Enabling Container Cluster Interoperability using a TOSCA Orchestration Framework

Domenico Calcaterra, Giuseppe Di Modica, Pietro Mazzaglia and Orazio Tomarchio
Department of Electrical, Electronic and Computer Engineering, University of Catania, Catania, Italy

Keywords: Containerised Applications, Deployment Orchestration, Cloud Provisioning, TOSCA, BPMN.

Abstract: Cloud orchestration frameworks are recognised as a useful tool to tackle the complexity of managing the life-cycle of Cloud resources. In scenarios where resources happen to be supplied by multiple providers, such complexity is further exacerbated by portability and interoperability issues due to incompatibility of providers' proprietary interfaces. Container-based technologies do provide a solution to improve portability in the Cloud deployment landscape. Though the majority of Cloud orchestration tools support containerisation, they usually provide integration with a limited set of container-based cluster technologies without focusing on standard-based approaches for the description of containerised applications. In this work we discuss how we managed to embed the containerisation feature into a TOSCA-based Cloud orchestrator in a way that enables it to theoretically interoperate with any container run-time software. Tests were run on a software prototype to prove the approach viability.

1 INTRODUCTION

Cloud Computing is a distributed system paradigm which enables the sharing of resource pools, on an on-demand basis model. For the IT industry, this leads to several benefits in terms of availability, scalability and costs, lowering the barriers to innovation (Marston et al., 2011). Moreover, Cloud technologies encourage a larger distribution of services across the internet (Dikaiakos et al., 2009). The resources, while being allocated in remote data centers, may be accessible from different parts of the world thanks to third-party providers offering extensive network infrastructures.

Since Cloud has emerged as a dominating paradigm for application distribution, providers have implemented several new features in order to offer services which are not restricted to infrastructure provisioning. This trend is depicted as "Everything as a Service", namely *XaaS* (Duan et al., 2015). Despite the wide choice of Cloud providers and services, there still exists an intrinsic complexity in the deployment and management of Cloud applications, which makes the process draining and time-consuming.

In this respect, orchestration tools have increased their popularity in recent years, becoming a main topic for Cloud research (Weerasiri et al., 2017). The development of high-level specification languages to describe the topology of Cloud services facilitates the orchestration process and aims to portability and interoperability across different providers (Petcu and Vasilakos, 2014). With regard to standard initiatives, OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) (OASIS, 2013) stands out for the large number of works which are based upon it (Bellendorf and Mann, 2018).

In recent years, container-based applications provided a solution in order to improve portability in the Cloud deployment landscape (Pahl, 2015). Containers offer packaged software units which run on a virtualised environment. Their decoupling from the running environment eases their deployment process and the management of their dependencies. These qualities, abetted by the lightweight nature of containers, their high reusability and their near-native performances (Ruan et al., 2016) raised significant interest in the business-oriented context.

Containers can be either run as standalone services or organised in swarm services. Swarm services increase the flexibility of containers, allowing them to run on clusters of resources. This approach combines well with the Cloud Computing paradigm, providing faster management operations while granting all the advantages of Cloud services.

In this paper, we describe a framework for the deployment and orchestration of containerised applications. Based on the work presented in (Calcaterra

Calcaterra, D., Di Modica, G., Mazzaglia, P. and Tomarchio, O.
Enabling Container Cluster Interoperability using a TOSCA Orchestration Framework.
DOI: 10.5220/0009410701270137
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 127-137
ISBN: 978-989-758-424-4
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

et al., 2017) and (Calcaterra et al., 2018), the framework provides several desirable features, such as the possibility to describe the application using standard languages, a fault-aware orchestration system built on business-process models, compatibility with the main Cloud providers, and integration with different container-based cluster technologies.

The rest of the paper is structured as follows. In Section 2, we arrange a background of the technologies exploited for this work. In Section 3, we present relevant works concerning container orchestration. Section 4 debates cluster orchestrators and their interoperability. In Section 5, the approach adopted for our framework is discussed. Section 6 discusses about a prototype implementation and an experiment run on a real-world application scenario. Finally, Section 7 concludes the work.

2 BACKGROUND

This work aims to provide synergy between containerisation technologies and the most famous topology specification language, namely OASIS TOSCA. In this section, we provide a more in-depth background for these topics.

2.1 Containerisation Technologies

In the container landscape, Docker[1] represents the leading technology for container runtimes (Sysdig, 2019). It provides a set of technologies for building and running containerised applications. Furthermore, DockerHub[2] offers a catalogue of Docker images ready to deploy, which allows users to share their work. Among competitors, containerd[3], CRI-O [4], and Containerizer[5] are worth mentioning.

In recent times, container-based cluster solutions have gained increasing popularity for deploying containers. Some of these solutions further support the orchestration of containers, providing greater scalability, improved reliability, and a sophisticated management interface. Kubernetes[6] currently represents the most widespread ecosystem for managing containerised workloads. With its wide ecosystem, it facilitates both declarative configuration and automation of container clusters. Docker Swarm[7] offers a native solution for cluster management to be integrated into Docker. Mesos[8] is an open-source project to manage computer clusters backed by Apache Software Foundation. It natively supports Docker containers and may be used in conjunction with Marathon[9], a platform for container orchestration.

