Friedman test for unreplicated designs [56] (p-value = 2.668×10^{-8}). To deal with tied observations in this test, we compute ranks using R's rank() method where ranks for tied observations are replaced by their mean.

We also performed during post-hoc analysis a pairwise comparison between CO frameworks using the Nemenyi multiple comparison test with q approximation for unreplicated blocked data [56] (see Figure 7).

	Sa	Si	Ku	Me	Au	Ma
Si	0.06877	-	-	-	-	-
Ku	3.5e-05	0.44516	-	-	-	-
Me	0.63667	0.90711	0.03010	-	-	-
Au	0.99977	0.17533	0.00021	0.85620	-	-
Ma	0.89562	0.65762	0.00609	0.99918	0.98311	-
Dc	0.00182	0.93666	0.97533	0.27595	0.00765	0.09599

Figure 7. Resulting p-values of the Nemenyi multiple comparison test. For p-values <= 0.05, we can reject the null hypothesis, i.e., there is no significant difference in overall ranking between a pair of CO frameworks).

Based on the p-values of the Nemenyi test, we find that Docker Swarm stand-alone and Aurora differ significantly in genericity from both Kubernetes and DC/OS. Moreover, there is a significant difference between Kubernetes on the one hand and Mesos and Marathon on the other hand:

- Docker Swarm stand-alone and Aurora are indeed clearly less generic in terms of offered features than the other CO frameworks. After all, Aurora is specifically designed for running long-running jobs and cron jobs, while Docker Swarm stand-alone is also a more simplified framework with substantial less automated management. We only recommend Docker Swarm stand-alone as a possible starting point for developing one's own CO framework. This is a relevant direction because 28% of surveyed users in the most recent OpenStack survey [4], responded that they have built their own CO framework instead of using existing CO frameworks (see also Figure 4). We make such recommendation because the API of Docker Swarm stand-alone is the least restrictive

in terms of the range of offered options for common commands such as creating, updating and stopping a container. For example, Docker Swarm stand-alone is the only framework that allows to dynamically change resource limits without restarting containers. Such less restrictive API is a more flexible starting point for implementing a custom developed CO framework.

- The significant difference between Kubernetes and Mesos can be partially explained by the fact that Mesos by itself is not a complete CO framework as Mesos enables fine-grained sharing of resources across different CO frameworks such as Marathon, Aurora and DC/OS.
- The significant difference between Kubernetes and Marathon can be explained by the fact that very few new features have been added to Marathon since the start of DC/OS. After all, DC/OS is the extended Mesos+Marathon distribution that has also an enterprise edition.

An important conclusion is that there are no significant differences between the three most generic CO frameworks: Docker Swarm integrated mode, Kubernetes and DC/OS.

Table 5. For each functional (sub)-aspect, the number of common feature implementation strategies by each CO framework are shown and the framework(s) with the highest number is/are also shown.

Aspects	Sub-aspects	CO frameworks							FW(s) with Most Common Features
		Sa	Si	Ku	Me	Au	Ma	Dc	
cluster architecture and setup		**7.5**	**9.5**	**12.5**	**8**	**7**	**9**	**9.5**	**Ku**
	Configuration management approach	1	1	1	0	1	1	1	All but Me
	Architectural patterns	5	5	4.5	3	4	5	5	Sa/Si/Ma/Dc
	Installation methods and deployment tools	1.5	3.5	7	5	2	3	3.5	Ku
CO system customization		**4**	**4**	**6**	**5**	**4**	**4**	**5**	**Ku**
	Unified container runtime architecture	3	3	3	2	2	2	2	Sa/Si/Ku
	Framework design of orchestration engine	1	1	3	3	2	2	3	Ku/Me/Dc
Container networks		**8**	**16.5**	**14.5**	**11**	**5**	**12**	**15**	**Si**
	Services networking	3	7.5	6	4.5	2	5	7	Si
	Host ports conflict management	1	2	1	0.5	1	1	1	Si
	Plugin architecture for network services	3	3	2.5	3	0	3	3	Sa/Si/Me/Ma/Dc
	Service discovery and external access	1	4	5	3	2	3	4	Ku
Application configuration and deployment		**14**	**21**	**27**	**12.5**	**14**	**18.5**	**23.5**	**Ku**
	Supported workload types	2	4	6.5	1	3	4	5	Ku
	Persistent volumes	7	7	7.5	8.5	3	6.5	7.5	Me
	Reusable container configuration	3	4	5	3	3	4	4	Ku
	Service upgrades	2	6	8	0	5	4	7	Ku
Resource quota management		**0**	**1**	**4**	**2.5**	**2**	**1**	**2**	**Ku**
Container QoS Management		**8**	**11**	**13**	**8**	**11**	**9**	**9**	**Ku**
	Container CPU and mem allocation with support for over-subscription	4	5	5	3	2	2	2	Si/Ku
	Allocation of other resources	0	0	1	2	2	2	2	Me/Au/Ma/Dc
	Controlling scheduling behavior by means of placement constraints	3	3	3	1	3	3	3	All but Me
	Controlling preemptive scheduling and re-scheduling behavior	1	3	4	2	4	2	2	Ku/Au
Securing clusters		**2**	**7.5**	**7.5**	**5**	**5**	**2**	**7**	**Si**
	User identity and access management	1	2.5	3	3	3	2	3	Ku/Me/Au/Dc
	Cluster network security	1	5	4.5	2	2	0	4	Si
Securing containers		**3.5**	**2**	**7**	**3**	**0**	**2**	**2**	**Ku**
	Protection of sensitive data and proprietary software	0	2	2	2	0	2	2	All but Sa/Au
	Improved security isolation	3.5	0	5	1	0	0	0	Ku
App and cluster management		**9.5**	**14.5**	**18**	**16.5**	**12.5**	**13**	**20**	**Dc**
	Creation, management and inspection of cluster and applications	3	3	4	3	2.5	3	4	Ku/Dc
	Monitoring resource usage and health	1.5	2.5	4	4	3	3	4	Ku/Me/Dc
	Logging and debugging of CO framework and containers	2.5	2.5	3	2	1	1	3	Ku/Dc
	Cluster maintenance	1.5	3.5	3.5	4.5	3	4	5	Dc
	Multi-cloud deployments	1	3	3.5	3	3	2	4	Dc
Total # common feature implementation strategies		56.5	87	109.5	71.5	60.5	70.5	93	548.6

However, for 13 sub-aspects, a specific CO framework distinguishes itself by offering the most common features in that sub-aspect. In particular, Kubernetes, Docker Swarm integrated mode, DC/OS and Mesos are the most distinguishing frameworks:

- Kubernetes has the absolutely most features for 7 sub-aspects:

    1. Installation methods and deployment tools
    2. Service discovery and external access
    3. Supported workloads
    4. Reusable container configuration
    5. Service upgrades
    6. Resource quota management
    7. Improved security isolation

    For all 7 sub-aspects, the open-source distribution of Kubernetes supports all common features of these sub-aspects. As such Kubernetes is very generic with respect to these sub-aspects.
- Docker Swarm integrated mode has the most features for 3 sub-aspects:

    1. Services networking
    2. Host ports conflict management
    3. Cluster network security

    For the first two sub-aspects, Docker Swarm integrated mode offers support for all common features, while for the last sub-aspect, the open-source distribution of Docker Swarm integrated mode offers support for all common features except *authorization of CO agents on worker nodes*.
- DC/OS has the most features for 2-sub-aspects:

    1. Cluster maintenance
    2. Multi-cloud deployments

    For the first sub-aspect, DC/OS offers support for all common features of this sub-aspect by building upon Mesos and Marathon and providing detailed manual instructions for upgrading DC/OS. For the second sub-aspect, DC/OS offers support for all common features except *recovery from network partitions*.
- Mesos has the most features for 1 sub-aspect:

    1. Persistent volumes

    After all, Mesos offers support for both Docker volumes as well as CSI-based volumes.

There are furthermore tied observations between Docker Swarm, Kubernetes and DC/OS for 12 sub-aspects where these CO frameworks offer an equal number of features (see Figure 8 for a visual overview of the differences in genericity between these frameworks for the 27 sub-aspects).

RQ6b. Which functional sub-aspects are best supported by a CO framework in terms of common features ánd unique features?

Kubernetes clearly offers the highest number of unique features (see Figure 5). When adding up common and unique features, Kubernetes even supports the highest number of features for all 9 aspects (see Figure 9). We argue that it is fair to take into account the large number of unique features of Kubernetes when ranking CO frameworks with respect to genericity. After all, as already stated in Section 1, both Docker EE and DC/OS also offer support for Kubernetes as an alternative orchestrator. So the unique features are not a source of vendor lock-in.

Table 6 provides an overview of the total number of features per (sub)-aspect and per CO framework. In comparison to the quantitative analysis of the common features only (i.e., RQ6a), we find a more significant difference in ranking between the frameworks when re-applying the Friedman test for unreplicated designs (p-value = 1.729×10^{-10}). However, a pairwise comparison between CO frameworks using the Nemenyi test [56] did not show any significant differences between the

three major frameworks Docker Swarm integrated mode, Kubernetes and DC/OS. We do observe an additional, significant difference between Docker Swarm integrated mode and Docker Swarm stand-alone. This can be explained by the fact that the former introduces more unique features than the latter.

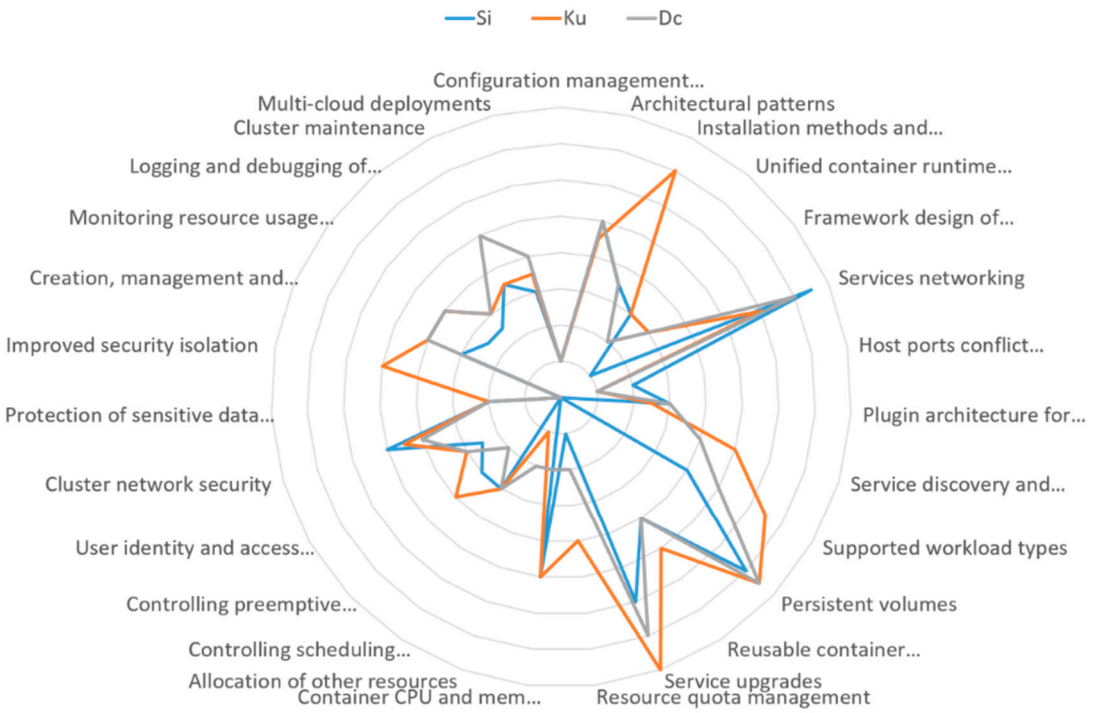

Figure 8. Radar chart of Docker Swarm integrated mode, Kubernetes and DC/OS to graphically present in which sub-aspects these CO frameworks support the highest number of common features.

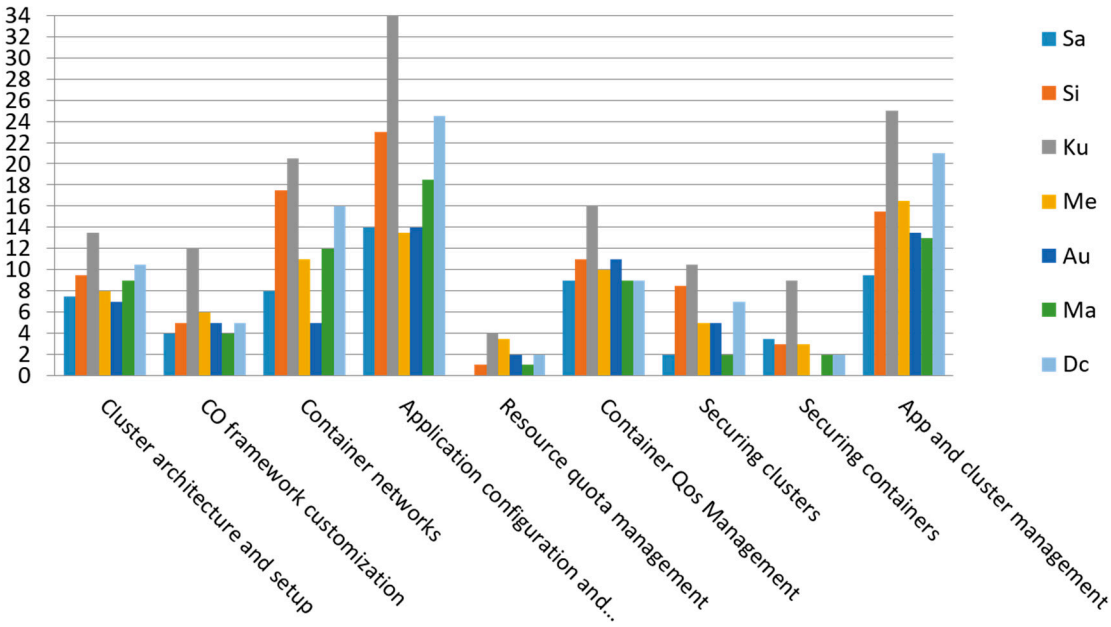

Figure 9. The total number of features supported by each of the CO frameworks is shown for the 9 aspects.

However, for 17 sub-aspects there is a specific CO framework that supports the highest number of common and unique features. We graphically present the top 3 CO frameworks using a radar chart (see Figure 10). Kubernetes offers the most features for 15 sub-aspects. Docker Swarm integrated

mode loses the 1st rank for the sub-aspect "cluster network security" to Kubernetes, but still offers the most features for the sub-aspects "services networking" and "host port conflict management". Mesos does not offer anymore the most features for the sub-aspect "persistent volumes", which is now more elaborately supported by Kubernetes. Instead, it offers the most features for the sub-aspect "allocation of other resources". DC/OS does not anymore offer the absolute most features in any sub-aspect.

Table 6. Overview of the number of total number of features (i.e., common + unique features) per (sub)-aspect and CO framework. The last column also shows which framework(s) support(s) the highest number of features per sub-aspect.

Aspects / Sub-aspects	CO Frameworks							FW(s) with Most Features
	Sa	Si	Ku	Me	Au	Ma	Dc	
Cluster architecture and setup	**7.5**	**9.5**	**13.5**	**8**	**7**	**9**	**10.5**	**Ku**
Configuration management approach	1	1	1	0	1	1	1	All but Me
Architectural patterns	5	5	4.5	3	4	5	5	Sa/Si/Ma/Dc
Installation methods and deployment tools	1.5	3.5	8	5	2	3	4.5	Ku
CO framework customization	**4**	**5**	**12**	**6**	**5**	**4**	**5**	**Ku**
Unified container runtime architecture	3	3	3	2	2	2	2	Sa/Si/Ku
Framework design of orchestration engine	1	2	9	4	3	2	3	Ku
Container networks	**8**	**17.5**	**20.5**	**11**	**5**	**12**	**16**	**Ku**
Services networking	3	8.5	6	4.5	2	5	8	Si
Host ports conflict management	1	2	1	0.5	1	1	1	Si
Plugin architecture for network services	3	3	2.5	3	0	3	3	Sa/Si/Me/Ma/Dc
Service discovery and external access	1	4	11	3	2	3	4	Ku
Application configuration and deployment	**14**	**23**	**34**	**13.5**	**14**	**18.5**	**24.5**	**Ku**
Supported workload types	2	4	8.5	1	3	4	5	Ku
Persistent volumes	7	7	11.5	9.5	3	6.5	8.5	Ku
Reusable container configuration	3	5	6	3	3	4	4	Ku
Service upgrades	2	7	8	0	5	4	7	Ku
Resource quota management	**0**	**1**	**4**	**3.5**	**2**	**1**	**2**	**Ku**
Container QoS Management	**9**	**11**	**16**	**10**	**11**	**9**	**9**	**Ku**
Container CPU and mem allocation with support for over-subscription	5	5	5	3	2	2	2	Sa/Si/Ku
Allocation of other resources	0	0	3	4	2	2	2	Me
Controlling scheduling behavior by means of placement constraints	3	3	3	1	3	3	3	All but Me
Controlling preemptive scheduling and re-scheduling behavior	1	3	5	2	4	2	2	Ku
Securing clusters	**2**	**8.5**	**10.5**	**5**	**5**	**2**	**7**	**Ku**
User identity and access management	1	2.5	4	3	3	2	3	Ku
Cluster network security	1	6	6.5	2	2	0	4	Si/Ku
Securing containers	**3.5**	**3**	**9**	**3**	**0**	**2**	**2**	**Ku**
Protection of sensitive data and proprietary software	0	2	2	2	0	2	2	Si/Ku/Me/Ma/Dc
Improved security isolation	3.5	1	7	1	0	0	0	Ku
App and cluster management	**9.5**	**15.5**	**25**	**16.5**	**13.5**	**13**	**21**	**Ku**
Creation, management and inspection	3	4	4	3	2.5	3	4	Si/Ku/Dc
Monitoring resource usage and health	1.5	2.5	5	4	4	3	5	Ku/Dc
Logging and debugging	2.5	2.5	4	2	1	1	3	Ku
Cluster maintenance	1.5	3.5	5.5	4.5	3	4	5	Ku
Multi-cloud deployments	1	3	6.5	3	3	2	4	Ku
Total number of feature implementation strategies	57.5	94	144.5	76.5	62.5	70.5	97	602.5

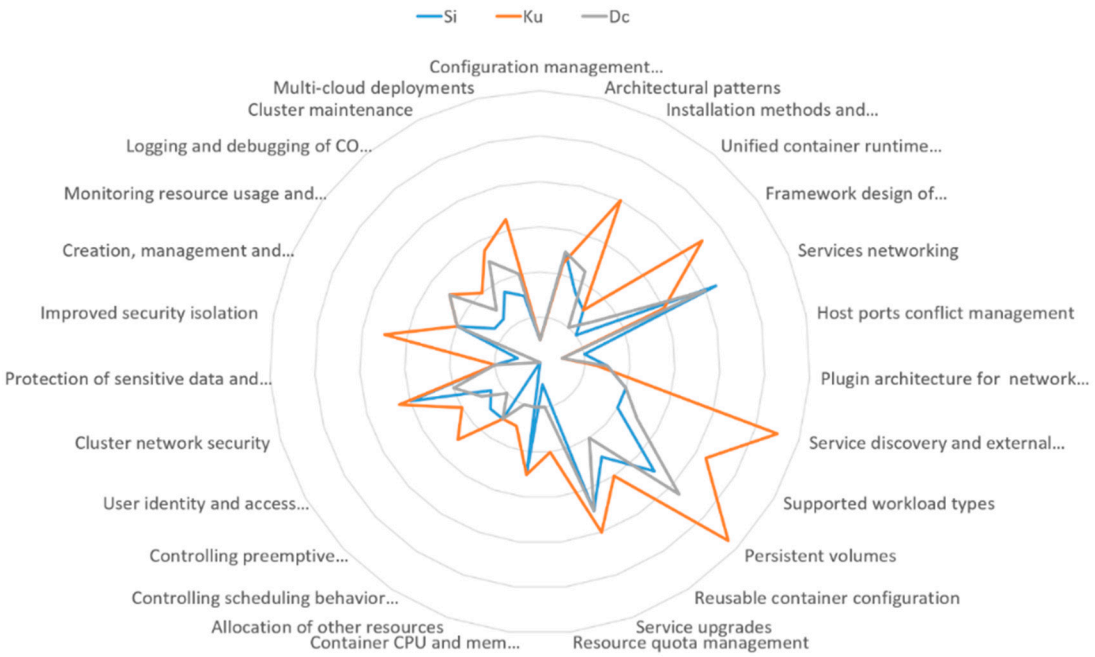

Figure 10. Radar chart of Docker Swarm integrated mode, Kubernetes and DC/OS for common + unique features.

7. Assessment of Maturity and Stability

This section answers research questions RQ7–RQ10:

RQ7. What is the maturity of a CO framework with respect to a common feature or a functional (sub)-aspect?

For each of the 9 functional aspects, we have created a table that maps each common feature to a timeline that orders CO frameworks according to the time they have released the alpha version of the common feature. The timelines also show when feature update and feature removal or deprecation events have occurred in order to assess the stability property. Because these timelines contain a lot of details we present in this article only a high-level summary as part of addressing RQ8, but the detailed timelines and their analysis are available in the associated technical report [31], Section 5.

RQ8. Which functional sub-aspects are mature enough to consider them as part of the stable foundation of the overall domain? Which CO frameworks have pioneered a particular sub-aspect?

Figure 11 shows an overall timeline that ranks sub-aspects with respect to their maturity. For each sub-aspect, the figure shows which CO framework has pioneered in consolidating the sub-aspect. We define a sub-aspect as being consolidated when a coherent subset of the common features of that sub-aspect has been established by the pioneering framework.

With respect to identifying those sub-aspects that are considered mature and well-understood, we are guided by the criteria that (i) the sub-aspect has been consolidated by the pioneering framework at least two traditional release cycles of 18 months [57] ago (these two release cycles are needed for letting other CO framework adopt and develop similar features.), (ii) the corresponding feature implementation strategies of the pioneering framework have at least reached beta-stage in the meantime and (iii) there are no deprecation or removal events of important features in the latest traditional release cycle.

This leads us to the observation that 15 out of 27 sub-aspects can be considered mature and well-understood (see green rectangle in Figure 11).

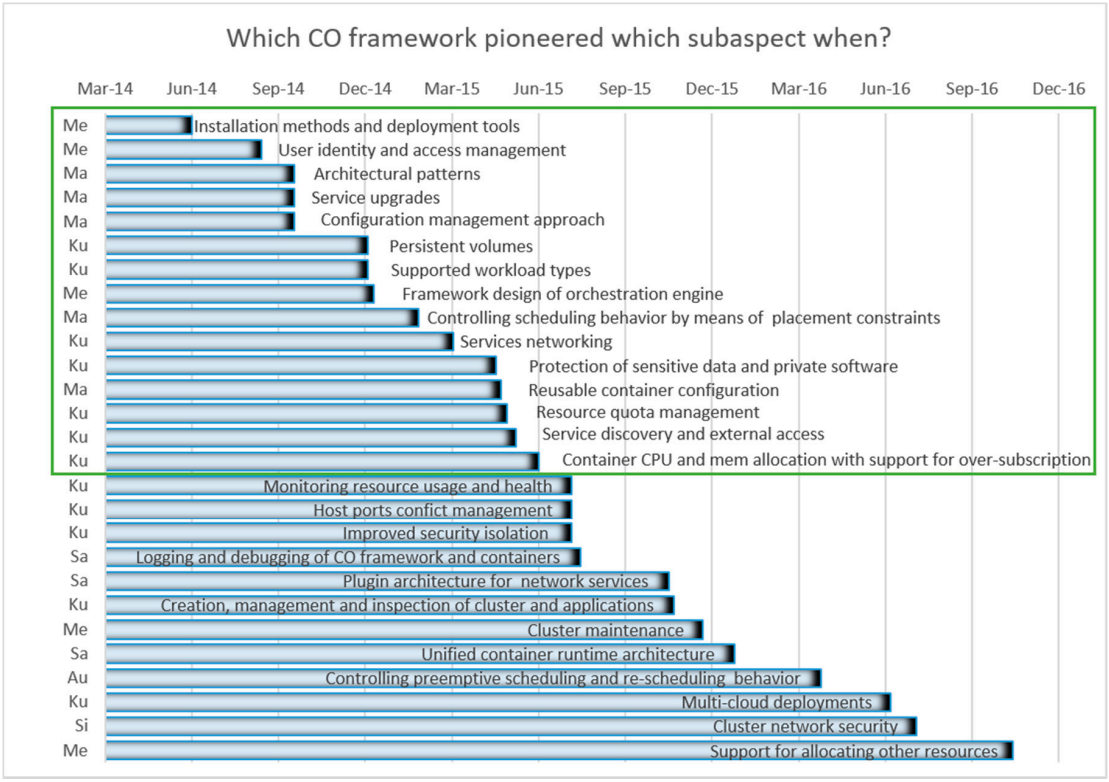

Figure 11. Timeline of when support for a sub-aspect have been consolidated by a CO framework.

Some sub-aspects that have been consolidated at least 36 months ago are not yet considered mature because they fail to meet one of the other two criteria:

- The sub-aspect "monitoring resource usage and health" is still in flux as Kubernetes' monitoring service (Heapster) has recently been completely replaced by two new monitoring services.
- Host port conflict management is expected to evolve due to the growing importance of supporting service networking in true host mode.
- Improved security isolation support by Kubernetes has not been substantially adopted by other orchestration frameworks; instead, security isolation is becoming a customizable property of container runtimes themselves.
- Logging support has remained very basic in all frameworks. Instead, many third-party companies have already offered commercial solutions for centralized log management.
- The network plugin architecture of Kubernetes has remained in alpha-stage for a very long time and is therefore expected to evolve. Docker's network plugin architecture is also expected to evolve because Docker EE supports Kubernetes as an alternative orchestrator.
- Inspection of cluster applications is expected to evolve towards a fully reflective interface so that it becomes possible to support application-specific instrumentation of different types of container orchestration functionality (see Section 8.3).
- Cluster maintenance, especially cluster upgrades, remains poorly automated.

Figure 11 also presents the creativity of CO frameworks by showing on the left of the timelines which CO frameworks have pioneered in consolidating a sub-aspect. Kubernetes has pioneered in 12 of the 27 sub-aspects. Mesos + Marathon in 10 of these sub-aspects, Docker Swarm in 4 sub-aspects, and Aurora in 1 sub-aspect. As such, the Kubernetes project has been the most creative in terms of pioneering new features despite being a younger project than Mesos, Marathon and Aurora.

RQ9. What are the relevant standardization initiatives and which CO frameworks align with these initiatives?

The stability of a CO framework software depends among other factors on its alignment with standardization initiatives. Increased openness to such standardization initiatives also creates more potential for researchers and entrepreneurs to contribute innovating technology that can be integrated in multiple CO frameworks.

In Section 4, we have identified several standardization initiatives towards common specifications to improve the plug-ability of various components including container runtimes, container networking services and storage drivers for external persistent volumes. Table 7 gives an overview of these standardization initiatives and which CO frameworks have aligned with these initiatives.

Table 7. Overview of existing standardization initiatives and their support.

Sub-aspects with features that relate to standardization initiatives	*Standardization initiatives*	Swarm stand-alone	Swarm integrated	Kubernetes	Mesos	Mesos + Aurora	Mesos + Marathon	DC/OS
		Sa	Si	Ku	Me	Au	Ma	Dc
Unified container runtime architecture	*Open Container Initiative (OCI) spec*	✓	✓	✓		future		
	Containerd container runtime architecture	✓	✓	✓				
Plugin architecture for network services	*Container Network Interface (CNI)*			✓	✓	✓	✓	✓
	Docker's libnetwork (aka CNM)	✓	✓		✓		✓	✓
Persistent volumes	*Docker's volume plugin system*	✓	✓		✓		✓	✓
	Common Storage Interface (CSI)			✓	✓			✓

The OCI specification for pluggable container runtimes has been accepted by Docker EE and Kubernetes, while Mesos has announced to add support for OCI soon.

Different standards for container networking (CNI, libnetwork) and persistent storage (CSI, Docker volumes) are not compatible across respectively Kubernetes and Docker Swarm. In opposition, DC/OS, provides encompassing support for all initiatives: DC/OS supports both CNI-based network plugins and Docker's libnetwork architecture. Moreover, it supports both Docker volumes as well as the CSI specification for persistent volumes. As such, with respect to networking and storage plugins, DC/OS and other Mesos-based frameworks are the most open frameworks. With respect to container runtimes, Kubernetes and Docker Swarm are the most open frameworks.

In general, we can state that DC/OS is the most interesting platform for prototyping novel techniques for container networking and persistent volumes because DC/OS' adherence to all relevant specifications in these two areas maximizes the potential to deploy these techniques in Docker Swarm and Kubernetes as well. Docker or Kubernetes are best fit for prototyping innovating container runtimes.

However, a widespread adoption of Kubernetes by cloud providers and cloud orchestration platforms has also occurred after the Cloud Native Computing Foundation has pushed Kubernetes as the de-facto standard in container orchestration and launched a certification programme for production-grade commercial Kubernetes offerings [23]. As a result, Docker volumes and Docker's libnetwork architecture, which are not supported by Kubernetes, may face the risk of not being further developed or halted. We estimate this risk to be low however because Docker offers its volume and networking architecture as separate building blocks that are relatively loosely coupled from its orchestrator layer. Therefore, we believe that Docker's volume and networking architectures will remain important alternatives to the existing standardization initiatives.

RQ10. What is the risk that common or unique features might become deprecated in the future?

If a particular CO framework halts the development of a particular feature or even deprecates the feature without offering a replacing feature update, then the risk arises that the development of company products or research prototypes that heavily rely on those features might also get compromised. In this section, we will assess that risk but for the future.

With respect to common features, we have studied the volatility of features in the past by counting the number of feature additions versus the number of feature deprecations (see Figure 1). Surprisingly, we have found very little volatility in terms of feature being deprecated without a replacing feature update. We recorded in total 626 feature additions; 48 out of these 626 additions comprised an update of an existing feature without deprecating the existing implementation strategy of the feature; finally, only 9 out of 626 feature additions comprised a feature update with deprecation or removal of the old implementation strategy of the feature. If we assume that the past is a good indicator for the future, the risk that a common feature will be deprecated by a CO framework without being replaced with an alternative new feature implementation strategy is less than 2%.