Some of the most renowned cloud providers, such as Amazon AWS, Microsoft Azure, and Google Cloud have built-in services to operate containers and clusters. In most cases, these built-in services are just ad-hoc implementations of the aforementioned technologies. OpenStack[10] represents an open-source alternative to control large pools of resources. In order to support container orchestration, it uses the *Heat*[11] and *Magnum*[12] components. The first is a service to orchestrate composite Cloud applications, which is required for Magnum to work properly. The latter allows clustered container platforms (Kubernetes, Mesos, Swarm) to interoperate with other OpenStack components through differentiated APIs.

The wide choice of technologies and providers gives developers many options in terms of flexibility, reliability, and costs. However, all these services are neither interchangeable nor interoperable. Switching from a service (or a platform) to another requires several manual operations to be performed, and the learning curve owing to the new tools functioning might be non-trivial. These shortcomings have led to the development of systems to automate deployment and management operations, able to manage the interface with multiple container technologies, clusters and Cloud providers.

2.2 The TOSCA Specification

Research community has focused on approaches using standardised languages to specify the topology and the management plans for Cloud applications. In this regard, TOSCA represents a notable contribution to the development of Cloud standards, since it allows to describe multi-tier applications and their life-cycle management in a modular and portable fashion (Bellendorf and Mann, 2018).

TOSCA is a standard designed by OASIS to enable the portability of Cloud applications and the related IT services (OASIS, 2013). This specification

[1] https://www.docker.com/
[2] https://hub.docker.com/
[3] https://containerd.io/
[4] https://cri-o.io/
[5] http://mesos.apache.org/documentation/latest/mesos-containerizer/
[6] https://kubernetes.io/
[7] https://docs.docker.com/engine/swarm/
[8] http://mesos.apache.org/
[9] https://mesosphere.github.io/marathon/
[10] https://www.openstack.org/
[11] https://wiki.openstack.org/wiki/Heat
[12] https://wiki.openstack.org/wiki/Magnum

permits to describe the structure of a Cloud application as a *service template*, which is composed of a topology template and the types needed to build such a template.

The topology template is a typed directed graph, whose nodes (called *node templates*) model the application components, and edges (called *relationship templates*) model the relations occurring among such components. Each node of a topology can also contain several information such as the corresponding component's *requirements*, the operations to manage it (*interfaces*), the *attributes* and the *properties* it features, the *capabilities* it provides, and the software *artifacts* it uses.

TOSCA supports the deployment and management of applications in two different flavours: *imperative processing* and *declarative processing*. The imperative processing requires that all needed management logic is contained in the Cloud Service Archive (CSAR). Management plans are typically implemented using workflow languages, such as BPMN[13] or BPEL[14]. The declarative processing shifts management logic from plans to runtime. TOSCA runtime engines automatically infer the corresponding logic by interpreting the application topology template. The set of provided management functionalities depends on the corresponding runtime and is not standardised by the TOSCA specification. OpenTOSCA (Binz et al., 2014) is a famous open-source TOSCA runtime environment.

TOSCA Simple Profile is an isomorphic rendering of a subset of the TOSCA specification in the YAML language (OASIS, 2019). It defines a few normative workflows that are used to operate a topology and specifies how they are declaratively generated: deploy, undeploy, scaling-workflows and auto-healing workflows. Imperative workflows can still be used for complex use-cases that cannot be solved in declarative workflows. However, they provide less reusability as they are defined for a specific topology rather than being dynamically generated based on the topology content. The work described in this paper heavily grounds on the TOSCA standard and, specifically, on TOSCA Simple Profile. This provides convenient definitions for container nodes. The *tosca.nodes.Container.Runtime* type represents the virtualised environment where containers run. The *tosca.nodes.Container.Application* type represents an application that uses container-level virtualisation.

Besides container types, the TOSCA Simple Profile specification provides other useful tools for the description of containerised applications, such as the *Repository Definition*, which can be exploited to define internal or external repositories for pulling container images, the non-normative *tosca.artifacts.Deployment.Image.Container.Docker* type for Docker images, and the *Configure* step in *Standard interface* node life-cycle, which allows to define post-deployment configuration operations or scripts to execute.

3 RELATED WORK

Several research and business-oriented projects have exploited the TOSCA standard for container orchestration.

Cloudify[15] delivers container orchestration integrating multiple technologies and providers. While it offers graphical tools for sketching and modelling an application, its data format is based on the TOSCA standard. Alien4Cloud[16] is an open-source platform which provides a TOSCA nearly-normative set of types for Docker support. Kubernetes and Mesos orchestrators are available through additional plugins. Both the above-mentioned works implement the interoperability different clusters and providers defining complex sets of nodes, which are specific to the technologies used. Moreover, their TOSCA implementations reckon on Domain-Specific Languages (DSLs) which, despite sharing the TOSCA template structure, do not use the node type hierarchy defined in the standard. With respect to Cloudify, the approach discussed in this paper focuses on TOSCA-compliant application descriptions, making no prior assumptions regarding the technology stack to be established.

In (Kiss et al., 2019) the authors present MiCADO, an orchestration framework which guarantees out-of-the-box reliable orchestration, by working closely with Swarm and Kubernetes. Unlike the precedent approaches, MiCADO does not overturn the TOSCA standard nodes, but the cluster orchestrator is still hardcoded in the *Interface* section of each node of the topology.