With respect to unique features, we assume that the risk may be higher. After all, if the team developing a specific unique feature faces even small problems, there is less incentive to resolve these problems in comparison to common features that are supported by other CO frameworks as well. This risk should be taken into account by research and development projects that consider relying on those unique features. The technical report [31] presents in Section 6 a detailed assessment of the risk of feature deprecation for the unique features of the three leading CO frameworks, Docker Swarm integrated mode, Kubernetes and DC/OS. In this article, we only present a summary of the most important findings for Kubernetes that offers a substantially higher number of unique features:

- If the performance overhead of *StatefulSets* for running database clusters cannot be resolved, DC/OS' approach to offer a user-friendly software development kit for generating custom scheduler frameworks for specific database may be the better approach.
- *Horizontal and vertical Pod autoscalers* are not fit to meet SLOs for complex stateful applications like databases. The generic design of these autoscalers will need to be sacrificed so that application managers can develop custom auto-scalers for particular workloads. As such, there is a substantial chance that the generic autoscaler will be replaced by different types of auto-scalers (see also Section 8.3).
- The development of the federation API for managing multiple Kubernetes clusters across cloud availability zones has been halted; instead, a new API is being planned [384]. Most likely the federation API will be replaced by a simplified API where some *existing federated instantiations* of Kubernetes API objects such as federated namespaces will be deprecated.

8. A Look-Ahead, Missing Functionality and Research Challenges

Section 8.1 provides an overview of likely future directions in the short term. Then, Section 8.2 provides an overview of what functional aspects are not yet covered by container orchestration frameworks, but could be envisioned to be engineered based on current scientific state-of-the-art. Finally, Section 8.3 presents a number of open research questions that we are trying to address in future work.

8.1. Further Evolutions in the Short Term

Likely areas for further evolution and innovation include improved system support for cluster network security and container security, performance isolation of GPU, disk and network resources and network plugin architectures:

- As stated above, Kubernetes is the only framework that offers rich support for container security isolation whereas Mesos and DC/OS offer very limited support and Docker EE uses another approach so that security isolation policies in Kubernetes are not easy to migrate to Docker. It is expected, however, that existing research [385] of how container security guarantees can be enforced using trusted computing architectures such as Intel SGX will influence the overall container security approach of CO frameworks.
- A weakness of Kubernetes is its limited support for performance isolation of GPU and disk resources and its lack of support for network isolation. Improved support for persistent volumes

as part of the Container Storage Interface (CSI) specification effort has been the main focus of the most recent releases of Kubernetes. Network isolation features for Kubernetes have also been subject to recent research [386,387]. It is expected that thus in the near future these features will be considerably improved.

- Finally, network plugin architectures themselves will change considerably due to recent research in the area of network function virtualization (NFV). Better support for high-performance networking without sacrificing automated management is currently also a main focus of current systems research [388]. It is expected that these innovations will also trigger improvements in virtual networking architectures for containers.

As Kubernetes offers an extensive set of unique features for managing container orchestrated services on public cloud providers, we believe these unique features are assets of Kubernetes that will be further developed to further strengthen the position of Kubernetes as the main CO framework for managing container clusters in public clouds.

Docker Swarm is the most light-weight framework in terms of complexity and memory footprint [389]. Moreover, it offers support for network protocols used in cellular network applications (see Table A1). As such, Docker Swarm could be further developed to be used in specific technology segments such as cellular networks, cyber-physical systems, and connected and autonomous vehicles.

Kubernetes and DC/OS are definitively two camps of opposite approaches with respect to their support for the management of database clusters. We believe that when high-performance database workloads must be targeted where database instances must run close to the physical data storage location in the data center, DC/OS' database services might be the preferred choice because they are installed natively without relying on containers and Mesos support for allocation and reservation of local disk resources is very mature. Performance enhancements for database workloads have already been added to Kubernetes and we expect that more features will be added to Kubernetes in the short-term.

8.2. Missing Sub-Aspects

8.2.1. Monitoring Dynamic Cloud Federations

Intercloud middleware [390] enables application managers to deploy their applications across multiple independent cloud platforms that are located in geographically different data centers. It also enables an application to more autonomously delegate parts of its application components to those cloud providers that offer suitable pricing models or SLA. Also, customers of applications may want to run parts of the application nearby or on their own trusted infrastructure in order to protect their data or because the data is too big to move.

While existing CO frameworks offer basic support for customizing the placement of applications across multiple cloud providers, they lack support for observing and dynamically exploiting the different VM types that are offered by a cloud provider in terms of pricing models and SLAs. For example, customer-facing latency-sensitive applications require VMs that can be leased for a longer period at a lower-cost, while non-critical batch workloads can acquire VMs that are only run when idle datacenter resources are available. The design of an autonomic monitoring service that is able to detect and annotate these different pricing models and SLAs in the cloud federation is thus lacking in current CO frameworks. It is neither possible to monitor for events about changes in pricing models and SLAs of cloud providers.

However, existing state-of-the-art [39,40] has already performed extensive research in this area. It is thus expected that existing multi-cloud support of CO frameworks can be extended with pricing-aware and SLA-aware selection of VM types.

8.2.2. Support for Application-Level Multi-Tenancy

SaaS providers continuously aim to optimize the cost-efficiency, scalability and trustworthiness of their offerings. Traditionally, these concerns have been addressed by a *shared-everything multi-tenant architecture* where multiple tenants are hosted by the same application instance.

Common requirements in multi-tenant SaaS applications are support for performance isolation between tenants and exploitation of hybrid clouds. Existing platform-level multi-tenancy support, as offered by existing CO frameworks in the form of resource quota management for different user groups and tenant-aware access control to API objects, is a great first start for accommodating these requirements but is not enough for accommodating multi-tenant SaaS applications. For example, it is possible to host multiple tenants in separate application containers, thus providing performance isolation and security isolation but at a higher operational cost. On the other hand, hosting multiple tenants in the same container is more cost-efficient but CO frameworks cannot offer security isolation and protection against aggressive tenants who flood the application container with a high request rate.

A key missing component is, therefore, a network admission component for throttling incoming network traffic from aggressive tenants before this traffic can hit the Layer 4 load balancing tier of container orchestration frameworks. Mesos' framework rate limiting [375] and so-called service meshes such as Istio [391] for Kubernetes already exist for protecting the Master API of the CO framework against aggressive tenants, but this should be extended for network traffic to any application deployed. Providing a single service mesh that can host different tenants with some level of isolation and security between the tenants is however planned future work in Istio [392].

8.3. Research Challenges

8.3.1. Accurate Estimation of Compute Resources

A first open research question is achieving both (i) cost-effective use of node resources by means of co-locating workloads and (ii) meeting tail latencies of service-level objectives (SLOs) (Tail latencies refer to 99th percentile of the response times of a service).

Competitive Software-as-a-Service (SaaS) markets drive application providers to offer high-quality services while reducing operating cost. Today's SaaS applications are typically developed as multi-tiered applications that can be easily deployed using container orchestration frameworks. By taking advantage of fine-grained resource control features offered by these frameworks, application managers already have the capability of maximizing their resource utilization. However, mapping SLOs into resource allocations for containers is a difficult task. Especially with the trend from monolithic business tiers to micro-service architectures, it is a difficult task for application managers to allocate optimal resources quantities for the different micro-services [393].

Therefore, an automated resource optimization approach is needed in order to assist the application managers and reduce human errors. However, existing techniques for resource optimization often require the specification of an accurate performance model representing the application [394,395]. Moreover, SaaS providers might offer custom features and custom SLOs to different tenants, which complicates this modelling task. Therefore, what is needed is a generic, black-box optimization solution that does not require any model of the application or another type of domain expertise.

8.3.2. Performance-Driven Instrumentations of CO Framework Functionality

Verma et al. [42] report that the implementation of the cgroups mechanism requires substantial tuning of the standard Linux CPU scheduler in order to achieve both low latency and high utilization for the typical latency-sensitive, user-facing workloads at Google.

Recent research indicates that the problem of meeting tail latencies also needs dedicated performance engineering strategies that are specifically designed for specific application families.

Recent contributions in this space include tail-latency aware caching for user-facing web services [396] and auto-tuning threading for on-line data-intensive micro-services [397].

Clearly, existing QoS management features of CO frameworks such as setting appropriate compute resources limits and reserving CPUs for specific containers are too generic in that they cannot accommodate deep customization of orchestration functionality for specific families of applications.

Research is therefore needed how to incept and design reflective APIs for application-specific instrumentation of particular container orchestration functionalities. An evolution towards such instrumentation already has happened at the level of container runtimes (e.g., crictl [398] and Mesos containerizer isolators [399]) but is expected to extend towards orchestration framework functionality as well. Examples of relevant instrumentation scenarios include customizations to service load balancing strategies and performance-sensitive enactment customizations of rolling upgrades. But also non-performance requirements, such as global checkpointing for recovering from failures [400], require instrumentations of the service IP addresses so that these can be persisted and recovered again.

8.3.3. Elastic SLO Management

With respect to auto-scaling concepts, Kubernetes provides besides the Horizontal Pod Autoscaler [163] (HPA) also the Vertical Pod Autoscaler [353] (VPA).

With the increasing focus of recent Kubernetes releases to improve QoS management, the question arises if these auto-scaling concepts can also be configured to meet service-level objectives (SLOs). However, we have demonstrated in previous research that the Horizontal Pod Autoscaler is too simplistic for meeting service level objectives (SLOs) of database clusters. We handled this problem by developing a tailored auto-scaler component that is customized to the type of database cluster [49]. Unless the HPA for StatefulSets can be tailored via Kubernetes' *annotations* or *modular interceptors*, the HPA for StatefulSets will need to be redeveloped by relying on a framework or library by means of which custom auto-scaling policies and complex event monitoring policies can be specified and enforced.

The Kubernetes' VPA concept is promising but there is one big disadvantage with respect to SLO compliance: adjusting resource allocation policies of Pods requires killing these Pods and waiting till the scheduler assigns a new Pod with the adjusted allocation policies. Obviously, this operation needs to be performed at run-time without restarting containers in order to avoid temporary performance degradation with SLO violations. Ironically, although run-time adjustment of container resource allocation policies is by default supported in Docker engine, they are not supported by any CO framework except Docker stand-alone. Indeed recent research presents a middleware for vertical scaling of containers that is implemented on top of Docker engine exactly because the presented middleware requires adjusting resource allocation policies without restarting containers [401].

Finally, cost-effective auto-scaling to meet SLOs has been shown a sub-linear resource scaling problem [402], especially when co-locating tenants that are being offered different types of SLOs [403]. However existing HPAs of Kubernetes only support linear scaling, therefore leading to either SLO violations or allocating too many resources. A work-around that simulates sub-linear scaling is to combine HPA and VPA, but it would be much better to support managing container replicas with heterogeneous resource allocation policies.

In summary, existing auto-scalers of Kubernetes are not ready for managing performance SLOs. This lack is also the main reason why we have not grouped these auto-scaling features under the "Container QoS Management" aspect.

9. Threats to Validity and Limitations of the Study

In essence, we present in this article a descriptive study based on expert reviews and expert assessments and therefore the main results are qualitative. All quantitative results are based on the identified features in the qualitative part of the study, which is inherently subjective to some extent.

We have thus not used variations of dependent and independent variables with different subject groups. Neither have we used automated metrics such as NLP-based processing of documentation, or amount of code/documentation.

As consequence, a large part of the standard threats to internal and external validity in experiment design are not relevant to this study. As a reminder, threats to internal validity compromise our confidence in stating that the found differences between CO frameworks are correct. Threats to external validity compromise our confidence in stating that the study's results are applicable to other CO frameworks.

As we don't make claims about other CO frameworks, only the following internal validity threats remain relevant:

- Selection bias, i.e., the decision what CO framework to select and the selection of the different features and the overarching (sub)-aspects may be determined subjectively. Thus, we may have missed features or interpreted feature implementation strategies inappropriately.
- Experimenter bias, i.e., unconscious preferences for certain CO frameworks that influence interpretation of documentation; e.g., whether a feature is partially or fully supported by a framework.

9.1. Selection Bias

We have tried to manage selection bias in our research method by means of three complementary approaches that have been explained in detail in Section 3. Firstly, we have applied a systematic approach and used existing methods if possible; for example, we have applied commonality and variability analysis in feature modelling to find common features (see Section 3.1.2) and we have applied card sorting to group features in usable aspects (see Section 3.1.3).

Secondly, we improved the accuracy of the description of the features and feature implementation strategies by means of an iterative approach. For example, we have first performed a pair-wise comparison of titles of documentation pages and thereafter a detailed review of the documentation pages in full detail. Then, we have asked customers and platform developers to review different versions of this article with respect to the question whether the set of identified features and their comparison makes sense and is complete (see Section 3.5).

Thirdly, we have continuously elaborated our practical experience of CO frameworks by not only testing specific features but also conducting performance evaluation research [49,50]. This practical experience helps to make better interpretations of documentation.

9.2. Experimenter Bias

It has been challenging to manage experimenter bias because container technology is currently at its peak of inflated expectations according to the Gartner hype cycle, has evolved quickly in the past, and Kubernetes has been adopted by Docker EE and DC/OS and all major public cloud providers.

To stay objective in the mid of such inflated expectations, we have consciously scoped the study to research questions with respect to software qualities that can be objectively measured using simple arithmetic: (i) genericity (in terms of number of supported features, (ii) maturity (i.e., mapping features to development history on GitHub) and (iii) stability (i.e., number of updated or deprecated features). To find evidence for overall significant differences between the CO frameworks with respect to genericity, we have used the Friedman and Nemenyi tests due to their effectiveness in un-replicated experimental designs for checking overall ranking of multiple systems with respect to different metrics [56]; in our research, metrics correspond with the 27 sub-aspects and systems with the CO frameworks.

9.3. Limitations of the Study

Besides the above threats to internal validity in experimental design, the study has the following limitations:

- We have only studied the documentation of CO frameworks, not the actual code. We have not used any automated methods for mining features/aspects from code. As such features that can only be extracted from code are not covered in this study.
- Any claims about performance or scalability of a certain CO framework's feature implementation strategy are based on actual performance evaluation of Kubernetes and Docker Swarm integrated mode in the context of the aforementioned publications [49,50]. However, projections of these claims towards performance and scalability of similar feature implementation strategies in Mesos-based frameworks are speculative.
- The study does not provide findings about the robustness of the CO frameworks such as or the ratio of bugs per line of code, or the number of bug reports per user.

10. Lessons Learned

We organize the main conclusions from this study according to the three aforementioned software qualities, and thereafter we summarize the highlights for each of the frameworks.

10.1. Genericity

- The ratio of common features over unique features is relatively large and most common features are supported by at least 50% of the CO frameworks. Such a high ratio of common features allows for direct comparison of the CO frameworks with respect to non-functional requirements such as scalability and performance of feature implementation strategies.
- Features in the sub-aspects "improved security isolation" and "allocation of other resources" are only supported by two or three CO frameworks
    - Although Kubernetes consolidated a full feature set for container isolation policies almost 36 months ago, there is little uptake of these features by the other CO frameworks.
    - Mesos-based support for allocating GPU and disk resources to co-located containers is only marginally supported by Kubernetes and not supported by Docker Swarm.
- Kubernetes offers the highest number of common features and the highest number of unique features. When adding up both common and unique features, Kubernetes even offers the highest number of features for all 9 aspects and it offers the highest number of features for 15 sub-aspects.
- Significant differences in genericity with Docker EE and DC/OS have however not been found. After all, when taking into account only common features, Kubernetes offers the absolute highest number of common features for 7 sub-aspects, whereas Docker Swarm integrated mode offers the highest number of common features for the sub-aspects "services networking", "host port conflict management" and "cluster network security". Mesos offers the most common features for the sub-aspect "persistent volumes" and DC/OS offers the most common features of the sub-aspects "cluster maintenance" and "multi-cloud deployments".
- In the sub-aspects "services networking" and "host port conflict management", Docker Swarm integrated mode and DC/OS offer support for the features *host mode services networking*, *stable DNS name for services* and *dynamic allocation of host ports*. We have found that the other approaches to services networking such as routing meshes and virtual IP networks introduce a substantial performance overhead in comparison to running Docker containers in host mode. As such, a host mode service networking approach with appropriate host port conflict management is a viable alternative for high-performance applications.
- For some of these sub-aspects, there remain differences between whether a particular set of features is offered by the open-source distribution or commercial version of these frameworks:

- Architectural patterns. The open-source distributions of Docker Swarm and DC/OS all support *automated setup of highly available clusters*, where Kubernetes only provides support for this feature in particular commercial versions.
- User identity and access management. Kubernetes offers the most extensive support for authentication and authorization of cluster administrators and application managers because the open-source distributions of these frameworks offer support for *tenant-aware access control lists*. In opposition, only the commercial versions of Docker Swarm and DC/OS offer support for this feature.
- Creation, management and inspection of cluster and applications. The open-source distributions of Kubernetes and DC/OS offer the most extensive *command-line interfaces* and *web-based user interfaces* with support for common features such as *labels for organizing API objects* and *visual inspection of resource usage graphs*. The commercial version of Docker Swarm also includes a web-based UI with the same set of features, though.
- Logging and debugging of containers and CO frameworks. The open-source distribution of Kubernetes and DC/OS offer support for *integrating existing log aggregation systems*. In opposition, only the commercial version of Docker Swarm supports this feature.
- Multi-cloud support. Docker Swarm and all Mesos-based systems have invested most of their effort in building extensive support for running a single container cluster in high availability mode where multiple masters are spread across different cloud availability zones. Support for such an automated HA cluster across multiple availability zones is not supported by the open-source contribution of Kubernetes; it is only supported by the commercial Kubernetes-as-a-Service offerings on top of AWS and Google cloud.