TosKer (Brogi et al., 2018) presents an approach which leverages on the TOSCA standard for the deployment of Docker-based applications. This work claims to be able to generalise its strategy to cluster systems, but neither a proof nor an explanation of how to deal with the differences between clustered and non-clustered scenarios is given. TosKer approach is very different from the one proposed in this paper,

[13] http://www.bpmn.org/

[14] https://www.oasis-open.org/committees/wsbpel/

[15] http://cloudify.co/

[16] https://alien4cloud.github.io/

since it does not provide any automatic provisioning of the deployment plan and it is based on the redefinition of several nodes of the TOSCA standard.

In (Kehrer and Blochinger, 2018) the authors propose a two-phase deployment method based on the TOSCA standard. They provide a good integration with Mesos and Marathon, but they do not either support other containerised clusters or furnish automation for the deployment of the cluster.

In summary, all the current works achieve container cluster interoperability, or partial interoperability, either associating platform-specific information to the nodes of the topology template or redefining the TOSCA hierarchy of nodes. Thus, in order to work with the above mentioned frameworks, it is necessary to know in advance both the technological stack and the framework-specific nodes to use.

The work we propose differentiates from all existing works in its goals, which are: enabling high interoperability between different technologies and providers; providing a standard-compliant approach, with no overturning of the standard-defined types and no prior assumptions about the technology stack to be established; and adopting the principle of separation of concerns between the topology of the application and its orchestration.

4 ACHIEVING CLUSTER INTEROPERABILITY

In this section, we first analyse existing swarm services to provide interoperability between multiple container cluster technologies. Then, the strategy adopted to describe the topology of containerised applications operating on top of multiple cluster platforms is presented.

4.1 Analysis of Cluster Orchestrators

To operate on top of different cluster platforms, a common specification model, compatible across diverse technologies, is required. To develop such a model, we analysed three of the most popular cluster orchestrators: Docker Swarm, Kubernetes and Mesos + Marathon. Our analysis focused on highlighting similarities and points of contrast within the aspects that affect the deployment of containers. We found that all the three platforms implement the main features for container orchestration in similar ways. For example, some entities and services represent identical concepts, even though they are named differently. The results of the comparison are available in Table 1.

Application Specification indicates the method to describe the scenario to deploy, i.e., specification formats and languages. *Deployment Unit* refers to the atomic deployable entity, which is managed by the cluster in terms of scalability and availability. *Container* and *Cluster* indicate the names used for container entities and for clusters of physical machines. *Volume Management* describes the strategies to manage the attachment of storage entities and *Networking Management* illustrates how to establish internal and external connections. *Configuration Operations* present methods to execute post-deployment configuration operations on containers.

Firstly, we identified the most important features for deploying and initialising containerised applications. Then, for each of these features, we found strategies leading to similar results in all the analysed orchestrators. This information can be found in the

Table 1: A comparison of how features are implemented in Docker Swarm, Kubernetes and Mesos + Marathon.

	Docker Swarm	Kubernetes	Mesos + Marathon
Application Specification	Docker Compose YAML	YAML format	JSON format
Deployment Unit	Service	Pod	Pod
Container	Container	Container	Task
Cluster	Swarm	Cluster	Cluster
Volume Management	Volumes can be attached to Services or be automatically created according to the volume specification on the service.	PersistentVolumes can be directly attached to Pods or may be automatically provisioned.	The appropriate amount of disk space is implicitly reserved, according to specification.
Networking Management	Overlay networks manage communications among the Docker daemons participating in the swarm.	Services provide networking, granting a way to access Pods.	Containers of each pod instance can share a network namespace, and communicate over a VLAN or private network.
Configuration Operations	It is possible to execute commands directly on the service. (eg *docker exec*)	It is possible to execute commands directly on the container (eg *kubectl exec*)	It is possible to execute commands directly on the task. (eg *dcos task exec*)

rows of the Table.

From this analysis, many similarities emerged between the three platforms. All of them allow to specify the desired application using a tree-like data model within portable formats, such as JSON and YAML. Furthermore, all the orchestrators map resources in similar ways for deployment units, containers, and clusters, where the main difference is given by the naming conventions.

With regard to volume and networking management, different platforms implement different strategies. However, all the volume management approaches share the possibility to delegate the provisioning of volumes to the platform, taking for granted that volume properties are indicated in the application specification. As for networking, each of the software grants accessibility to deployment units and containers, both within and outside the cluster, although they manage it in different ways. Finally, all the platforms allow to execute configuration commands on the deployed instances, by accessing them directly.

The analysis of container cluster interoperability laid the groundwork for a unified approach. This is further explored in the next section, where the common specification format and the interfaces to the different cluster orchestrators are discussed.

4.2 Application Description

One of the key aspects of this work is the development of a standard-based approach for the topology description of containerised applications, which leverages the TOSCA standard.

As discussed in Section 2.2, TOSCA Simple Profile includes several node types for container-based application topologies. According to the analysis in Table 1, we mapped TOSCA Container Runtime to Deployment Unit entities and TOSCA Container Application to containers. This allows to easily describe containerised applications within a cluster in terms of nodes. However, we found that using the plain TOSCA Container Application would flatten the node hierarchy present in the Simple Profile specification, removing the possibility to assign meaningful roles to each node in the topology (e.g. Database, WebServer).

```
tosca.nodes.Container.Application:
  derived_from: tosca.nodes.Root
  requirements:
    - host:
        capability: tosca.capabilities.Compute
        node: tosca.nodes.Container.Runtime
        relationship: tosca.relationships.HostedOn
    - storage:
        capability: tosca.capabilities.Storage
    - network:
        capability: tosca.capabilities.Endpoint
```

Listing 1: TOSCA Container Application node.

For the sake of clarity, Listing 1 shows the TOSCA Container Application node which represents a generic container-based application. Other than hosting, storage and network requirements, no properties are defined. Besides, it directly derives from the root node as all other TOSCA base node types do. If, on the one hand, this allows to have consistent definitions for basic requirements, capabilities and lifecycle interfaces, on the other one, customisation is only possible by type extension.

```
tosca.nodes.Database:
  derived_from: tosca.nodes.Root
  properties:
    name:
      type: string
      description: the logical name of the database
    port:
      type: integer
      description: >
        the port the underlying database service
        will listen to for data
    user:
      type: string
      description: >
        the user account name for DB administration
      required: false
    password:
      type: string
      description: >
        the password for the DB user account
      required: false
  requirements:
    - host:
        capability: tosca.capabilities.Compute
        node: tosca.nodes.DBMS
        relationship: tosca.relationships.HostedOn
  capabilities:
    database_endpoint:
      type: tosca.capabilities.Endpoint.Database
```

Listing 2: TOSCA Database node.

```
tosca.nodes.Container.Database:
    derived_from: tosca.nodes.Container.Application
    description: >
      TOSCA Container for Databases which employs
      the same capabilities and properties of the
      tosca.nodes.Database but which extends from
      the Container.Application node_type
    properties:
      user:
        required: false
        type: string
        description: >
          User account name for DB administration
      port:
        required: false
        type: integer
        description: >
          The port the database service will use
          to listen for incoming data and requests.
      name:
        required: false
        type: string
        description: >
          The name of the database.
      password:
        required: false
        type: string
        description: >
          The password for the DB user account
    capabilities:
      database_endpoint:
        type: tosca.capabilities.Endpoint.Database
```

Listing 3: TOSCA Container Database node.

As a result, we extended the TOSCA Simple Profile hierarchy for containers, by deriving from the

TOSCA Container Application type and defining the same properties and capabilities that are present in each of the corresponding TOSCA node in the standard. Listing 2 and Listing 3 further explain our methodology, describing, by way of example, the TOSCA Database node and the TOSCA Container Database node respectively.

While using the plain TOSCA Container Application type would still allow to deploy a scenario in our framework, we believe that preserving a node typing system would make the specification more descriptive. Moreover, this choice enables the use of the standard-defined typed relationships (i.e. ConnectsTo, DependsOn, HostedOn, ...) between different types of container nodes.

Another resource mapping was required for managing Volumes. TOSCA Simple Profile provides useful Storage node types for representing storage resources, such as *tosca.nodes.Storage.BlockStorage*. We mapped TOSCA Block Storage to volumes. Each volume should be connected to the respective container using the standard-defined relationship *tosca.relationships.AttachesTo*. TOSCA AttachesTo already defines the *location* property which is of primary importance for containers, since it allows to define the mount path of a volume.

Networking management did not need any additional specification. Cluster networks may be arranged using the *port* property of a node and analysing its relationships with the other nodes in the topology.

5 SYSTEM DESIGN

The aim of this work is to design a TOSCA Orchestrator for the deployment of containerised applications on clusters. The Orchestrator should also be able to interface with several Cloud providers and a variety of container technologies. The main features of the framework are described in the following subsections.

5.1 Framework Architecture

Starting from the Cloud application description, the framework is capable of devising and orchestrating the workflow of the provisioning operations to execute. Along with the application description, several application properties may be provided using the dashboard tool. This is the main endpoint in order to interact with the framework since it allows to configure and start the deployment process.

Firstly, the dashboard allows users to either sketch the topology of their desired applications, using graphical modelling tools, or upload and validate previously worked application descriptions. Then, it is possible to deploy the uploaded applications, providing many configuration parameters, such as the target Cloud provider or the cluster technology to use for containers. At a later stage, the dashboard can also be used to display information about the deployment status and debug information.

We have designed and implemented a TOSCA Orchestrator which transforms the YAML model into an equivalent BPMN model, which is fed to a BPMN engine that instantiates and coordinates the related process. The process puts in force all the provisioning activities needed to build up the application stack. The overall provisioning scenario is depicted in Figure 1.

For each framework service, multiple implementations can be provided for the different supported Cloud providers. All the services are offered within the framework through an Enterprise Service Bus (ESB). The original work provides two categories of provisioning services that need to be integrated in the ESB: *Cloud Resource Services* and *Packet-based Services*. This work requires an additional category, *Container Cluster Services*, which includes functionalities to deploy applications on cluster platforms. In order to integrate all the mentioned services in the ESB, we deploy a layer of *Service Connectors* which are responsible for connecting requests coming from the Provisioning Tasks with the Provisioning Services. Service Connectors allow to achieve service location transparency and loose coupling between Provisioning BPMN plans (orchestrated by the Process Engine) and Provisioning Services.