10.2. Maturity

- The 15 sub-aspects identified by the green rectangle in Figure 11 shape a mature foundation for the overall technology domain as these sub-aspects are well-understood by now and little feature deprecations have been found in these sub-aspects.
- Figure 11 further indicates that Kubernetes is the most creative project in terms of pioneering common features despite being a younger project than Mesos, Aurora and Marathon.

10.3. Stability

- Mesos is the most interesting platform for prototyping novel techniques for (i) container networking and (ii) persistent volumes because Mesos' adherence to all relevant standardization initiatives in these two areas maximizes the potential to deploy these techniques in Docker Swarm and Kubernetes as well. Docker or Kubernetes are best fit for prototyping innovating techniques for container runtimes.
- The overall rate of feature deprecations among common features in the past is about 2% of the total number of feature updates (i.e., feature additions, feature replacements, and feature deprecations).
- Only one unique feature of Kubernetes, *federated instantiations of the Kubernetes API objects*, has been halted and will probably be deprecated without a replacing feature update.

10.4. Main Insights with Respect to Docker Swarm

Although Docker Swarm is the youngest and also least generic framework among the three leading CO frameworks, Docker Swarm has clearly contributed an innovative services networking approach and networking plugin architecture.

Docker has actually separated services networking support from Docker Swarm. As such we believe that Docker's networking architecture is here to stay. Docker has also recently released an enterprise edition with support for deploying and managing Kubernetes clusters next to Swarm clusters. While the current release does not show any strong integration between Docker

and Kubernetes, support for Docker's networking architecture in Kubernetes is a likely future feature request.

10.5. Main Insights with Respect to Kubernetes

Kubernetes is the most generic orchestration framework for 7 out of 27 of sub-aspects. Yet, in many sub-aspects the absolute differences in number of supported features is small with respect to the two other leading frameworks Docker EE and DC/OS.

Kubernetes has also the most unique features. This may be a higher source of vendor lock-in on the one hand, but mainly constitutes a competitive edge. Our analysis of genericity has shown that many unique features of Kubernetes are much stronger a source for increased genericity than a source of vendor lock-in. When taking into account the total of common and unique features of Kubernetes, it counts the highest number of features of 15 sub-aspects.

Kubernetes is also the most mature container orchestration framework as it has pioneered 12 out of the 27 sub-aspects.

Kubernetes is in particular unique by its support for integrating with public cloud platform's load-balancing tier and offering a wide range of external service discovery options. As a result, a large number of public cloud providers have offered a hosted solution or even a Kubernetes-as-a-Service offering.

A weakness of the open source distribution of Kubernetes is that it does not offer support for automated installation of a highly-available cluster with multiple master nodes. Kubernetes is also less scalable than Mesos as it is currently limited to supporting not more than 5000 nodes in a single cluster [404].

10.6. Main Insights with Respect to Mesos and DC/OS

DC/OS, an extended Mesos+Marathon distribution is the second most generic framework. Mesos+Marathon has pioneered also 10 out of the 27 aspects. A strength of Mesos is that it allows fine-grained sharing of cluster resources across multiple scheduler frameworks, which include not only CO frameworks but also non-CO frameworks like Hadoop, Kafka and NoSQL databases. This decentralized scheduling architecture is the reason why Mesos-based clusters can support large-scale clusters till the size of 50,000 nodes [41]. Mesos or DC/OS may also be a viable alternative to companies who seek to set up a highly available cluster in a private cloud with the broadest range of possibilities to integrate container-based applications with non-container-based applications. After all, DC/OS offers support for load balancing non-container orchestrated workloads such as databases or high-performance computing applications.

10.7. Main Insights with Respect to Docker Swarm Alone and Apache Aurora

Docker Swarm stand-alone and Apache Aurora are relatively small CO frameworks that do differ significantly in terms of genericity from DC/OS and Kubernetes. Indeed, Aurora is specifically designed for running long-running jobs and cron jobs, while Docker Swarm stand-alone is also a more simplified framework with substantial less automated management.

We only recommend Docker Swarm stand-alone as a possible starting point for developing one's own CO framework. This is a relevant direction because 28% of surveyed users in the most recent OpenStack survey [4], responded that they have built their own CO framework instead of using existing CO frameworks (see also Figure 4). We make such recommendation because the API of Docker Swarm stand-alone is the least restrictive in terms of the range of offered options for common commands such as creating, updating and stopping a container. For example, Docker Swarm stand-alone is the only framework that allows to dynamically change resource limits without restarting containers. Such less restrictive API is a more flexible starting point for implementing a custom developed CO framework.

11. Conclusions

In this article, we have presented a descriptive feature comparison study of the three most prominent orchestration frameworks: Docker Swarm, Kubernetes and Mesos that can be combined with Marathon, Aurora or DC/OS. The goals of this study were three-fold: (i) identifying the common and unique features of all frameworks, (ii) comparing these frameworks qualitatively ánd quantitatively with respect to genericity in terms of supported features, and (iii) investigating the maturity and stability of the frameworks as well as the pioneering nature of each framework by studying the historical evolution of the frameworks on GitHub.

We have identified 124 common features and 54 unique features that we divided into a taxonomy of 9 functional aspects and 27 functional sub-aspects. Although, Kubernetes supports the highest number of accumulated common and unique features for all 9 functional aspects, no evidence has been found for significant differences in genericity with the other two leading frameworks, i.e., Docker Swarm and the Mesos-based DC/OS. Fifteen out of 27 sub-aspects have been identified as mature and stable. These are pioneered in descending order by Kubernetes, Mesos and Marathon. Finally, less than 2% of common feature implementation strategies have been deprecated without introducing a replacing implementation strategy. We conclude therefore that a broad, mature and stable foundation underpins the studied container orchestration frameworks.

The main differentiating characteristics between the three main types of vendors (Docker EE, CNCF-certified Kubernetes solutions, and the Mesos-based DC/OS) are as follows. The large number of unique features of Kubernetes, especially those for managing clusters in public clouds, is a strong asset of Kubernetes without creating a risk of vendor lock-in. After all, not only all major public cloud providers, but also Docker EE and DC/OS already offer support for Kubernetes. DC/OS is, like any other Mesos-based framework, the best choice for large-scale cluster deployments from 5000 till 50,000 nodes due to Mesos' inherent decentralized scheduler architecture. Finally, Docker Swarm stand-alone is expected to be used and customized for specific technology segments, such as the domain of Internet of Things, due to its low memory footprint, its support for co-existing virtual networks and its support for run-time updates of container images without the need to restart containers. The future of Docker Swarm integrated mode depends on whether its extensive networking features can be integrated with co-located Kubernetes clusters.

Likely areas for further evolution and innovation include system support for improved cluster security and container security, performance isolation of GPU, disk and network resources, and network plugin architectures. Two currently missing functional sub-aspects are price- and SLA-aware selection of VMs for setting up clusters in hybrid or federated clouds and support for controlling application-level tenancy so that SaaS providers can fully control the trade-off between improved resource utilization and performance isolation between tenants. Open research challenges include (i) correct estimation of required resources for individual containers and container replica levels to meet a certain performance or scalability requirement, especially in micro-service based applications, (ii) support for performance-sensitive instrumentations of CO framework functionality and (iii) support for sub-linear auto-scaling in order to implement cost-effective elastic SLO management that keeps the ratio of over-provisioned resources within a certain tolerance level.

Author Contributions: Conceptualization, E.T. and D.V.L.; Data curation, E.T.; Funding acquisition, B.L. and W.J.; Investigation, E.T.; Methodology, E.T.; Project administration, B.L.; Software, E.T.; Supervision, B.L. and W.J.; Validation, E.T. and D.V.L.; Visualization, E.T.; Writing—original draft, E.T.; Writing—review & editing, E.T., D.V.L. and D.P.

Funding: This research was funded by the Agency for Innovation and Entrepreneurship IWT, grant DeCoMAdS, grant number 179K2315, and the Research Fund KU Leuven.

Acknowledgments: We thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of this article. We thank Bert Robben for his feedback and reviews of drafts of this article. We thank the developers of the CO frameworks, especially the technical writers for the excellent documentation. Finally, we thank GitHub for offering the documentation of the CO frameworks in versioned format.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A

Table A1. Unique features of Docker Swarm, Kubernetes, Mesos, Aurora, Marathon and DC/OS.

Column Legend:

- **Sa**: Docker Swarm stand-alone
- **Si**: Docker Swarm integrated mode
- **Ku**: Kubernetes
- **Me**: Mesos
- **Au**: Mesos+Aurora
- **Ma**: Mesos+Marathon
- **Dc**: DC/OS

***Aspects* and Sub-Aspects**	**Container Orchestration Frameworks**						
Cluster architecture and setup	Sa	Si	Ku	Me	Au	Ma	Dc
Configuration management approach							
Architectural patterns							
Installation methods and tools for setting up a cluster			Kubernetes-as-a				GUI-based installer
CO framework customization	Sa	Si	Ku	Me	Au	Ma	Dc
Unified container runtime architecture							
Framework design of orchestration engine		install plugins as global Swarm services	cloud-provider plugin custom API objects aggregation of additional APIs annotations to API objects discovery of a node's hardware features dynamic kubelet reconfig	resource provider abstraction to customize how Mesos agent synchronizes with the Mesos master about available resources and operations on those resources	custom worker agent software		
Container networking	Sa	Si	Ku	Me	Au	Ma	Dc
Services networking		SCTP protocol support					load balancing of non-containerized services
Host ports conflict management							
Plugin architecture for network services							

Table A1. *Cont.*

***Aspects* and Sub-Aspects**	**Container Orchestration Frameworks**						
Service discovery and external access			exposing service via LB of cloud provider synchronize services with external DNS providers hide Pod's virtual IP behind Node IP override DNS lookup with custom /etc/hosts entries in Pod override name server with custom /etc/resolv in Pod install another DNS server in cluster				
App configuration and deployment	Sa	Si	Ku	Me	Au	Ma	*Dc*
Supported workload types			Initial. of containers vertical pod auto-scaler				
Persistent volumes			deploying and managing stateful services raw block volumes dynamically grow volume size dynamic maximum volume count	local volume can be shared between tasks from different frameworks			tools and libraries for integration with and deployment of stateful services
Reusable container configuration		system init inside a container	injection of configs at Pod creation time				
Service upgrades		customizing the rollback of a service					
	Sa	Si	Ku	Me	Au	Ma	Dc
Resource quota management				request rate limiting of frameworks			
Container QoS management	Sa	Si	Ku	Me	Au	Ma	Dc
Container CPU and memory allocation with support for oversubscription	updating resource policies without restarting the container						
Allocation of other resources			define custom node resources of random kind scheduling of huge pages	network isolation for routing mesh networks network isolation between virtual networks			

Table A1. *Cont.*

Aspects and Sub-Aspects	Container Orchestration Frameworks						
Controlling scheduling behavior							
Controlling preemptive scheduling and re-scheduling			cpu-cache affinity policies				
Securing clusters	Sa	Si	Ku	Me	Au	Ma	Dc
User identity and access management			audit of master API requests				
Cluster network security		encryption of master/ manager logs	access control for the kubelet network policies for Pods				
Securing containers	Sa	Si	Ku	Me	Au	Ma	Dc
Protection of sensitive data and proprietary software							
Improved security isolation		customize service isolation mode in Windows	run-time verification of system-wide Pod security policies configuring kernel parameters at run-time				
App and cluster management	Sa	Si	Ku	Me	Au	Ma	Dc
Creation, management and inspection of cluster and applications		command-line auto-complete					
Monitoring resource usage and health			auto-scaling of cluster		SLA metrics		custom health checks
Logging and debugging of CO framework and containers			debug running Pod from local work station				
Cluster maintenance			control the number of Pod disruptions automated upgrade of Google Kubernetes Engine				
Multi-cloud support			API for using externally managed services federated API objects discovery of the closest healthy service shard				

References

1. Xavier, M.G.; de Oliveira, I.C.; Rossi, F.D.; Passos, R.D.D.; Matteussi, K.J.; de Rose, C.A.F. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. In Proceedings of the 2015 23rd Euromicro International Conference Parallel, Distributed, Network-Based Processing, Turku, Finland, 4–6 March 2015; pp. 253–260.
2. Truyen, E.; van Landuyt, D.; Reniers, V.; Rafique, A.; Lagaisse, B.; Joosen, W. Towards a container-based architecture for multi-tenant SaaS applications. In Proceedings of the ARM 2016 Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware, Trento, Italy, 12–16 December 2016.
3. Kratzke, N. A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms. *Open J. Mob. Comput. Cloud Comput.* **2014**, *1*, 17–30.
4. OpenStack. User Survey. April 2016. Available online: https://www.openstack.org/assets/survey/April-2016-User-Survey-Report.pdf (accessed on 1 March 2019).
5. Openstack. User Survey. October 2016. Available online: https://www.openstack.org/assets/survey/October2016SurveyReport.pdf (accessed on 27 October 2016).
6. OpenStack. User Survey—A Snapshot of the OpenStack Users' Attitudes and Deployments. 2017. Available online: https://www.openstack.org/assets/survey/OpenStack-User-Survey-Nov17.pdf (accessed on 1 March 2019).