The *Service Registry* is responsible for the registration and discovery of Connectors. The *Service Broker* is in charge of taking care of the requests coming from the Tasks. *Cloud Service Connectors* implement interactions with Cloud Providers for the allocation of Cloud resources. For each service type, a specific Connector needs to be implemented. For instance, *Instantiate Cluster* represents the generic Connector interface to the instantiation of Cloud resources of "Container Cluster" type. All concrete Connectors to real Cloud services (AWS, OpenStack, Azure, etc.) must implement the Instantiate Cluster interface. Likewise, *Instantiate VM* is the generic Connector interface to "Virtual Machine" services, which concrete Connectors to real services in the Cloud must implement.

Packet-based Service Connectors are meant to implement interactions with all service providers that provide packet-based applications. When the YAML-to-BPMN conversion takes place, three types of BPMN service tasks might be generated: "Create",

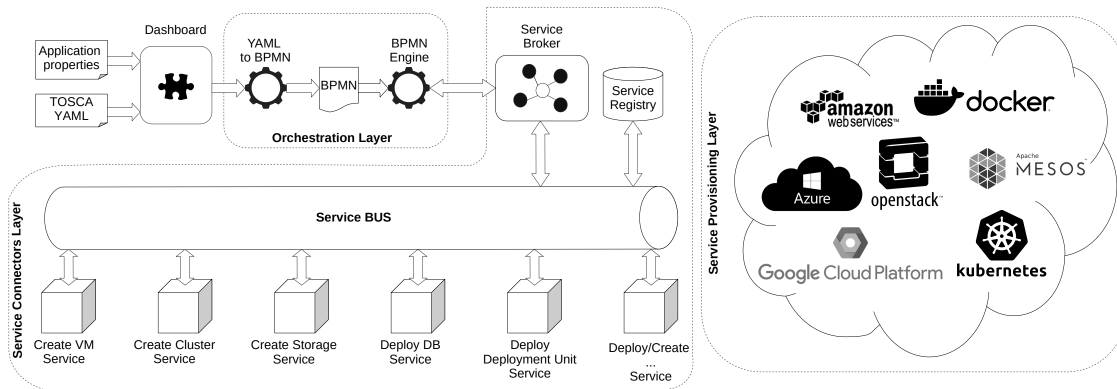

Figure 1: Overview of the Provisioning Scenario, Showing the Different Layers of the Framework.

"Configure" and "Start". To each of these tasks corresponds a generic connector interface (Create, Configure and Start). These interfaces are then extended in order to manage several types of applications (DBMS, Web Servers, etc.). The latter are the ones that concrete Connectors must implement in order to interact with real packet-based application providers.

In this work we focus on container-based applications which use container cluster technologies. The TOSCA operations for container orchestration are different from resource and package operations, and cluster technologies frequently perform management operations that are relieved from the framework. Thus, the orchestration process for Deployment Units will be discussed later in this paper.

5.2 YAML Parsing

In our framework, the first step towards the deployment orchestration is the YAML processing. The Parser software component is widely based on the OpenStack parser[17] for TOSCA Simple Profile in YAML, a Python project licensed under Apache 2.0. The Parser builds an in-memory graph which keeps track of all nodes and dependency relationships between them in the TOSCA template.

We extended the Parser features to adapt it for containerised applications. The new module developed for the Parser is able to identify, analyse and output Deployment Units specification in a suitable format for the BPMN plans. A bottom-up approach has been used. Starting from a Container Runtime, it identifies the Deployment Unit and recursively find all the containers stacked upon it and their dependencies, making a clear distinction between volume dependencies, which bind a storage volume to a container, and external dependencies, which bind a container to another container hosted either on the same Deployment Unit or on a different one.

Each volume must be linked to its corresponding container using the "AttachesTo" relationship. It is important to specify the *location* parameter, which would serve as the mount path for the volume. This allows the Parser to correctly associate each volume to its container. External dependencies are identified and output by the Parser, since they would be used to setup Networking for each Deployment Unit.

Finally, the Parser produces BPMN data objects which are provided as data inputs for the BPMN plans.

5.3 BPMN Plans

The BPMN plans in our platform rework the strategy adopted in (Calcaterra et al., 2018). In Figure 2 the overall service provision workflow is depicted. The diagram is composed of a parallel multi-instance sub-process, i.e., a set of sub-processes (called "Instantiate Node") each processing a TOSCA node in a parallel fashion. Originally, a TOSCA node was either a cloud resource or a software package. In this work we expanded the BPMN Plans for our purpose, modelling a workflow path for deployment unit nodes.

In Figure 3 the detailed workflow for a deployment unit node is depicted. The top pool called "Node Instance" represents the pool of all instances of either the "create cloud resource" sub-process or the "create deployment unit" sub-process, which are running in parallel to the "create deployment unit" sub-process being analysed. The bottom pool called "Container Cluster Service Connectors" represents the pool of the software connectors deployed in the ESB. In the middle pool, the sequence of tasks carried out to create and instantiate a deployment unit are depicted. Interactions of the middle pool with the "Node Instance" pool represent points of synchronization between the

[17] https://wiki.openstack.org/wiki/TOSCA-Parser

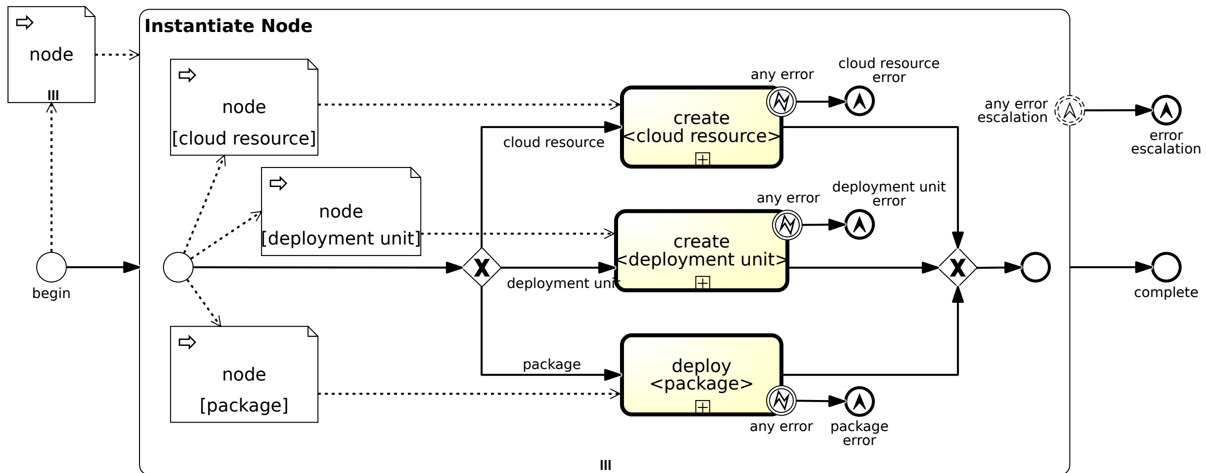

Figure 2: Instantiate Node Overall Workflow.

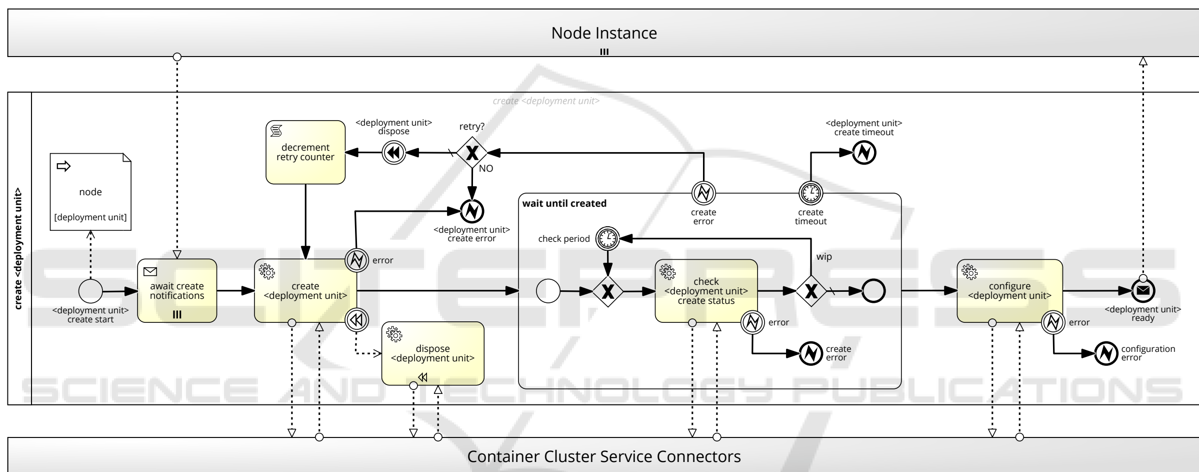

Figure 3: Node Deployment Unit Workflow.

multiple installation instances, that may be involved in a provision process.

The creation of a deployment unit starts with a task that awaits notifications coming from the preceding sub-processes, which may consist of the "create cloud resource" sub-process for the creation of the cluster, in case this was not instantiated before, or other "create deployment unit" sub-processes. A service task will then trigger the actual instantiation by invoking the appropriate Connector on the ESB. Here, if a fault occurs, it is immediately caught and the whole sub-process is cancelled. Following the path up to the parent process, an escalation is engaged. If the creation step is successful, a "wait-until-created" sub-process is activated.

Checks on the status are iterated until the cluster platform returns an "healthy status" for the deployed instance. The "check deployment unit create status" service task is committed to invoke the Connector on the ESB to check the status on the selected swarm service. The deployment unit's status is strongly dependent on the hosted containers' status. However, container cluster platforms automatically manage the life-cycle of containers, then the check is executed to detect errors which are strictly related to deployment units' resources.

Checking periods are configurable, so is the timeout put on the boundary of the sub-process. An error event is thrown either when the timeout has expired or when an explicit error has been signalled in response to a status check call. In the former case, the escalation is immediately triggered; in the latter case, an external loop will lead the system to autonomously re-run the whole deployment unit creation sub-process a configurable number of times, before yielding and eventually triggering an escalation event. Moreover, a compensation mechanism ("dispose deployment unit" task) allows to dispose of the

deployment unit, whenever a fault has occurred.