7. Mesosphere. mesos/docker-containerizer.md at 0.20.0·apache/mesos. Available online: https://github.com/apache/mesos/blob/0.20.0/docs/docker-containerizer.md (accessed on 9 November 2018).
8. Pieter Noordhuis. Kubernetes. Available online: https://github.com/kubernetes/kubernetes/blob/release-0.4/README.md (accessed on 9 November 2018).
9. kubernetes/networking.md at v0.6.0 · kubernetes/kubernetes. Available online: https://github.com/kubernetes/kubernetes/blob/v0.6.0/docs/networking.md (accessed on 9 November 2018).
10. kubernetes/pods.md at release-0.4 · kubernetes/kubernetes. Available online: https://github.com/kubernetes/kubernetes/blob/release-0.4/docs/pods.md (accessed on 9 November 2018).
11. kubernetes/volumes.md at v0.6.0 · kubernetes/kubernetes. Available online: https://github.com/kubernetes/kubernetes/blob/v0.6.0/docs/volumes.md (accessed on 9 November 2018).
12. Docker Inc. Manage Data in Containers. Available online: https://docs.docker.com/v1.10/engine/userguide/containers/dockervolumes/ (accessed on 9 November 2018).
13. Mesosphere. mesos/docker-volume.md at 1.0.0 · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.0.0/docs/docker-volume.md (accessed on 9 November 2018).
14. Mesosphere. mesos/networking-for-mesos-managed-containers.md at 0.25.0 · apache/mesos. Available online: https://github.com/apache/mesos/blob/0.25.0/docs/networking-for-mesos-managed-containers.md (accessed on 9 November 2018).
15. Docker Inc. swarm/networking.md at v1.0.0 · docker/swarm. Available online: https://github.com/docker/swarm/blob/v1.0.0/docs/networking.md (accessed on 9 November 2018).
16. ClusterHQ. ClusterHQ/Flocker: Container Data Volume Manager for Your Dockerized Application. Available online: https://github.com/ClusterHQ/flocker/ (accessed on 9 November 2018).
17. Lardinois, F. ClusterHQ Raises $12M Series A Round to Expand Its Container Data Management Service. techcrunch.com. 2015. Available online: https://techcrunch.com/2015/02/05/clusterhq-raises-12m-series-a-round-to-help-developers-run-databases-in-docker-containers/ (accessed on 27 March 2018).
18. ClusterHQ. Flocker Integrations. Available online: https://flocker.readthedocs.io/en/latest/ (accessed on 9 November 2018).
19. Lardinois, F. ClusterHQ, an Early Player in the Container Ecosystem, Calls It Quits. techcrunch.com. 2016. Available online: https://techcrunch.com/2016/12/22/clusterhq-hits-the-deadpool/ (accessed on 27 March 2018).
20. Containerd—An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. Available online: https://containerd.io/ (accessed on 9 November 2018).
21. Open Container Initiative. Available online: https://github.com/opencontainers/ (accessed on 9 November 2018).
22. General Availability of Containerd 1.0 is Here! The Cloud Native Computing Foundation. 2017. Available online: https://www.cncf.io/blog/2017/12/05/general-availability-containerd-1-0/ (accessed on 27 March 2018).
23. Cloud Native Computing Foundation Launches Certified Kubernetes Program with 32 Conformant Distributions and Platforms. The Cloud Native Computing Foundation. 2017. Available online: https://www.cncf.io/announcement/2017/11/13/cloud-native-computing-foundation-launches-certified-kubernetes-program-32-conformant-distributions-platforms/ (accessed on 27 March 2018).
24. OpenStack. OPENSTACK USER SURVEY: A Snapshot of OpenStack Users' Attitudes and Deployments; Openstack.org. 2015. Available online: https://www.openstack.org/assets/survey/Public-User-Survey-Report.pdf (accessed on 1 March 2019).
25. GitHub. The State of the Octoverse 2017—Ten Most-Discussed Repositories. 2018. Available online: https://octoverse.github.com/2017/ (accessed on 1 March 2019).
26. Mesosphere. Kubernetes—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/services/kubernetes/ (accessed on 9 November 2018).
27. Docker Inc. Run Swarm and Kubernetes Interchangeably | Docker. Available online: https://www.docker.com/products/orchestration (accessed on 9 November 2018).
28. Amazon Web Services (AWS). Amazon EKS—Managed Kubernetes Service. Available online: https://aws.amazon.com/eks/ (accessed on 9 November 2018).
29. Kratzke, N. A Brief History of Cloud Application Architectures. *Appl. Sci.* **2018**, *8*, 1368. [CrossRef]

30. Truyen, E.; van Landuyt, D. Structured Feature Comparison between Container Orchestration Frameworks. 2018. Available online: https://zenodo.org/record/1494190#.XDh2ls17lPY (accessed on 11 January 2019).
31. Truyen, E.; van Landuyt, D.; Preuveneers, D.; Lagaisse, B.; Joosen, W. A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks. 2019. Available online: https://doi.org/10.5281/zenodo.2547979 (accessed on 11 January 2019).
32. Soltesz, S.; Soltesz, S.; Pötzl, H.; Pötzl, H.; Fiuczynski, M.E.; Fiuczynski, M.E.; Bavier, A.; Bavier, A.; Peterson, L.; Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.* **2007**, *41*, 275–287. [CrossRef]
33. Xavier, M.G.; Neves, M.V.; Rossi, F.D.; Ferreto, T.C.; Lange, T.; de Rose, C.F. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In Proceedings of the 2013 21st Euromicro International Conference Parallel, Distributed, Network-Based Processing, Belfast, UK, 27 February–1 March 2013; pp. 233–240.
34. Dua, R.; Raja, A.R.; Kakadia, D. Virtualization vs Containerization to Support PaaS. In Proceedings of the 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, 11–14 March 2014; pp. 610–614.
35. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and Linux containers. In Proceedings of the 2015 IEEE international symposium on performance analysis of systems and software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 171–172.
36. Tosatto, A.; Ruiu, P.; Attanasio, A. Container-Based Orchestration in Cloud: State of the Art and Challenges. In Proceedings of the 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, Blumenau, Brazil, 8–10 July 2015; pp. 70–75.
37. Casalicchio, E. Autonomic Orchestration of Containers: Problem Definition and Research Challenges. In Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools, Taormina, Italy, 25–28 October 2017.
38. Heidari, P.; Lemieux, Y.; Shami, A. "QoS Assurance with Light Virtualization—A Survey. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, 12–15 December 2016; pp. 558–563.
39. Jennings, B.; Stadler, R. Resource Management in Clouds: Survey and Research Challenges. *J. Netw. Syst. Manag.* **2014**, *23*, 567–619. [CrossRef]
40. Costache, S.; Dib, D.; Parlavantzas, N.; Morin, C. Resource management in cloud platform as a service systems: Analysis and opportunities. *J. Syst. Softw.* **2017**, *132*, 98–118. [CrossRef]
41. Hindman, B.; Konwinski, A.; Platform, A.; Resource, F.-G.; Zaharia, M. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI 2011), Boston, MA, USA, 30 March–1 April 2011.
42. Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems, Bordeaux, France, 21–24 April 2015.
43. Pahl, C. Containerisation and the PaaS Cloud. *IEEE Cloud Comput.* **2015**, *2*, 24–31. [CrossRef]
44. Kratzke, N.; Peinl, R. ClouNS-a Cloud-Native Application Reference Model for Enterprise Architects. In Proceedings of the 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), Vienna, Austria, 5–9 September 2016; pp. 198–207.
45. Quint, P.-C.; Kratzke, N. Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), Madeira, Portugal, 19–21 March 2018.
46. Kratzke, N. Smuggling Multi-cloud Support into Cloud-native Applications using Elastic Container Platforms. *Proc. 7th Int. Conf. Cloud Comput. Serv. Sci.* **2017**, *2017*, 57–70.
47. Kratzke, N.; Quint, P. *Project CloudTRANSIT—Transfer Cloud-Native Applications at Runtime*; Technische Hochschule Lübeck: Lübeck, Germany, 2018. [CrossRef]
48. DeCoMAdS: Deployment and Configuration Middleware for Adaptive Software-As-A-Service. 2015. Available online: https://distrinet.cs.kuleuven.be/research/projects/DeCoMAdS (accessed on 1 March 2019).
49. Truyen, E.; Bruzek, M.; van Landuyt, D.; Lagaisse, B.; Joosen, W. Evaluation of container orchestration systems for deploying and managing NoSQL database clusters. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), Zurich, Switzerland, 17–20 December 2018.

50. Delnat, W.; Truyen, E.; Rafique, A.; van Landuyt, D.; Joosen, W. K8-Scalar: A workbench to compare autoscalers for container-orchestrated database clusters. In Proceedings of the 2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Gothenburg, Sweden, 27 May–3 June 2018; pp. 33–39.
51. Campbell, J.C.; Zhang, C.; Xu, Z.; Hindle, A.; Miller, J. Deficient documentation detection: A methodology to locate deficient project documentation using topic analysis. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013.
52. Al-Subaihin, A.A.; Sarro, F.; Black, S.; Capra, L.; Harman, M.; Jia, Y.; Zhang, Y. Clustering Mobile Apps Based on Mined Textual Features. In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement—ESEM '16, Ciudad Real, Spain, 8–9 September 2016; pp. 1–10.
53. Mei, H.; Zhang, W.; Gu, F. A feature oriented approach to modeling and reusing requirements of software product lines. In Proceedings of the 27th Annual International Computer Software and Applications Conference. COMPAC 2003, Dallas, TX, USA, 3–6 November 2003; pp. 250–256.
54. Kang, K.C.; Cohen, S.G.; Hess, J.A.; Novak, W.E.; Peterson, A.S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*; Software Engineering Inst.: Pittsburgh, PA, USA, 1990.
55. Spencer, D.; Garrett, J.J. *Card Sorting: Designing Usable Categories*; Rosenfeld Media: Brooklyn, NY, USA, 2009.
56. Demsar, J. Statistical Comparisons of Classifiers over Multiple Data Sets. *J. Mach. Learn. Res.* **2006**, *7*, 1–30.
57. Khomh, F.; Dhaliwal, T.; Zou, Y.; Adams, B.; Engineering, C. *Do Faster Releases Improve Software Quality ? An Empirical Case Study of Mozilla Firefox*; IEEE Press: Piscataway, NJ, USA, 2012; pp. 179–188.
58. Truyen, E.; van Landuyt, D.; Lagaisse, B.; Joosen, W. A Comparison between Popular Open-Source container Orchestration Frameworks. 2017. Available online: https://docs.google.com/document/d/19ozfDwmbeeBmwuAemCxNtKO1OFm7FsXyMioYjUJwZVo/ (accessed on 1 March 2019).
59. Red Hat. Overview—Core Concepts | Architecture | OpenShift Container Platform 3.11. Available online: https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/index.html (accessed on 9 November 2018).
60. Cloud Foundry. Powered by Kubernetes—Container Runtime | Cloud Foundry. Available online: https://www.cloudfoundry.org/container-runtime/ (accessed on 9 November 2018).
61. Kubernetes Home Page. Available online: https://kubernetes.io/ (accessed on 1 March 2018).
62. Docker Inc. Docker Swarm | Docker Documentation. Available online: https://docs.docker.com/swarm/ (accessed on 9 November 2018).
63. Docker Inc. Swarm Mode Overview | Docker Documentation. Available online: https://docs.docker.com/engine/swarm/ (accessed on 9 November 2018).
64. Mesosphere. Apache Mesos. Available online: http://mesos.apache.org/ (accessed on 9 November 2018).
65. Apache. Apache Aurora. Available online: http://aurora.apache.org/ (accessed on 9 November 2018).
66. Mesosphere. Marathon: A Container Orchestration Platform for Mesos and DC/OS. Available online: https://mesosphere.github.io/marathon/ (accessed on 9 November 2018).
67. Mesosphere. The Definitive Platform for Modern Apps | DC/OS. Available online: https://dcos.io/ (accessed on 9 November 2018).
68. Breitenbücher, U.; Binz, T.; Kopp, O.; Képes, K.; Leymann, F.; Wettinger, J. *Hybrid TOSCA Provisioning Plans: Integrating Declarative and Imperative Cloud Application Provisioning Technologies*; Springer International Publishing: Cham, Switzerland, 2016; pp. 239–262.
69. Docker Inc. docker.github.io/deploy-app.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/swarm/swarm_at_scale/deploy-app.md#extra-credit-deployment-with-docker-compose (accessed on 9 November 2018).
70. docker.github.io/index.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/compose/compose-file/index.md (accessed on 9 November 2018).
71. Cloud Native Computing Foundation. website/overview.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/configuration/overview.md (accessed on 9 November 2018).
72. Apache. Apache Aurora Configuration Reference. Available online: http://aurora.apache.org/documentation/latest/reference/configuration/ (accessed on 9 November 2018).

73. Mesosphere. Marathon REST API. Available online: https://mesosphere.github.io/marathon/api-console/index.html (accessed on 9 November 2018).
74. Mesosphere. Creating Services—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/deploying-services/creating-services/ (accessed on 9 November 2018).
75. Mesosphere. mesos/architecture.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/architecture.md (accessed on 9 November 2018).
76. Ghodsi, A.; Zaharia, M.; Hindman, B.; Konwinski, A.; Shenker, S.; Stoica, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types Maps Reduces. In Proceedings of the NSDI 2011, Boston, MA, USA, 30 March–1 April 2011.
77. Mesosphere. mesos/architecture.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/architecture.md#example-of-resource-offer (accessed on 9 November 2018).
78. Mesosphere. "mesos/reservation.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/reservation.md#offeroperationunreserve (accessed on 9 November 2018).
79. Mesosphere. marathon/high-availability.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/high-availability.md (accessed on 9 November 2018).
80. Mesosphere. mesos/reconciliation.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/reconciliation.md (accessed on 9 November 2018).
81. Docker Inc. docker.github.io/admin_guide.md at v17.06 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06/engine/swarm/admin_guide.md (accessed on 9 November 2018).
82. Apache. aurora/configuration.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/operations/configuration.md#replicated-log-configuration (accessed on 9 November 2018).
83. Mesosphere. High Availability—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/overview/high-availability/ (accessed on 9 November 2018).
84. Cloud Native Computing Foundation. website/highly-available-master.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/administer-cluster/highly-available-master.md (accessed on 9 November 2018).
85. Google LLC. Google Kubernetes Engine | Kubernetes Engine | Google Cloud. Available online: https://cloud.google.com/kubernetes-engine/ (accessed on 9 November 2018).
86. Canonical. Kubernetes | Ubuntu. Available online: https://www.ubuntu.com/kubernetes (accessed on 9 November 2018).
87. CoreOS. Coreos/Tectonic-Installer: Install a Kubernetes Cluster the CoreOS Tectonic Way: HA, Self-Hosted, RBAC, Etcd Operator, and More. Available online: https://github.com/coreos/tectonic-installer (accessed on 9 November 2018).
88. Apache. aurora/constraints.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/constraints.md (accessed on 9 November 2018).
89. Mesosphere. marathon/constraints.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/constraints.md#operators (accessed on 9 November 2018).
90. Gog, I.; Schwarzkopf, M.; Gleave, A.; Watson, R.N.M.; Hand, S. Firmament: Fast, Centralized Cluster Scheduling at Scale Firmament: Fast, centralized cluster scheduling at scale. In Proceedings of the OSDI, Savannah, GA, USA, 2–4 November 2016; pp. 99–115.
91. Delimitrou, C.; Kozyrakis, C. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, Salt Lake City, UT, USA, 1–5 March 2014.
92. Delimitrou, C.; Kozyrakis, C. QoS-Aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.* **2013**, *31*, 1–34. [CrossRef]
93. Jyothi, S.A.; Curino, C.; Menache, I.; Narayanamurthy, S.M.; Tumanov, A.; Yaniv, J.; Mavlyutov, R.; Goiri, Í.; Krishnan, S.; Kulkarni, J.; et al. Morpheus: Towards Automated SLAs for Enterprise Clusters. In Proceedings of the OSDI 2016, Savannah, GA, USA, 2–4 November 2016.

94. Grillet, A. Comparison of Container Schedulers. 2016. Available online: https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421 (accessed on 31 October 2017).
95. Cloud Native Computing Foundation. Home—Open Containers Initiative. Available online: https://www.opencontainers.org/ (accessed on 9 November 2018).
96. Cloud Native Computing Foundation. kubernetes-sigs/cri-o: Open Container Initiative-Based Implementation of Kubernetes Container Runtime Interface. Available online: https://github.com/kubernetes-sigs/cri-o/ (accessed on 9 November 2018).
97. Mesosphere. [MESOS-5011] Support OCI Image Spec.—ASF JIRA. Available online: https://issues.apache.org/jira/browse/MESOS-5011 (accessed on 9 November 2018).
98. Cloud Native Computing Foundation. opencontainers/runc: CLI Tool for Spawning and Running Containers According to the OCI Specification. Available online: https://github.com/opencontainers/runc (accessed on 9 November 2018).
99. Cloud Native Computing Foundation. website/components.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/overview/components.md#container-runtime (accessed on 9 November 2018).
100. Mesosphere. mesos/containerizers.md at 1.5.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.5.x/docs/containerizers.md (accessed on 9 November 2018).
101. Cloud Native Computing Foundation. community/scheduler_extender.md at master · kubernetes/community. Available online: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md (accessed on 9 November 2018).
102. Mesosphere. "mesos/allocation-module.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/allocation-module.md#writing-a-custom-allocator (accessed on 9 November 2018).
103. Apache. aurora/RELEASE-NOTES.md at master · apache/aurora. Available online: https://github.com/apache/aurora/blob/master/RELEASE-NOTES.md#0200 (accessed on 9 November 2018).
104. Apache. aurora/scheduler-configuration.md at rel/0.20.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.20.0/docs/reference/scheduler-configuration.md (accessed on 9 November 2018).
105. Mesosphere. marathon/plugin.md at v1.6.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.6.0/docs/docs/plugin.md#scheduler (accessed on 9 November 2018).
106. Cloud Native Computing Foundation. website/configure-multiple-schedulers.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/tasks/administer-cluster/configure-multiple-schedulers.md (accessed on 9 November 2018).
107. Cloud Native Computing Foundation. website/admission-controllers.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/reference/access-authn-authz/admission-controllers.md (accessed on 12 November 2018).
108. Cloud Native Computing Foundation. website/extensible-admission-controllers.md#initializers at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/reference/access-authn-authz/extensible-admission-controllers.md#initializers (accessed on 12 November 2018).
109. Cloud Native Computing Foundation. website/extensible-admission-controllers.md#admission webhooks at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/reference/access-authn-authz/extensible-admission-controllers.md#admission-webhooks (accessed on 12 November 2018).
110. Mesosphere. mesos/modules.md#hook at 1.6.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.6.x/docs/modules.md#hook (accessed on 12 November 2018).
111. Mesosphere. aurora/client-hooks.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/reference/client-hooks.md (accessed on 12 November 2018).
112. Apache. aurora/RELEASE-NOTES.md at master · apache/aurora. Available online: https://github.com/apache/aurora/blob/master/RELEASE-NOTES.md#0190 (accessed on 12 November 2018).
113. Linux Virtual Server Project. IPVS Software—Advanced Layer-4 Switching. Available online: http://www.linuxvirtualserver.org/software/ipvs.html (accessed on 12 November 2018).

114. Docker Inc. docker.github.io/networking.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/swarm/networking.md (accessed on 12 November 2018).
115. Cloud Native Computing Foundation. website/service.md at release-1.9 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.9/docs/concepts/services-networking/service.md (accessed on 12 November 2018).
116. Mesosphere. marathon/networking.md at v1.6.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.6.0/docs/docs/networking.md#specifying-service-ports (accessed on 12 November 2018).
117. Mesosphere. Networking—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/ (accessed on 12 November 2018).
118. Mesosphere. Networking—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/#layer-7 (accessed on 12 November 2018).
119. Docker Inc. docker.github.io/overlay.md at v17.12 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.12/network/overlay.md#bypass-the-routing-mesh-for-a-swarm-service (accessed on 12 November 2018).
120. Apache. aurora/service-discovery.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/service-discovery.md#using-mesos-dns (accessed on 12 November 2018).
121. Mesosphere. marathon/service-discovery-load-balancing.md at v1.6.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.6.0/docs/docs/service-discovery-load-balancing.md#mesos-dns (accessed on 12 November 2018).
122. Cloud Native Computing Foundation. website/overview.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/overview.md#services (accessed on 12 November 2018).
123. Docker Inc. docker.github.io/services.md at master · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/master/engine/swarm/services.md#publish-a-services-ports-directly-on-the-swarm-node (accessed on 12 November 2018).
124. Apache. aurora/services.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/services.md#ports (accessed on 12 November 2018).
125. Mesosphere. marathon/ports.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/ports.md#random-port-assignment (accessed on 12 November 2018).
126. Docker Inc. docker.github.io/services.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/services.md#publish-a-services-ports-directly-on-the-swarm-node (accessed on 12 November 2018).
127. RedHat. Default Scheduling—Scheduling | Cluster Administration | OKD Latest. Available online: https://docs.okd.io/latest/admin_guide/scheduling/scheduler.html#scheduler-sample-policies (accessed on 12 November 2018).
128. Cloud Native Computing Foundation. containernetworking/cni: Container Network Interface—Networking for Linux Containers. Available online: https://github.com/containernetworking/cni (accessed on 12 November 2018).
129. Cloud Native Computing Foundation. website/network-plugins.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/network-plugins.md#cni (accessed on 12 November 2018).
130. Mesosphere. mesos/cni.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/cni.md (accessed on 12 November 2018).
131. Mesosphere. CNI Plugin Support—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/networking/virtual-networks/cni-plugins/ (accessed on 12 November 2018).
132. Cloud Native Computing Foundation. CNI Plugins Should Allow Hairpin Traffic · Issue #476 · containernetworking/cni. Available online: https://github.com/containernetworking/cni/issues/476 (accessed on 12 November 2018).

133. Docker Inc. docker/libnetwork: Docker Networking. Available online: https://github.com/docker/libnetwork (accessed on 12 November 2018).
134. Mesosphere. mesos/networking.md at 1.5.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.5.x/docs/networking.md (accessed on 12 November 2018).
135. Mesosphere. Virtual Networks—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/networking/virtual-networks/ (accessed on 12 November 2018).
136. Mesosphere. DC/OS Overlay—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/SDN/dcos-overlay/#replacing-or-adding-new-virtual-networks (accessed on 12 November 2018).
137. Mesosphere. mesos/networking.md at 1.5.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.5.x/docs/networking.md#limitations-of-docker-containerizer (accessed on 12 November 2018).
138. Docker Inc. docker.github.io/overlay.md at v17.12 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.12/network/overlay.md#separate-control-and-data-traffic (accessed on 12 November 2018).
139. Intel. intel/multus-cni: Multi-Homed pod cni. Available online: https://github.com/intel/multus-cni (accessed on 12 November 2018).
140. Daboo, C.; Daboo, C. Use of SRV Records for Locating Email Submission/Access Services; Internet Engineering Task Force | IETF]. 2011. Available online: https://tools.ietf.org/html/rfc6186 (accessed on 1 March 2019).
141. Cloud Native Computing Foundation. website/dns-pod-service.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/services-networking/dns-pod-service.md#srv-records (accessed on 12 November 2018).
142. Mesosphere. Service Naming—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/DNS/mesos-dns/service-naming/#srv-records (accessed on 12 November 2018).
143. Cloud Native Computing Foundation. website/service.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/services-networking/service.md#headless-services (accessed on 12 November 2018).
144. Mesosphere. Edge-LB—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/services/edge-lb/ (accessed on 13 November 2018).
145. Mesosphere. Service Docs—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/services/ (accessed on 12 November 2018).
146. Cloud Native Computing Foundation. website/create-external-load-balancer.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/access-application-cluster/create-external-load-balancer.md (accessed on 12 November 2018).
147. Docker Inc. docker.github.io/services.md at v17.09-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.09-release/engine/swarm/services.md (accessed on 13 November 2018).
148. Mesosphere. Load Balancing and Virtual IPs (VIPs)—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/load-balancing-vips/ (accessed on 12 November 2018).
149. Mesosphere. DC/OS Domain Name Service—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/DNS/ (accessed on 12 November 2018).
150. Docker Inc. swarmkit/task_model.md at master · docker/swarmkit. Available online: https://github.com/docker/swarmkit/blob/master/design/task_model.md (accessed on 12 November 2018).
151. Cloud Native Computing Foundation. website/pod.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/workloads/pods/pod.md (accessed on 12 November 2018).
152. Mesosphere. mesos/nested-container-and-task-group.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/nested-container-and-task-group.md (accessed on 12 November 2018).
153. Mesosphere. marathon/pods.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/pods.md (accessed on 12 November 2018).

154. Mesosphere. Pods—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/deploying-services/pods/ (accessed on 13 November 2018).
155. Mesosphere. mesos/mesos-containerizer.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/mesos-containerizer.md#posix-disk-isolator (accessed on 12 November 2018).
156. Cloud Native Computing Foundation. website/jobs-run-to-completion.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/workloads/controllers/jobs-run-to-completion.md (accessed on 12 November 2018).
157. Apache. aurora/configuration-tutorial.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/reference/configuration-tutorial.md (accessed on 12 November 2018).
158. Cloud Native Computing Foundation. website/cron-jobs.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/workloads/controllers/cron-jobs.md (accessed on 12 November 2018).
159. Apache. aurora/cron-jobs.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/cron-jobs.md (accessed on 12 November 2018).
160. Cloud Native Computing Foundation. website/replicaset.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/workloads/controllers/replicaset.md (accessed on 12 November 2018).
161. Cloud Native Computing Foundation. website/labels.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/overview/working-with-objects/labels.md (accessed on 12 November 2018).
162. Cloud Native Computing Foundation. website/labels.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/overview/working-with-objects/labels.md#label-selectors (accessed on 12 November 2018).
163. Cloud Native Computing Foundation. website/horizontal-pod-autoscale.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/run-application/horizontal-pod-autoscale.md (accessed on 12 November 2018).
164. Mesosphere. Autoscaling with Marathon—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/tutorials/autoscaling/ (accessed on 12 November 2018).
165. Docker Inc. docker.github.io/services.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/services.md#control-service-scale-and-placement (accessed on 9 November 2018).
166. Cloud Native Computing Foundation. website/daemonset.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/workloads/controllers/daemonset.md (accessed on 12 November 2018).
167. Docker Inc. docker.github.io/stack-deploy.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/stack-deploy.md (accessed on 12 November 2018).
168. Mesosphere. marathon/application-groups.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/application-groups.md (accessed on 12 November 2018).
169. Cloud Native Computing Foundation. helm/helm: The Kubernetes Package Manager. Available online: https://github.com/helm/helm (accessed on 12 November 2018).
170. Cloud Native Computing Foundation. Kubernetes/Kompose: Go from Docker Compose to Kubernetes. Available online: https://github.com/kubernetes/kompose (accessed on 12 November 2018).
171. Docker Inc. docker.github.io/index.md at v18.03-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v18.03/engine/extend/index.md (accessed on 12 November 2018).
172. Mesosphere. mesos/docker-volume.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/docker-volume.md#motivation (accessed on 12 November 2018).

173. Mesosphere. marathon/external-volumes.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/external-volumes.md (accessed on 12 November 2018).
174. Portworx. Using Portworx Volumes with DCOS. Available online: https://docs.portworx.com/scheduler/mesosphere-dcos/portworx-volumes.html (accessed on 12 November 2018).
175. Dell. thecodeteam/mesos-module-dvdi: Mesos Docker Volume Driver Isolator Module. Available online: https://github.com/thecodeteam/mesos-module-dvdi (accessed on 12 November 2018).
176. K. and M. Cloud Foundry. Container-Storage-Interface/Spec: Container Storage Interface (CSI) Specification. Available online: https://github.com/container-storage-interface/spec (accessed on 12 November 2018).
177. Cloud Native Computing Foundation. website/volumes.md at release-1.10 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.10/content/en/docs/concepts/storage/volumes.md#csi (accessed on 12 November 2018).
178. Mesosphere. mesos/csi.md at 1.7.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.7.x/docs/csi.md (accessed on 12 November 2018).
179. Mesosphere. Volume Plugins—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/services/beta-storage/0.3.0-beta/volume-plugins/ (accessed on 12 November 2018).
180. Docker Inc. docker.github.io/plugins_volume.md at v18.03-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v18.03/engine/extend/plugins_volume.md (accessed on 12 November 2018).
181. Cloud Native Computing Foundation. website/downward-api-volume-expose-pod-information.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information.md (accessed on 12 November 2018).
182. Mesosphere. marathon/networking.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/networking.md#downward-api (accessed on 12 November 2018).
183. Docker Inc. docker.github.io/configs.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/configs.md (accessed on 12 November 2018).
184. Cloud Native Computing Foundation. website/configmap.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/configure-pod-container/configmap.md (accessed on 12 November 2018).
185. Apache. aurora/job-updates.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/job-updates.md (accessed on 12 November 2018).
186. Cloud Native Computing Foundation. website/configure-liveness-readiness-probes.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/configure-pod-container/configure-liveness-readiness-probes.md#define-readiness-probes (accessed on 12 November 2018).
187. Mesosphere. marathon/readiness-checks.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/readiness-checks.md (accessed on 12 November 2018).
188. Docker Inc. docker.github.io/index.md at master · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06/compose/compose-file/index.md#update_config (accessed on 12 November 2018).
189. Cloud Native Computing Foundation. website/deployment.md at release-1.10 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.10/content/en/docs/concepts/workloads/controllers/deployment.md#proportional-scaling (accessed on 12 November 2018).
190. Mesosphere. marathon/deployments.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/deployments.md#rolling-restarts (accessed on 12 November 2018).
191. Docker Inc. docker Service Update | Docker Documentation. Available online: https://docs.docker.com/engine/reference/commandline/service_update/#roll-back-to-the-previous-version-of-a-service (accessed on 12 November 2018).

192. Cloud Native Computing Foundation. website/deployment.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/workloads/controllers/deployment.md#rolling-back-to-a-previous-revision (accessed on 12 November 2018).
193. Apache. aurora/configuration.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/reference/configuration.md#updateconfig-objects (accessed on 12 November 2018).
194. Mesosphere. Dcos Marathon Deployment Rollback—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/cli/command-reference/dcos-marathon/dcos-marathon-deployment-rollback/ (accessed on 12 November 2018).
195. Limoncelli, T.A.; Chalup, S.R.; Hogan, C.J. The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems. 2014, Volume 2. Available online: http://the-cloud-book.com/ (accessed on 14 January 2016).
196. Schermann, G.; Leitner, P.; Gall, H.C. Bifrost—Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies. In Proceedings of the 17th International Middleware Conference (Middleware '16), Trento, Italy, 12–16 December 2016.
197. Cloud Native Computing Foundation. website/manage-deployment.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/manage-deployment.md#canary-deployments (accessed on 12 November 2018).
198. Mesosphere. marathon/blue-green-deploy.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/blue-green-deploy.md (accessed on 12 November 2018).
199. Mesosphere. mesos/roles.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/roles.md (accessed on 12 November 2018).
200. Cloud Native Computing Foundation. website/resource-quotas.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/policy/resource-quotas.md#compute-resource-quota (accessed on 12 November 2018).
201. Cloud Native Computing Foundation. website/resource-quotas.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/policy/resource-quotas.md#storage-resource-quota (accessed on 12 November 2018).
202. Mesosphere. mesos/quota.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/quota.md (accessed on 12 November 2018).
203. Mesosphere. mesos/operator-http-api.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/operator-http-api.md#get_roles (accessed on 12 November 2018).
204. Mesosphere. mesos/weights.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/weights.md (accessed on 12 November 2018).
205. Apache. aurora/multitenancy.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/multitenancy.md#configuration-tiers (accessed on 12 November 2018).
206. Cloud Native Computing Foundation. website/resource-quotas.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/policy/resource-quotas.md#object-count-quota (accessed on 12 November 2018).
207. Docker Inc. docker.github.io/index.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/datacenter/ucp/2.2/guides/access-control/index.md (accessed on 12 November 2018).
208. Docker Inc. docker.github.io/isolate-nodes-between-teams.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/datacenter/ucp/2.2/guides/access-control/isolate-nodes-between-teams.md (accessed on 12 November 2018).
209. Docker Inc. docker.github.io/isolate-volumes-between-teams.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/datacenter/ucp/2.2/guides/access-control/isolate-volumes-between-teams.md (accessed on 12 November 2018).
210. Mesosphere. Tutorial—Restricting Access to DC/OS Service Groups—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/security/ent/restrict-service-access/#create-users-and-groups (accessed on 12 November 2018).

211. Cloud Native Computing Foundation. website/reserve-compute-resources.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/administer-cluster/reserve-compute-resources.md (accessed on 12 November 2018).
212. Mesosphere. marathon/pods.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/pods.md#executor-resources (accessed on 12 November 2018).
213. Cloud Native Computing Foundation. website/manage-compute-resources-container.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/manage-compute-resources-container.md (accessed on 12 November 2018).
214. Cloud Native Computing Foundation. kubernetes/resource-qos.md at release-1.2 · kubernetes/kubernetes. Available online: https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/proposals/resource-qos.md (accessed on 12 November 2018).
215. Cloud Native Computing Foundation. website/assign-memory-resource.md at release-1.12 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.12/content/en/docs/tasks/configure-pod-container/assign-memory-resource.md#exceed-a-containers-memory-limit (accessed on 12 November 2018).
216. Docker Inc. docker.github.io/index.md at master · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06/compose/compose-file/index.md#resources (accessed on 12 November 2018).
217. Mesosphere. mesos/gpu-support.md at 1.7.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.7.x/docs/gpu-support.md (accessed on 12 November 2018).
218. Apache. aurora/resource-isolation.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/resource-isolation.md (accessed on 12 November 2018).
219. Mesosphere. marathon/pod.json at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/rest-api/public/api/v2/examples/pod.json (accessed on 12 November 2018).
220. Mesosphere. Using GPUs—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/deploying-services/gpu/ (accessed on 12 November 2018).
221. Cloud Native Computing Foundation. website/scheduling-gpus.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/tasks/manage-gpus/scheduling-gpus.md (accessed on 12 November 2018).
222. Apache. aurora/resource-isolation.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/resource-isolation.md#disk-space (accessed on 12 November 2018).
223. Cloud Native Computing Foundation. website/manage-compute-resources-container.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/configuration/manage-compute-resources-container.md#local-ephemeral-storage-alpha-feature (accessed on 12 November 2018).
224. Docker Inc. Docker Service Update | Docker Documentation. Available online: https://docs.docker.com/v17.06/engine/reference/commandline/service_update/ (accessed on 12 November 2018).
225. Cloud Native Computing Foundation. website/assign-pod-node.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/assign-pod-node.md#step-one-attach-label-to-the-node (accessed on 12 November 2018).
226. Mesosphere. marathon/constraints.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/constraints.md (accessed on 12 November 2018).
227. Mesosphere. Frequently Asked Questions—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/installing/installation-faq/#q-how-to-add-mesos-attributes-to-nodes-to-use-marathon-constraints (accessed on 12 November 2018).
228. Mesosphere. mesos/attributes-resources.md at master · apache/mesos. Available online: https://github.com/apache/mesos/blob/master/docs/attributes-resources.md#attributes (accessed on 12 November 2018).
229. Docker Inc. docker.github.io/services.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/services.md#specify-service-placement-preferences---placement-pref (accessed on 12 November 2018).

230. Cloud Native Computing Foundation. website/assign-pod-node.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/assign-pod-node.md#nodeselector (accessed on 12 November 2018).
231. Cloud Native Computing Foundation. website/assign-pod-node.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/assign-pod-node.md#inter-pod-affinity-and-anti-affinity-beta-feature (accessed on 12 November 2018).
232. Cloud Native Computing Foundation. website/taint-and-toleration.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/taint-and-toleration.md (accessed on 12 November 2018).
233. Cloud Native Computing Foundation. website/pod-priority-preemption.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/pod-priority-preemption.md (accessed on 12 November 2018).
234. Apache. aurora/multitenancy.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/multitenancy.md#preemption (accessed on 12 November 2018).
235. Cloud Native Computing Foundation. website/manage-compute-resources-container.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/configuration/manage-compute-resources-container.md#how-pods-with-resource-requests-are-scheduled (accessed on 12 November 2018).
236. Mesosphere. mesos/high-availability-framework-guide.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/high-availability-framework-guide.md#dealing-with-partitioned-or-failed-agents (accessed on 12 November 2018).
237. Mesosphere. mesos/agent-recovery.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/agent-recovery.md (accessed on 12 November 2018).
238. Docker Inc. docker.github.io/configure-tls.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/swarm/configure-tls.md (accessed on 12 November 2018).
239. Docker Inc. docker.github.io/pki.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/how-swarm-mode-works/pki.md (accessed on 12 November 2018).
240. Cloud Native Computing Foundation. website/kubelet-tls-bootstrapping.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/admin/kubelet-tls-bootstrapping.md (accessed on 12 November 2018).
241. Mesosphere. mesos/authentication.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/authentication.md (accessed on 12 November 2018).
242. Apache. aurora/security.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/operations/security.md#announcer-authentication (accessed on 12 November 2018).
243. Docker Inc. docker.github.io/create-swarm.md at v17.06 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06/engine/swarm/swarm-tutorial/create-swarm.md (accessed on 12 November 2018).
244. Cloud Native Computing Foundation. website/bootstrap-tokens.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/admin/bootstrap-tokens.md (accessed on 12 November 2018).
245. Cloud Native Computing Foundation. website/node.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/reference/access-authn-authz/node.md (accessed on 14 January 2019).
246. Mesosphere. mesos/authorization.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/authorization.md#authorizable-actions (accessed on 12 November 2018).
247. Apache. ZooKeeper Programmer's Guide. Available online: https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html#sc_ZooKeeperAccessControl (accessed on 12 November 2018).

248. Docker Inc. docker.github.io/overlay-security-model.md at v17.06 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06/engine/userguide/networking/overlay-security-model.md (accessed on 16 November 2018).
249. Cloud Native Computing Foundation. website/master-node-communication.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/architecture/master-node-communication.md#ssh-tunnels (accessed on 16 November 2018).
250. Mesosphere. DC/OS Enterprise Security—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/security/ent/#transport-layer-security-tls-encryption (accessed on 16 November 2018).
251. Docker Inc. docker.github.io/networking.md at v17.06 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06/engine/swarm/networking.md#configure-encryption-of-application-data (accessed on 16 November 2018).
252. Weaveworks, "Weave Net". Available online: https://www.weave.works/oss/net/ (accessed on 1 March 2019).
253. Docker Inc. moby/swarm_init.md at 17.05.x · moby/moby. Available online: https://github.com/moby/moby/blob/17.05.x/docs/reference/commandline/swarm_init.md#--listen-addr (accessed on 16 November 2018).
254. Cloud Native Computing Foundation. community/nodeport-ip-range.md at master · kubernetes/community. Available online: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/network/nodeport-ip-range.md (accessed on 16 November 2018).
255. Mesosphere. Node Types—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/overview/architecture/node-types/ (accessed on 16 November 2018).
256. Shu, R.; Gu, X.; Enck, W. A Study of Security Vulnerabilities on Docker Hub. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AZ, USA, 22–24 March 2017; pp. 269–280.
257. Combe, T.; Martin, A.; di Pietro, R. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Comput.* **2016**, *3*, 54–62. [CrossRef]
258. Docker Inc. docker.github.io/secrets.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/secrets.md (accessed on 16 November 2018).
259. Cloud Native Computing Foundation. website/secret.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/secret.md (accessed on 16 November 2018).
260. Mesosphere. mesos/secrets.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/secrets.md (accessed on 16 November 2018).
261. Mesosphere. marathon/secrets.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/secrets.md (accessed on 16 November 2018).
262. Mesosphere. Creating Secrets—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/security/ent/secrets/create-secrets/ (accessed on 16 November 2018).
263. Xavier, M.G.; Neves, M.V.; de Rose, C.A.F. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. In Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Torino, Italy, 12–14 February 2014; pp. 299–306.
264. SELinux Project. SELinux Policy Analysis Tools. Available online: https://github.com/SELinuxProject/setools (accessed on 16 November 2018).
265. AppArmor/apparmor · GitLab. Available online: https://gitlab.com/apparmor/apparmor (accessed on 15 November 2018).
266. Jonathan Corbet. Yet Another New Approach to Seccomp [LWN.net]. Available online: https://lwn.net/Articles/475043/ (accessed on 16 November 2018).
267. bpf(4) Berkeley Packet Filter. FreeBSD. Available online: https://www.freebsd.org/cgi/man.cgi?bpf(4) (accessed on 1 March 2019).
268. Linux Audit. Linux Capabilities 101—Linux Audit. Available online: https://linux-audit.com/linux-capabilities-101/ (accessed on 15 November 2018).

269. Docker Inc. docker.github.io/security.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/security/security.md#linux-kernel-capabilities (accessed on 15 November 2018).
270. Cloud Native Computing Foundation. website/security-context.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/configure-pod-container/security-context.md#set-capabilities-for-a-container (accessed on 15 November 2018).
271. RedHat. Chapter 6. Docker SELinux Security Policy—Red Hat Customer Portal. Available online: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/docker_selinux_security_policy (accessed on 15 November 2018).
272. Cloud Native Computing Foundation. website/security-context.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/configure-pod-container/security-context.md#assign-selinux-labels-to-a-container (accessed on 15 November 2018).
273. Docker Inc. docker.github.io/apparmor.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/security/apparmor.md (accessed on 15 November 2018).
274. Cloud Native Computing Foundation. website/apparmor.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tutorials/clusters/apparmor.md (accessed on 15 November 2018).
275. Docker Inc. docker.github.io/seccomp.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/security/seccomp.md (accessed on 15 November 2018).
276. RedHat. Restricting Application Capabilities Using Seccomp | Cluster Administration | OpenShift Container Platform 3.3. Available online: https://docs.openshift.com/container-platform/3.3/admin_guide/seccomp.html (accessed on 15 November 2018).
277. Cloud Native Computing Foundation. website/security-context.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/configure-pod-container/security-context.md (accessed on 15 November 2018).
278. Docker Inc. Moby/Libentitlement: Entitlements Library for High Level Control of Container Permissions. Available online: https://github.com/moby/libentitlement (accessed on 15 November 2018).
279. Docker Inc. Entitlements on Moby and Kubernetes—Google Docs. Available online: https://docs.google.com/document/d/1j3BJUNBsgi-nxJHoIJHsXRRtVWT5lrwsI2EN9WMQaes/edit#heading=h.yhnr195944yh (accessed on 15 November 2018).
280. Docker Inc. Universal Control Plane Overview | Docker Documentation. Available online: https://docs.docker.com/v17.06/datacenter/ucp/2.2/guides/ (accessed on 15 November 2018).
281. Cloud Native Computing Foundation. Kubernetes/Dashboard: General-Purpose Web UI for Kubernetes Clusters. Available online: https://github.com/kubernetes/dashboard (accessed on 15 November 2018).
282. Mesosphere. GUI—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/gui/ (accessed on 15 November 2018).
283. Docker Inc. docker.github.io/index.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/compose/compose-file/index.md#labels-1 (accessed on 15 November 2018).
284. Docker Inc. docker.github.io/manage-nodes.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/manage-nodes.md#add-or-remove-label-metadata (accessed on 15 November 2018).
285. Mesosphere. Apache Mesos 0.22.0 Released—Mesosphere. Available online: https://mesosphere.com/blog/mesos-0-22-0-released/ (accessed on 15 November 2018).
286. Mesosphere. Labeling Tasks and Jobs—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/tutorials/task-labels/ (accessed on 15 November 2018).
287. Apache. aurora/observer-configuration.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/reference/observer-configuration.md (accessed on 15 November 2018).

288. Cloud Native Computing Foundation. heapster/deprecation.md at master · kubernetes/heapster. Available online: https://github.com/kubernetes/heapster/blob/master/docs/deprecation.md (accessed on 15 November 2018).
289. Cloud Native Computing Foundation. Core Metrics Pipeline—Kubernetes. Available online: https://kubernetes.io/docs/tasks/debug-application-cluster/core-metrics-pipeline/ (accessed on 15 November 2018).
290. Cloud Native Computing Foundation. website/resource-usage-monitoring.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/tasks/debug-application-cluster/resource-usage-monitoring.md#full-metrics-pipelines (accessed on 15 November 2018).
291. Mesosphere. mesos/statistics.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/endpoints/slave/monitor/statistics.md (accessed on 15 November 2018).
292. Mesosphere. mesosphere/marathon-lb: Marathon-lb Is a Service Discovery Load Balancing Tool for DC/OS. Available online: https://github.com/mesosphere/marathon-lb (accessed on 12 November 2018).
293. Mesosphere. mesos/port-mapping-isolator.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/port-mapping-isolator.md#monitoring-container-network-statistics (accessed on 15 November 2018).
294. Mesosphere. [MESOS-5647] Expose Network Statistics for Containers on CNI Network in the 'network/cni' Isolator.—ASF JIRA. Available online: https://issues.apache.org/jira/browse/MESOS-5647 (accessed on 15 November 2018).
295. Mesosphere. Metrics—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/metrics/ (accessed on 15 November 2018).
296. Docker Inc. docker.github.io/prometheus.md at v17.12 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.12/config/thirdparty/prometheus.md (accessed on 15 November 2018).
297. Cloud Native Computing Foundation. website/controller-metrics.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/controller-metrics.md (accessed on 15 November 2018).
298. Mesosphere. mesos/monitoring.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/monitoring.md (accessed on 15 November 2018).
299. Apache. aurora/monitoring.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/operations/monitoring.md (accessed on 15 November 2018).
300. Mesosphere. marathon/metrics.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/metrics.md (accessed on 15 November 2018).
301. Mesosphere. Performance Monitoring—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/monitoring/performance-monitoring/ (accessed on 15 November 2018).
302. Cloud Native Computing Foundation. community/accelerator-monitoring.md at master · kubernetes/community. Available online: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/accelerator-monitoring.md (accessed on 15 November 2018).
303. Mesosphere. mesos/monitoring.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/monitoring.md#resources (accessed on 15 November 2018).
304. Docker Inc. docker.github.io/troubleshoot-with-logs.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-with-logs.md (accessed on 15 November 2018).
305. Cloud Native Computing Foundation. website/logging.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/logging.md (accessed on 15 November 2018).
306. Mesosphere. mesos/logging.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/logging.md#containers (accessed on 15 November 2018).
307. Mesosphere. Logging—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/monitoring/logging/#service-task-and-node-logs (accessed on 15 November 2018).
308. Linux. systemd-journald.service(8)—Linux Manual Page. Available online: http://man7.org/linux/man-pages/man8/systemd-journald.service.8.html (accessed on 15 November 2018).

309. logz.io. Docker Swarm Logging with ELK and the Logz.io Log Collector. Available online: https://logz.io/blog/docker-swarm-logging/ (accessed on 15 November 2018).
310. Cloud Native Computing Foundation. website/logging-elasticsearch-kibana.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/debug-application-cluster/logging-elasticsearch-kibana.md (accessed on 15 November 2018).
311. Mesosphere. Log Aggregation—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/monitoring/logging/aggregating/ (accessed on 15 November 2018).
312. Docker Inc. docker.github.io/admin_guide.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/admin_guide.md#back-up-the-swarm (accessed on 15 November 2018).
313. Mesosphere. mesos/replicated-log-internals.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/replicated-log-internals.md (accessed on 15 November 2018).
314. Apache. aurora/backup-restore.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/operations/backup-restore.md (accessed on 15 November 2018).
315. Mesosphere. marathon/backup-restore.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/backup-restore.md (accessed on 15 November 2018).
316. Mhausenblas/Reshifter: Kubernetes Cluster State Management. Available online: https://github.com/mhausenblas/reshifter (accessed on 15 November 2018).
317. Cloud Native Computing Foundation. website/cluster-management.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/administer-cluster/cluster-management.md#upgrading-a-cluster (accessed on 15 November 2018).
318. Mesosphere. Upgrading—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/installing/production/upgrading/ (accessed on 15 November 2018).
319. Apache. aurora/upgrades.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/operations/upgrades.md (accessed on 15 November 2018).
320. Mesosphere. marathon/index.md at v1.5.0 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.5.0/docs/docs/upgrade/index.md (accessed on 15 November 2018).
321. Docker Inc. docker.github.io/drain-node.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/swarm-tutorial/drain-node.md (accessed on 15 November 2018).
322. Cloud Native Computing Foundation. website/safely-drain-node.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/administer-cluster/safely-drain-node.md (accessed on 15 November 2018).
323. Mesosphere. mesos/maintenance.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/maintenance.md (accessed on 15 November 2018).
324. Mesosphere. marathon/maintenance-mode.md at v1.6.322 · mesosphere/marathon. Available online: https://github.com/mesosphere/marathon/blob/v1.6.322/docs/docs/maintenance-mode.md (accessed on 15 November 2018).
325. Mesosphere. Updating Nodes—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.10/administering-clusters/update-a-node/ (accessed on 15 November 2018).
326. Docker Inc. docker.github.io/garbage-collection.md at v17.09-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.09-release/registry/garbage-collection.md (accessed on 15 November 2018).
327. Cloud Native Computing Foundation. website/kubelet-garbage-collection.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/kubelet-garbage-collection.md (accessed on 15 November 2018).
328. Mesosphere. mesos/container-image.md at 1.5.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.5.x/docs/container-image.md#garbage-collect-unused-container-images (accessed on 15 November 2018).
329. Mesosphere. Components—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/overview/architecture/components/#docker-gc (accessed on 15 November 2018).

330. Docker Inc. docker.github.io/plan-for-production.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/swarm/plan-for-production.md#multiple-clouds (accessed on 15 November 2018).
331. Docker Inc. docker.github.io/admin_guide.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/engine/swarm/admin_guide.md#distribute-manager-nodes (accessed on 15 November 2018).
332. Cloud Native Computing Foundation. website/multiple-zones.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/admin/multiple-zones.md (accessed on 15 November 2018).
333. Mesosphere. mesos/high-availability-framework-guide.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/high-availability-framework-guide.md#dealing-with-partitioned-or-failed-masters (accessed on 15 November 2018).
334. Apache. aurora/configuration.md at rel/0.20.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.20.0/docs/reference/configuration.md#job-objects (accessed on 15 November 2018).
335. Docker Inc. docker.github.io/index.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/docker-cloud/cloud-swarm/index.md (accessed on 15 November 2018).
336. Cloud Native Computing Foundation. website/set-up-cluster-federation-kubefed.md at release-1.9 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.9/docs/tasks/federation/set-up-cluster-federation-kubefed.md (accessed on 15 November 2018).
337. Mesosphere. Multiple Clusters—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/administering-clusters/multiple-clusters/ (accessed on 15 November 2018).
338. Cloud Native Computing Foundation. website/set-up-cluster-federation-kubefed.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/federation/set-up-cluster-federation-kubefed.md#basic-and-token-authentication-support (accessed on 15 November 2018).
339. Mesosphere. Cluster Links—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/administering-clusters/multiple-clusters/cluster-links/ (accessed on 15 November 2018).
340. Mesosphere. mesos/fault-domains.md at 1.6.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.6.x/docs/fault-domains.md (accessed on 15 November 2018).
341. Apache. aurora/constraints.md at rel/0.18.0 · apache/aurora. Available online: https://github.com/apache/aurora/blob/rel/0.18.0/docs/features/constraints.md#limit-constraints (accessed on 15 November 2018).
342. Mesosphere. Fault Domain Awareness and Capacity Extension—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/deploying-services/fault-domain-awareness/ (accessed on 15 November 2018).
343. Cloud Native Computing Foundation. website/federation.md at release-1.9 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.9/docs/concepts/cluster-administration/federation.md (accessed on 15 November 2018).
344. Cloud Native Computing Foundation. website/pick-right-solution.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/setup/pick-right-solution.md#hosted-solutions (accessed on 9 November 2018).
345. Cloud Native Computing Foundation. kubernetes-incubator/external-dns: Configure External DNS Servers (AWS Route53, Google CloudDNS and Others) for Kubernetes Ingresses and Services. Available online: https://github.com/kubernetes-incubator/external-dns (accessed on 12 November 2018).
346. Cloud Native Computing Foundation. website/ip-masq-agent.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/administer-cluster/ip-masq-agent.md (accessed on 12 November 2018).
347. Cloud Native Computing Foundation. website/extend-cluster.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/extend-kubernetes/extend-cluster.md (accessed on 12 November 2018).
348. Cloud Native Computing Foundation. website/cloud-controller.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/architecture/cloud-controller.md (accessed on 12 November 2018).

349. Cloud Native Computing Foundation. website/annotations.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/overview/working-with-objects/annotations.md (accessed on 12 November 2018).
350. Cloud Native Computing Foundation. website/custom-resources.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/extend-kubernetes/api-extension/custom-resources.md (accessed on 12 November 2018).
351. Cloud Native Computing Foundation. website/apiserver-aggregation.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation.md (accessed on 12 November 2018).
352. Cloud Native Computing Foundation. website/device-plugins.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins.md (accessed on 12 November 2018).
353. Cloud Native Computing Foundation. autoscaler/vertical-pod-autoscaler at master · kubernetes/autoscaler. Available online: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler (accessed on 12 November 2018).
354. Cloud Native Computing Foundation. website/podpreset.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/inject-data-application/podpreset.md (accessed on 12 November 2018).
355. Cloud Native Computing Foundation. website/statefulset.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/workloads/controllers/statefulset.md (accessed on 12 November 2018).
356. Cloud Native Computing Foundation. website/persistent-volumes.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/storage/persistent-volumes.md#raw-block-volume-support (accessed on 12 November 2018).
357. Cloud Native Computing Foundation. website/persistent-volumes.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/storage/persistent-volumes.md#resizing-an-in-use-persistentvolumeclaim (accessed on 12 November 2018).
358. Cloud Native Computing Foundation. website/storage-limits.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/concepts/storage/storage-limits.md (accessed on 12 November 2018).
359. Cloud Native Computing Foundation. website/scheduling-hugepages.md at release-1.11 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/tasks/manage-hugepages/scheduling-hugepages.md (accessed on 12 November 2018).
360. Cloud Native Computing Foundation. website/cpu-management-policies.md at release-1.10 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.10/content/en/docs/tasks/administer-cluster/cpu-management-policies.md (accessed on 12 November 2018).
361. Cloud Native Computing Foundation. website/manage-compute-resources-container.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/configuration/manage-compute-resources-container.md#extended-resources (accessed on 12 November 2018).
362. Cloud Native Computing Foundation. website/audit.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/debug-application-cluster/audit.md (accessed on 16 November 2018).
363. Cloud Native Computing Foundation. website/kubelet-authentication-authorization.md at release-1.10 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.10/content/en/docs/reference/command-line-tools-reference/kubelet-authentication-authorization.md (accessed on 16 November 2018).
364. Cloud Native Computing Foundation. website/network-policies.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/services-networking/network-policies.md (accessed on 16 November 2018).
365. Cloud Native Computing Foundation. website/pod-security-policy.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/policy/pod-security-policy.md#what-is-a-pod-security-policy (accessed on 15 November 2018).

366. Cloud Native Computing Foundation. website/sysctl-cluster.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/sysctl-cluster.md (accessed on 15 November 2018).
367. Cloud Native Computing Foundation. autoscaler/cluster-autoscaler at master · kubernetes/autoscaler. Available online: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler (accessed on 15 November 2018).
368. Cloud Native Computing Foundation. website/port-forward-access-application-cluster.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/access-application-cluster/port-forward-access-application-cluster.md (accessed on 15 November 2018).
369. Cloud Native Computing Foundation. website/configure-pdb.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/tasks/run-application/configure-pdb.md (accessed on 15 November 2018).
370. Cloud Native Computing Foundation. website/federation.md at release-1.8 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.8/docs/concepts/cluster-administration/federation.md#api-resources (accessed on 15 November 2018).
371. Cloud Native Computing Foundation. website/federation-service-discovery.md at release-1.9 · kubernetes/website. Available online: https://github.com/kubernetes/website/blob/release-1.9/docs/tasks/federation/federation-service-discovery.md (accessed on 15 November 2018).
372. Mesosphere. mesos/resource-provider.md at 1.7.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.7.x/docs/resource-provider.md (accessed on 12 November 2018).
373. Mesosphere. Networking—Mesosphere DC/OS Documentation. Available online: https://docs.mesosphere.com/1.11/networking/#edge-lb-enterprise (accessed on 13 November 2018).
374. Mesosphere. mesos/shared-resources.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/shared-resources.md (accessed on 12 November 2018).
375. Mesosphere. mesos/framework-rate-limiting.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/framework-rate-limiting.md (accessed on 12 November 2018).
376. Mesosphere. mesosphere/dcos-commons: Simplifying Stateful Services. Available online: https://github.com/mesosphere/dcos-commons/ (accessed on 12 November 2018).
377. Mesosphere. mesos/port-mapping-isolator.md at 1.4.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.4.x/docs/port-mapping-isolator.md (accessed on 12 November 2018).
378. Mesosphere. mesos/cgroups-net-cls.md at 1.5.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.5.x/docs/isolators/cgroups-net-cls.md (accessed on 12 November 2018).
379. Docker Inc. docker.github.io/compose-file-v2.md at v17.06-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.06-release/compose/compose-file/compose-file-v2.md#init (accessed on 12 November 2018).
380. Docker Inc. docker.github.io/index.md at v18.03 · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v18.03/compose/compose-file/index.md#init (accessed on 12 November 2018).
381. Docker Inc. Docker Engine API v1.37 Reference. Available online: https://docs.docker.com/engine/api/v1.37/#operation/ContainerUpdate (accessed on 15 November 2018).
382. Al-Dhuraibi, Y.; Paraiso, F.; Djarallah, N.; Merle, P. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. In Proceedings of the IEEE International Conference on Cloud Computing, CLOUD, Honolulu, CA, USA, 25–30 June 2017.
383. Docker Inc. docker.github.io/completion.md at v17.09-release · docker/docker.github.io. Available online: https://github.com/docker/docker.github.io/blob/v17.09-release/compose/completion.md (accessed on 15 November 2018).
384. Cloud Native Computing Foundation. Federation—Kubernetes. Available online: https://v1-11.docs.kubernetes.io/docs/concepts/cluster-administration/federation/ (accessed on 16 November 2018).
385. Arnautov, S.; Trach, B.; Gregor, F.; Knauth, T.; Martin, A.; Priebe, C.; Lind, J.; Muthukumaran, D.; O'Keeffe, D.; Stillwell, M.L.; et al. SCONE: Secure Linux Containers with Intel SGX SCONE: Secure Linux Containers with Intel SGX. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), Savannah, GA, USA, 2–4 November 2016.

386. Khalid, J.; Rozner, E.; Felter, W.; Xu, C.; Rajamani, K.; Ferreira, A.; Akella, A.; Design, S.; Nsdi, I. Iron: Isolating Network-Based CPU in Container Environments. In Proceedings of the NSDI '18, Renton, WA, USA, 9–11 April 2018.
387. Xu, C.; Rajamani, K.; Felter, W. NBWGuard: Realizing Network QoS for Kubernetes. In Proceedings of the 19th International Middleware Conference Industry on—Middleware '18, Rennes, France, 10–14 December 2018; pp. 32–38.
388. Woo, S.; Sherry, J.; Han, S.; Moon, S.; Ratnasamy, S.; Shenker, S. Elastic Scaling of Stateful Network Functions. In Proceedings of the USENIX NSDI, Renton, WA, USA, 9–11 April 2018.
389. Truyen, E.; van Landuyt, D.; Lagaisse, B.; Joosen, W. Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), Limassol, Cyprus, 8–12 April 2019.
390. Buyya, R.; Ranjan, R.; Calheiros, R.N. *InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services*; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6081, pp. 13–31.
391. Istio. Istio: Connect, Secure, Control and Oberve Services. Available online: https://istio.io/ (accessed on 23 January 2019).
392. John, J.; Curran, R. Istio/Istio Soft Multi-tenancy Support. Available online: https://istio.io/blog/2018/soft-multitenancy/ (accessed on 23 January 2019).
393. Soldani, J.; Tamburri, D.A.; van den Heuvel, W.J. The pains and gains of microservices: A Systematic grey literature review. *J. Syst. Softw.* **2018**, *146*, 215–232. [CrossRef]
394. Chen, Y.; Iyer, S.; Liu, X.; Milojicic, D.; Sahai, A. SLA Decomposition: Translating Service Level Objectives to System Level Thresholds. In Proceedings of the Fourth International Conference on Autonomic Computing (ICAC'07), Jacksonville, FL, USA, 11–15 June 2007; p. 3.
395. Barna, C.; Khazaei, H.; Fokaefs, M.; Litoiu, M. Delivering Elastic Containerized Cloud Applications to Enable DevOps. In Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Buenos Aires, Argentina, 22–23 May 2017; pp. 65–75.
396. Berger, D.; Berg, B.; Zhu, T.; Harchol-Balter, M.; Sen, S. RobinHood: Tail Latency-Aware Caching—Dynamically Reallocating from Cache-Rich to Cache-Poor Daniel. In Proceedings of the OSDI '18, Carlsbad, CA, USA, 8–10 October 2018.
397. Sriraman, A.; Wenisch, T.F.; Osdi, I.; Wenisch, T.F. µTune: Auto-Tuned Threading for OLDI Microservices. In Proceedings of the OSDI 2018, Carlsbad, CA, USA, 8–10 October 2018.
398. Cloud Native Computing Foundation. cri-tools/crictl.md at release-1.11 · kubernetes-sigs/cri-tools. Available online: https://github.com/kubernetes-sigs/cri-tools/blob/release-1.11/docs/crictl.md (accessed on 21 January 2019).
399. Mesosphere. mesos/mesos-containerizer.md at 1.7.x · apache/mesos. Available online: https://github.com/apache/mesos/blob/1.7.x/docs/mesos-containerizer.md#isolators (accessed on 9 November 2018).
400. Yang, C. Checkpoint and Restoration of Micro-Service in Docker Containers. In Proceedings of the 3rd International Conference on Mechatronics and Industrial Informatics, Zhuhai, China, 30–31 October 2015. [CrossRef]
401. Shekhar, S.; Abdel-Aziz, H.; Bhattacharjee, A.; Gokhale, A.; Koutsoukos, X. Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018.
402. Kalavri, V.; Liagouris, J.; Hoffmann, M.; Dimitrova, D.; Zurich, E.; Forshaw, M.; Roscoe, T. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In Proceedings of the OSDI '18, Carlsbad, CA, USA, 8–10 October 2018.
403. Lang, W.; Shankar, S.; Patel, J.M.; Kalhan, A. Towards multi-tenant performance SLOs. *IEEE Trans. Knowl. Data Eng.* **2014**, *26*, 1447–1463. [CrossRef]
404. Cloud Native Computing Foundation. website/cluster-large.md at release-1.11 · kubernetes/website · GitHub. Available online: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/setup/cluster-large.md (accessed on 11 February 2019).