Then the "configure deployment unit" task may be invoked to execute potential configuration operations on the deployed containers. When the workflow successfully reaches the end, a notification is sent. Otherwise, the occurred faults are caught and handled via escalation.

5.4 Service Connectors

Different kinds of Service Connectors serve the cause of container orchestration in our framework. The first category contains the services related to the different Cloud providers, such as AWS, Azure, Google Cloud, and OpenStack. In particular, the cluster for deploying the scenario should be provisioned and the parameters for authenticating on the cluster should be provided to the ESB for future operations.

The second category of services is related to the container cluster platforms, namely Docker Swarm, Kubernetes and Mesos. After creating the cluster, the ESB should be able to authenticate and communicate with the cluster for starting the operations which realise the deployment of the scenario. These connectors also perform a translation from the parsed topology to the specific format of the container cluster platform.

6 PROTOTYPE IMPLEMENTATION AND TESTS

In this section, we discuss the implementation of the framework in more detail and corroborate the working behaviour of our software with a test on a simple real-world scenario. This would be a containerised version of a WordPress (WP) scenario including two Deployment Units, MySQL and WordPress, which are both stacked with a Volume and a Container. The scenario is depicted in Figure 4, by using TOSCA standard notation.

In the WP scenario, container images are Docker images which need to be pulled from the DockerHub repository, as specified in the template. The TOSCA Artifact image fills the *implementation* parameter for the Create step in the container life-cycle. Any environment variable for the Docker image should be given as an input of the implementation in the Create section of the containers. For being correctly parsed, the environment variables should have the same names that are indicated in the DockerHub instructions for the image. Another parameter that can be specified in the Create inputs is the port. Otherwise, a port would be automatically chosen for the service by the container orchestrator.

In Listing 4, the TOSCA Simple Profile description of the types and the templates used for the MySQL deployment unit is provided as an example. The description was drafted according to the principles defined in Section 4.2.

With regard to the BPMN plans execution, the deployment unit workflow has to be processed two times. The first time the instance to be created is the MySQL deployment, while the second unit to be processed corresponds to the WordPress deployment. We used Flowable[18], which is a Java based open-source business process engine, for the implementation of the BPMN workflow processing.

```
node_types:
  tosca.nodes.Container.Database.MySQL:
    description: >
      MySQL container from the Docker Hub repository
    derived_from: tosca.nodes.Container.Database
    requirements:
      - volume:
          capability: tosca.capabilities.Attachment
          relationship: tosca.relationships.AttachesTo

relationship_templates:
  tosca.relationships.MySQLAttachesToVolume:
    type: tosca.relationships.AttachesTo
    properties:
      location: { get_input: mysql_location }

node_templates:
  mysql_container:
    type: tosca.nodes.Container.Database.MySQL
    requirements:
      - host: mysql_deployment_unit
      - volume:
          node: mysql_volume
          relationship: tosca.relationships.
              MySQLAttachesToVolume
    artifacts:
      mysql_image:
        file: mysql:5.7
        type: tosca.artifacts.Deployment.Image.Container.
            Docker
        repository: docker_hub
    properties:
      port: { get_input: mysql_port }
      password: { get_input: mysql_root_pwd }
    interfaces:
      Standard:
        create:
          implementation: mysql_image
          inputs:
            port: {get_property:[SELF, port]}
            mysql_root_password: {get_property:[SELF,
                password]}
  mysql_volume:
    type: tosca.nodes.BlockStorage
    properties:
      size: { get_input: mysql_volume_size}
  mysql_deployment_unit:
    type: tosca.nodes.Container.Runtime
```

Listing 4: MySQL deployment unit specification.

We tested the WP scenario with an OpenStack Cloud provider, using Kubernetes as the container cluster platform. A local cluster consisting of two identical off-the-shelf PCs was considered in order

[18]https://www.flowable.org/

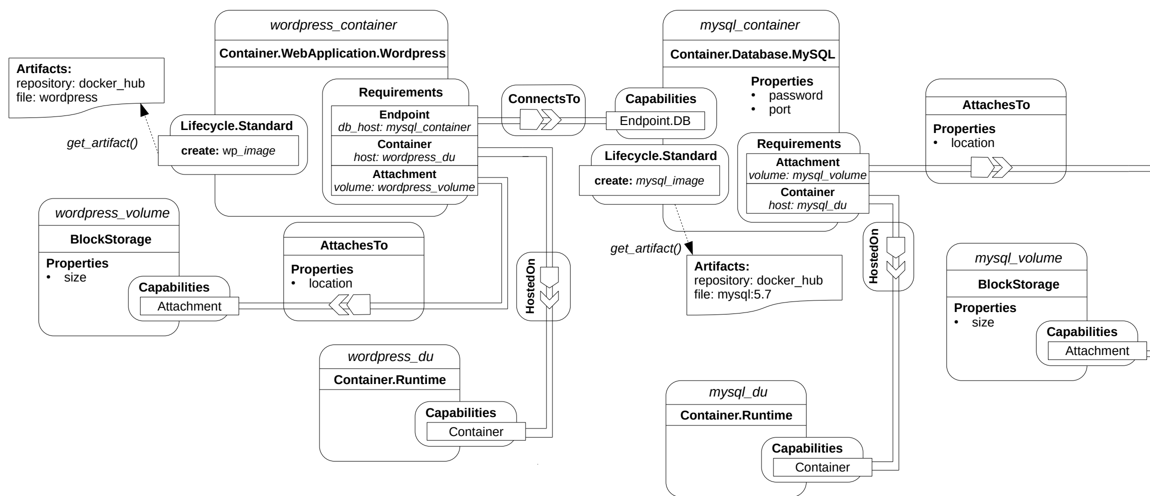

Figure 4: The WordPress Application Topology, Described Using TOSCA Specification.

to create a minimal OpenStack set-up, i.e., Controller node and Compute node. The former also runs Heat and Magnum services. Both nodes run Ubuntu Server x86-64 Linux distributions. The Service Connectors for OpenStack and Kubernetes were implemented complementing Eclipse Vert.x[19], a Java toolkit for event-driven applications, with OpenStack4J[20], for the creation of the cluster, and the official Kubernetes Java client, for the deployment of the Deployment Units. Overall, the scenario was correctly provisioned, returning a working WP application.

7 CONCLUSIONS

The automated provisioning of complex Cloud applications has become a key factor for the competitiveness of Cloud providers. The ever-increasing usage of Cloud container technologies shows the importance of their management and orchestration also in this context. Organisations do indeed package applications in containers and need to orchestrate multiple containers across multiple Cloud providers.

In this work, starting from a previously designed Cloud orchestration and provisioning framework, we extended it in order to allow the deployment and orchestration of containerised applications. The main effort has been devoted to provide interoperability between multiple container cluster technologies: a strategy to describe the topology of containerised applications operating on top of multiple cluster platforms has been also presented. The developed prototype and the simple test presented showed the viability of the approach.

Future work will include the development and testing on top of different container-based cluster platforms using more complex scenarios, which feature multiple layers of resources, in order to further validate our system.

[19]https://vertx.io/

[20]http://www.openstack4j.com/

REFERENCES

Bellendorf, J. and Mann, Z. Á. (2018). Cloud Topology and Orchestration Using TOSCA: A Systematic Literature Review. In Kritikos, K., Plebani, P., and de Paoli, F., editors, *Service-Oriented and Cloud Computing*, pages 207–215. Springer International Publishing.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). TOSCA: Portable Automated Deployment and Management of Cloud Applications. *Advanced Web Services*, pages 527–549.

Brogi, A., Rinaldi, L., and Soldani, J. (2018). Tosker: A synergy between tosca and docker for orchestrating multicomponent applications. *Software: Practice and Experience*, 48(11):2061–2079.

Calcaterra, D., Cartelli, V., Di Modica, G., and Tomarchio, O. (2017). Combining TOSCA and BPMN to Enable Automated Cloud Service Provisioning. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 159–168, Porto (Portugal).

Calcaterra, D., Cartelli, V., Di Modica, G., and Tomarchio, O. (2018). Exploiting BPMN features to design a fault-aware TOSCA orchestrator. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 533–540, Funchal-Madeira (Portugal).

Dikaiakos, M. D., Katsaros, D., Mehra, P., Pallis, G., and Vakali, A. (2009). Cloud Computing: Distributed

Internet Computing for IT and Scientific Research. *IEEE Internet Computing*, 13(5):10–13.

Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B. (2015). Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In *8th International Conference on Cloud Computing*, pages 621–628.

Kehrer, S. and Blochinger, W. (2018). TOSCA-based container orchestration on Mesos. *Computer Science - Research and Development*, 33(3):305–316.

Kiss, T., Kacsuk, P., Kovacs, J., Rakoczi, B., Hajnal, A., Farkas, A., Gesmier, G., and Terstyanszky, G. (2019). MiCADO—Microservice-based Cloud Application-level Dynamic Orchestrator. *Future Generation Computer Systems*, 94:937 – 946.

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., and Ghalsasi, A. (2011). Cloud computing — the business perspective. *Decision Support Systems*, 51(1):176 – 189.

OASIS (2013). Topology and Orchestration Specification for Cloud Applications Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html. Last accessed: 2019-12-23.

OASIS (2019). TOSCA Simple Profile in YAML Version 1.2. https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html. Last accessed: 2019-12-23.

Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31.

Petcu, D. and Vasilakos, A. (2014). Portability in Clouds: Approaches and Research Opportunities. *Scalable Computing: Practice and Experience*, 15(3):251–270.

Ruan, B., Huang, H., Wu, S., and Jin, H. (2016). A Performance Study of Containers in Cloud Environment. In Wang, G., Han, Y., and Martínez Pérez, G., editors, *Advances in Services Computing*, pages 343–356, Cham. Springer International Publishing.

Sysdig (2019). Sysdig 2019 container usage report. https://sysdig.com/blog/sysdig-2019-container-usage-report/. Last accessed: 2019-12-23.

Weerasiri, D., Barukh, M. C., Benatallah, B., Sheng, Q. Z., and Ranjan, R. (2017). A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.*, 50(2):26:1–26:41.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS