Cluster-Agnostic Orchestration of Containerised Applications

Domenico Calcaterra, Giuseppe Di Modica, Pietro Mazzaglia, and Orazio Tomarchio

Department of Electrical, Electronic and Computer Engineering, University of Catania, V.le A. Doria 6, 95125 Catania, Italy {domenico.calcaterra,giuseppe.dimodica,orazio.tomarchio}@unict.it

Abstract. The complexity of managing cloud applications’ life-cycle increases with the widening of the cloud landscape, as new IT players gain market share. Cloud orchestration frameworks promise to handle such complexity offering user-friendly management tools that help customers to transparently deal with portability and interoperability issues, by hiding away the heterogeneity of the cloud providers’ proprietary interfaces. Regarding the provisioning of cloud-enabled applications, the containerisation paradigm, along with the related micro-services technology, has managed to deliver the portability promise. While most of cloud orchestration frameworks support container-based cluster technologies, a standard-based approach to describe containerised applications still lacks. In this work, we propose TORCH, a TOSCA-based cloud orchestrator capable of interfacing to theoretically any container run-time software by leveraging a standard-aligned and easy-to-use language to describe application requirements. Validation tests run on a small-scale test-bed prove the viability of the proposed solution.

Keywords: Containerised applications · Deployment orchestration · Cloud provisioning · TOSCA · BPMN

1 Introduction

Nowadays, due to several benefits in terms of availability, scalability and costs, more and more applications are deployed in a cloud environment. However, the deployment and management of complex applications, which are built out of many components with different technical dependencies and constraints and make use of different platform components, has become a complicated and error-prone task [11]. Several tools help to simplify this task but, in a scenario moving towards multi-cloud environments, the lack of interoperability between different tools and platforms is a major limitation [31]. Thus, the need to describe the topology and life-cycle management tasks of cloud applications at a higher abstraction level, by using standardised formats, becomes a fundamental capability of a modern cloud orchestration tool [36]. In this context, the OASIS
consortium has proposed TOSCA (Topology and Orchestration Specification for Cloud Applications) [25], a specification that aims to standardise the definition of application topologies for cloud orchestration. As such, it enables customers to define the topology of their cloud applications in a reusable manner and deploy them on TOSCA compliant clouds [4].

In this scenario, over the course of the past few years, several cloud platforms have shifted from managing VMs to managing more fine-grained units of work, from containers and services, through microservices and even “functions” (FaaS paradigm) [10,33]. Containers, in particular, 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.

Therefore, modern orchestrators should also take into account these new types of resources and services while providing their management tasks [35]. Our work just addresses this scenario: in this paper, we extend the work presented in [13] where we proposed TORCH, a framework for the deployment and orchestration of containerised applications. TORCH 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. In this work, we provide further details about the prototype implementation and some performance results obtained from running the framework prototype on a small-scale test-bed.

The remainder of the paper is structured as follows. In Sect. 2, we provide a brief background about container technologies, along with related works concerning container orchestration. In Sect. 3, the TOSCA specification is briefly presented. In Sect. 4, we analyse cluster orchestrators from the interoperability point of view, while in Sect. 5 we present our approach to describe containerised applications in order to operate on top of multiple cluster platforms. In Sect. 6, the design and a prototype implementation of TORCH are presented. Section 7 shows preliminary results of some experiments run on a small-scale test-bed. Finally, Sect. 8 concludes the work.

2 Related Work

In this section, first, we provide a more in-depth background on containers and, then, we present a few business-oriented and research projects exploiting the TOSCA standard for container orchestration.

In the container landscape, Docker [16] represents the leading technology for container runtimes [34]. It provides a set of technologies for building and running containerised applications. Furthermore, Docker Hub\(^1\) offers a catalogue of ready-to-deploy Docker images, which allows users to share their work. Among

\(^1\) https://hub.docker.com/.
competitors, containerd [14], CRI-O [15], and Containerizer [3] are worth mentioning.

Recently, 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 [20] currently represents the most widespread ecosystem to manage containerised workloads, which facilitates both declarative configuration and automation of container clusters. Docker Swarm [17] offers a native solution for cluster management to be integrated into Docker. Mesos [2] is an open-source project to manage computer clusters backed by the Apache Software Foundation. It natively supports Docker containers and may be used in conjunction with Marathon [23], 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 former cluster technologies. OpenStack [28] represents an open-source alternative to control large pools of resources. In order to support container orchestration, it uses the Heat [29] and Magnum [30] components. The first is a service to orchestrate composite cloud applications, while the second allows clustered container platforms (i.e., 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 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.

Cloudify [19] 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 [18] is an open-source platform providing a TOSCA nearly-normative set of types for Docker support, where Kubernetes and Mesos orchestrators are available through additional plugins. Both Cloudify and Alien4Cloud achieve interoperability between different clusters and providers by defining complex sets of TOSCA non-normative nodes, which are specific to the technologies used. Their TOSCA implementations reckon on Domain-Specific Languages that, despite sharing the TOSCA template structure, diverge from the node type hierarchy defined in the standard. With respect to them, TORCH focuses on TOSCA-compliant application descriptions, making no prior assumptions regarding the technology stack to be established.

Apache ARIA TOSCA [1] is an open-source framework offering an implementation of the TOSCA Simple Profile v1.0 specification. Unlike Cloudify and Alien4Cloud, it provides an extension of TOSCA normative types for Docker.
Compared to TORCH, the derived node types lack the possibility to use role-specific containers and no cluster orchestrator is natively supported.

In [22] the authors present MiCADO, an orchestration framework that 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. However, the cluster container platform is still hardcoded in the Interface section of each node of the topology.

TosKer [9] presents an approach that leverages the TOSCA standard for the deployment of Docker-based applications. TosKer approach is very different from the one proposed in this paper: 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. Clustered scenarios are also out of the picture, even though some recent work [6] has been done to deploy applications on top of existing Docker-based container orchestrators.

In [21], 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.

Other approaches worth mentioning are OpenTOSCA [5], Indigo-DataClouds [11,32], and SeaClouds [7,8]. OpenTOSCA is a famous open-source runtime environment for deploying and managing TOSCA applications. Despite supporting the orchestration of Docker containers, it was designed to work with a former, XML-based version of TOSCA [25]. Although authors have provided a converter\(^2\) to work with TOSCA Simple Profile, only a few YAML elements are supported thereby limiting the description of application components.

INDIGO-DataCloud is an open-source data and computing platform for the automatic distribution of applications and/or services over a hybrid and heterogeneous set of IaaS infrastructures. It adopts an extension of TOSCA for describing applications and services, and leverages Docker containers as the preferred underlying technology to encapsulate user applications. INDIGO-DataCloud also provides a good integration with Mesos and Marathon/Chronos, but no support for other containerised clusters is provided.

SeaClouds is an open-source middleware solution for deploying and managing multi-component applications on heterogeneous clouds. While SeaClouds fully supports TOSCA, it lacks support for Docker containers making it not suitable for orchestrating multi-component applications including Docker containers.

In summary, all the current works achieve container cluster interoperability, or partial interoperability, by either associating platform-specific information to the nodes of the topology template or by redefining the TOSCA standard nodes. In order to work with such 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-

\(^2\) https://github.com/CloudCycle2/YAML_Transformer.
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.

3 The TOSCA Specification

Research community has focused on approaches using standardised languages to specify the topology and 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 [4].

TOSCA is a standard designed by OASIS to enable the portability of cloud applications and the related IT services [25]. This specification 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 some 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, the software artifacts it uses, and the policies applied to it. Inter-node dependencies associate the requirements of a node with the capabilities featured by other nodes.

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 [27] or BPEL [24]. The declarative processing shifts management logic from plans to runtime. TOSCA runtime engines automatically infer the corresponding logic by interpreting the application topology template. This requires a precise definition of the semantics of nodes and relations based on well-defined Node Types and Relationship Types. The set of provided management functionalities depends on the corresponding runtime and is not standardised by the TOSCA specification.

TOSCA Simple Profile is an isomorphic rendering of a subset of the TOSCA specification in the YAML language [26]. 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.
By way of illustration, Listing 1.1 shows a sample template to deploy a MySQL DBMS on a server and create a custom database content on top of it. The `mysql` node template (of type `tosca.nodes.DBMS.MySQL`) is related to the `db_server` node template (of type `tosca.nodes.Compute`) via the requirements section to indicate which compute resource (of type `tosca.nodes.Compute`) MySQL is to be hosted on. Specifically, the `host` requirement is fulfilled by referencing the `db_server` node template. The `my_db` node template (of type `tosca.nodes.Database.MySQL`) represents an actual MySQL database instance managed by a MySQL DBMS installation. The requirements section of the `my_db` node template stresses that the database is to be hosted on a MySQL DBMS node template named `mysql`. In the artifacts section of the `my_db` node template, there is an artifact definition named `db_content`, representing a text file that will be used to add content to the database as part of the `create` life-cycle operation.

The logical diagram for this sample template is depicted in Fig. 1.

The work described in this paper grounds on the TOSCA standard and, specifically, on TOSCA Simple Profile which 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 resources 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.

4 Analysis of Cluster Orchestrators

In this section, we analyse existing swarm services to provide interoperability between multiple container cluster technologies. 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 differences within the aspects that affect the deployment of containers. We found that all the three platforms implement the main features for container orchestration in a similar fashion. For instance, some entities and services represent identical concepts, even though they are named differently. The results of the comparison are available in Table 1.
Table 1. A comparison of how features are implemented in Docker Swarm, Kubernetes and Mesos + Marathon [13].

| Feature                        | 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 (e.g. docker exec) | It is possible to execute commands directly on the container (e.g. kubectl exec) | It is possible to execute commands directly on the task (e.g. dcos task exec) |

The field 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 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 for deployment units, containers, and clusters in a similar manner, 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.

5 Achieving Cluster Interoperability

In this section, we present the strategy adopted to describe the topology of containerised applications operating on top of multiple cluster platforms, which leverages the TOSCA standard.

As discussed in Sect. 3, 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 specification, removing the possibility to assign meaningful roles to each node in the topology (e.g. Database, WebServer).

For the sake of clarity, Listing 1.2 shows the TOSCA Container Application node which represents a generic container-based application. Apart from 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. On the one hand, this allows to have consistent definitions for basic requirements, capabilities and life-cycle interfaces; on the other hand, customisation is only viable by type extension.

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 types in the standard. Listing 1.3 and Listing 1.4 further explain our methodology, describing, by way of example, the TOSCA Database node and the TOSCA Container Database node respectively.

```xml
<node type="tosca.nodes.Container.Application">
  <derived_from>tosca.nodes.Root</derived_from>
  <requirements>
    <host>
      <capability>tosca.capabilities.Compute</capability>
      <node>tosca.nodes.Container.Runtime</node>
      <relationship>tosca.relationships.HostedOn</relationship>
    </host>
    <storage>
      <capability>tosca.capabilities.Storage</capability>
    </storage>
    <network>
      <capability>tosca.capabilities.Endpoint</capability>
    </network>
  </requirements>
</node>
```

**Listing 1.2.** TOSCA Container Application node [13].
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
  password:
    type: string
description: >
the password for the DB user account
requirements:
  - host:
    capability: tosca.capabilities.Compute
    node: tosca.nodes.DBMS
    relationship: tosca.relationships.HostedOn
  capabilities:
    database_endpoint:
      type: tosca.capabilities.Endpoint.Database

Listing 1.3. TOSCA Database node [13].

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 1.4. TOSCA Container Database node [13].
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 nodes 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.

6 System Design and Implementation

The aim of this work is to design and implement 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 further in the following subsections.

6.1 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 can be provided using the Dashboard tool. This is the main endpoint 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 Fig. 2.

For each framework service, multiple implementations can be provided for the supported cloud providers. All the services are offered within the framework through an Enterprise Service Bus (ESB). Three categories of provisioning
services are provided: *Cloud Resource Services, Packet-based Services*, and *Container Cluster Services*. The latter include functionalities to deploy applications on cluster platforms.

In order to integrate such 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 BPMN 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 Provisioning Tasks.

**Fig. 2.** Overview of the provisioning scenario, showing the different layers of the framework [13].

In this work we focus on container-based applications that 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. The orchestration process for Deployment Units will be discussed later in this paper.

### 6.2 YAML Parsing

In our framework, the first step towards the deployment orchestration is the YAML processing. The Parser is the software component in charge of processing the TOSCA Template of an application and outputting the necessary information for the deployment of the application.

The Parser is able to deconstruct the complexity of the application scenario by analysing the nodes and the relationships present, acquiring the relevant configuration settings for the deployment, and compressing them into a format that can be processed by the framework.

In order to analyse a Deployment Unit, a bottom-up approach has been adopted. Starting from a Container Runtime, the Parser identifies the Deployment Unit and recursively finds 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.

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.

6.3 BPMN Plans

The BPMN plans in our platform rework the strategy adopted in [12]. In Fig. 3, 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. We expanded the BPMN Plans for our purpose, modelling a workflow path for deployment unit nodes.

![Fig. 3. Instantiate node overall workflow [13].](image)

In Fig. 4, the detailed workflow for a deployment unit node is depicted. 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. If a fault occurs, it is immediately caught and the entire 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 invokes 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.

### 6.4 Service Connectors

Service Connectors are software modules that include the logic to provision a specific resource or service, interacting with the external providers.

*Cloud Service Connectors* implement interactions with cloud providers for the allocation of computational, networking and storage resources. *Container Cluster Connectors* concern the deployment of containerised units on different container
cluster platforms. *Packet-based Connectors* implement interactions with all service providers that provide packet-based applications.

Cloud Service Connectors and Container Cluster Connectors contribute to the cause of container orchestration in our framework. The first category contains the services related to the different cloud providers. 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 *Instantiate Cluster* connector interface provides an endpoint to deploy different kinds of container clusters on the cloud. All concrete Connectors to real cloud services (AWS, OpenStack, Azure, etc.) should implement the Instantiate Cluster interface.

The second category 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 that realise the deployment of the scenario. The *Instantiate DU* connector interface contains methods to interpret, deploy and configure a Deployment Unit on specific container-management platforms. All concrete Connectors to container cluster services should implement the Instantiate DU interface. These connectors perform a translation from the parsed topology to the specific format of the container cluster platform and communicate with the cluster to operate the deployment.

### 6.5 Prototype Implementation

In this subsection, we discuss the implementation of the framework.

The Dashboard is a web application, which has been implemented using the Vue.js framework\(^3\), for the front-end user interface, and the PHP-based Laravel framework\(^4\), for the back-end. The users’ data, such as personal information, custom settings and deployments, are stored in an SQL database. The visualisation of an application’s deployment graph as well as the tool to sketch a TOSCA template have been developed using the Cytoscape.js library\(^5\), which is a JavaScript tool for networks’ visualisation and analysis.

The Parser software component is widely based on the OpenStack parser\(^6\) for TOSCA Simple Profile, 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 format that allows to integrate it into the BPMN plans.

---

\(^3\) [https://vuejs.org/](https://vuejs.org/).

\(^4\) [https://laravel.com/](https://laravel.com/).

\(^5\) [https://js.cytoscape.org/](https://js.cytoscape.org/).

\(^6\) [https://wiki.openstack.org/wiki/TOSCA-Parser](https://wiki.openstack.org/wiki/TOSCA-Parser).
We used Flowable\textsuperscript{7}, which is a Java based open-source business process engine, for the implementation of the BPMN workflow processing. Flowable includes several different components, such as Flowable Modeller, a module which allows to design new BPMN plans, and Flowable API, which represents a REST endpoint to the Flowable BPMN Engine. In our work, we used the Modeller to sketch the BPMN plans and the API module to integrate the Flowable Engine in our framework.

Finally, we implemented four Service Connectors: two OpenStack connectors, for the creation of either a Kubernetes cluster or a Docker Swarm cluster, one Kubernetes connector and one Docker Swarm connector, for the deployment of the DUs. The implementation has been done complementing Eclipse Vert.x\textsuperscript{8}, a Java toolkit for event-driven applications, with the OpenStack4J\textsuperscript{9} library, for the creation of the clusters, and with the official Kubernetes Java client\textsuperscript{10} and the Spotify Docker client\textsuperscript{11}, for the deployment of the DUs.

Overall, the implementation of TORCH is strongly loosely-coupled. The communication between the different components of the framework mainly builds upon web-based approaches, such as the adoption of REST APIs or CRUD services. The heterogeneity of the languages and the libraries used, along with the possibility to utilise different clients for the Service Connectors, provides a tangible proof of the extensibility of the framework.

7 Experiments

In this section, we assess the performance of our framework within two use cases: the first one is a simple Wordpress (WP) scenario, while the second one is a load test scenario, containing 20 Deployment Units (DU) in total, which represent ten instances of the original WP scenario. Our testbed consists of a machine running the full framework, along with the monitoring tools to collect some metrics about the system performance.

7.1 Use Case

We corroborate the working behaviour of our software with a test on a simple real-world scenario. This is a containerised version of a WP application including two DUs, MySQL and Wordpress, which are both stacked with a Volume and a Container. The scenario is depicted in Fig. 5, by using the TOSCA Simple Profile standard notation.

In the WP scenario, container images are Docker images that need to be pulled from the Docker Hub repository, as specified in the template. The TOSCA Artifact image fills the implementation parameter for the Create step in the

\textsuperscript{7} https://www.flowable.org/.
\textsuperscript{8} https://vertx.io/.
\textsuperscript{9} http://www.openstack4j.com/.
\textsuperscript{10} https://github.com/kubernetes-client/java.
\textsuperscript{11} https://github.com/spotify/docker-client.
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 container. For being correctly parsed, the environment variables have the same names that are indicated in the Docker Hub 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 1.5, the TOSCA Simple Profile description of the types and the templates used for the MySQL deployment unit is exemplary provided. The description is drafted according to the principles defined in Sect. 5.

```yaml
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 }

deployment:
  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:
```

Fig. 5. The Wordpres scenario, described using TOSCA notation [13].
7.2 Performance Evaluation

We used the Prometheus\textsuperscript{12} toolkit together with the Node Exporter\textsuperscript{13} metrics collector to monitor the CPU, the memory and the network usage of the system. The machine used for the tests is equipped with an Intel(R) Core(TM) i7–4770 processor, 16 GB RAM, a 1 TB hard-drive and a 128 GB SSD, and it runs the Ubuntu 16.04 × 86–64 Linux distribution.

Table 2. Steady-state resource statistics about the testing system.

|                | CPU avg | MEM avg | NET IN avg | NET OUT avg |
|----------------|---------|---------|------------|-------------|
| w/o Framework  | 0.39%   | 1.53 GiB| 32.9 kbps | 0 kbps      |
| with Framework | 0.94%   | 4.89 GiB| 33.4 kbps | 0 kbps      |

Table 2 displays the average values for the metrics when the machine is found in a steady state with and without the framework running. The metrics have been collected on a 5-min time frame and averaged over time, to prevent occasional system processes from interfering with our analysis. The table shows that the framework has a major impact on the RAM of the system, with around 3.45 GiB employed, a small impact on the CPU, with an increase of about 0.55% usage, and almost no impact on the network traffic. This is due to the absence of incoming network requests when the machine is in a steady state.

Overall, our tests utilise the framework in an end-to-end fashion. The testbed machine, hosting the whole framework, is accessed to upload the TOSCA scenario and provide any deployment properties, by using the Dashboard. Then, at the provisioning stage, the framework communicates with an OpenStack cluster to accomplish the deployment of the scenario.

\textsuperscript{12}https://prometheus.io/.

\textsuperscript{13}https://github.com/prometheus/node_exporter.
The OpenStack cluster is deployed on local machines and it consists of two computers: a Controller node and a Compute node. The Controller is equipped with an Intel(R) Core(TM) i7-4770S, 8 GB RAM and a 1 TB hard drive, and it also runs the Heat and Magnum services. The Compute is equipped with an Intel(R) Core(TM) i5-4460, 8 GB RAM, a 128 GB SSD and a 1 TB hard drive. Both nodes run the Ubuntu Server 18.04 × 86–64 Linux distribution.

**Table 3.** Deployment average times - WP single instance.

|                     | Google Kubernetes | Docker Swarm   |
|---------------------|-------------------|----------------|
| **Create cluster** | 5min 41s ± 19s    | 4min 4s ± 17s  |
| **Create du**      | 2min 59s ± 16s    | 1min 39s ± 13s |
| **Total**          | 8min 40s ± 20s    | 5min 43s ± 16s |

**Table 4.** Deployment average times - WP multiple instances.

|                     | Google Kubernetes | Docker Swarm   |
|---------------------|-------------------|----------------|
| **Create cluster** | 5min 54s ± 18s    | 4min 7s ± 19s  |
| **Create du**      | 4min 58s ± 25s    | 2min 34s ± 11s |
| **Total**          | 10min 52s ± 40s   | 6min 41s ± 2s  |

Both the simple WP scenario and the load test scenario have been correctly deployed on Kubernetes and Swarm. The deployment times are shown in Table 3 and Table 4. The tables display the average times ± the standard deviation of ten deployment trials. Data is provided about the total amount of time needed to complete the deployment, but we also distinguish the necessary time to create the container cluster from the time employed to deploy all the DUs.

In general terms, the deployments are faster on the Swarm platform. This applies to both the cluster and the DU deployments, and it is likely due to the major resource requirements to run the Google’s container cluster platform. As expected, the deployment times are longer for the load test scenario.

We detail the testbed machine resource usage in Fig. 6 and Fig. 7. The plots show the CPU, memory and network traffic states over time, for Kubernetes and Swarm deployment trials. We highlight the presence of three operational stages in the plots: the setup, the cluster creation and the DU creation. Overall, the performance of the testbed is very similar for the two container cluster platforms tested, across all the phases. However, we observe a difference in the CPU usage between the single WP and the load test scenarios, with the latter having a higher CPU usage.
The setup stage consists of the pre-deployment operations that are required to start the orchestration process, such as the communication of the application description to the orchestrator and the check of provisioning services that are adequate for the deployment. As for the plots, this stage implies an increase in the CPU usage, which reaches peaks of 6–7% usage for Kubernetes and 2–3% for Swarm. We also observe the largest peak in the network transmitted traffic, with the network usage being higher for the load test scenario. This is because the application specification file has a bigger size and there are more BPMN workflows to transfer, due to the higher number of DUs.

The cluster creation phase entails a steady usage of the CPU, with a value that is higher than the steady-state CPU usage but lower than in the other two phases. There are no heavy network transfers, but frequent spikes are present in the network plots because of the requests sent by the orchestrator monitoring the resources’ provisioning status and by the Dashboard monitoring the deployment state of the DUs.

Finally, in the DU creation stage, it occurs another CPU increase. This is mainly due to the parallelism capabilities of the framework, which is able to deploy multiple DUs at the same time, exploiting the resource availability of
the testbed machine. The network traffic is mainly busy with the transmission of monitoring information, even though we can notice a small increase in the received traffic, likely due to the numerous parallel incoming connections, communicating information about the DU deployment status.

During the DU creation stage, we observe the most significant difference between the single WP scenario and the load test scenario deployments: for the first, the CPU usage has its highest peaks at 4%, while for the second the highest peaks reach 7% of usage.

The results of our tests show that the framework is computationally inexpensive. In our evaluation, the CPU usage rarely goes beyond the 5% threshold and we often find the value to be lower than 3%. From a memory perspective, the framework requires a few GiB of RAM for being setup but, then, at the provisioning stage, the framework is parsimonious in the memory management and does not require substantial additional resources. Moreover, we noticed a regular network usage during all the provisioning phases.

Finally, we could not find tremendous differences between the deployment of a simple two-DU WP scenario and a twenty-DU load test scenario. This implies that the framework might have good scalability capabilities, although
tests with multiple clusters should be performed to corroborate this hypothesis. Unfortunately, we were not able to deploy multiple clusters because of hardware constraints, but extensive scalability tests will be part of future work.

8 Conclusion

The ever-growing interest in cloud computing on the part of both academia and industry has promoted a business scenario where several cloud providers offer similar services and contend for an extremely lucrative market. One of the keys to competitive cloud providers is the automated provisioning of complex cloud applications. As a result, a number of orchestration tools have appeared in order to simplify the entire life-cycle management of cloud applications. Additionally, container technologies have come into the limelight in recent years, since organisations package applications in containers and orchestrate multiple containers across multiple cloud providers.

In this paper, we introduced a TOSCA-enabled framework for the deployment and orchestration of cloud applications across multiple providers, with special focus on containerised applications operating on top of multiple cluster platforms. A prototype implementation of the framework was developed and a few experiments were carried out on a small-scale test-bed. Preliminary tests corroborated the viability of the approach and showed promising results with regard to scalability. In the future, more experiments will be performed on top of different container-based cluster platforms to assess the maturity of the system on a large scale.

References

1. Apache Software Foundation: ARIA TOSCA. https://ariatosca.incubator.apache.org/. Accessed 10 July 2020
2. Apache Software Foundation: Mesos. http://mesos.apache.org/. Accessed 10 July 2020
3. Apache Software Foundation: Mesos Containerizer. http://mesos.apache.org/documentation/latest/mesos-containerizer/. Accessed 10 July 2020
4. Bellendorf, J., Mann, Z.Á.: Cloud topology and orchestration using TOSCA: a systematic literature review. In: Kritikos, K., Plebani, P., de Paoli, F. (eds.) ESOCC 2018. LNCS, vol. 11116, pp. 207–215. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99819-0_16
5. Binz, T., et al.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 692–695. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_62
6. Bogo, M., Soldani, J., Neri, D., Brogi, A.: Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes. Softw. Pract. Experience 50(9), 1793–1821 (2020). https://doi.org/10.1002/spe.2848
7. Brogi, A., et al.: SeaClouds: seamless adaptive multi-cloud management of service-based applications. In: 17th Conferencia Iberoamericana en Software Engineering (CIbSE 2014), pp. 95–108 (2014)
8. Brogi, A., et al.: Adaptive management of applications across multiple clouds: the SeaClouds Approach. CLEI Electron. J. 18, 2–2 (2015). https://doi.org/10.19153/cleiej.18.1.1
9. Brogi, A., Rinaldi, L., Soldani, J.: TosKer: a synergy between TOSCA and Docker for orchestrating multicomponent applications. Softw. Pract. Experience 48(11), 2061–2079 (2018). https://doi.org/10.1002/spe.2625
10. Buyya, R., et al.: A manifesto for future generation cloud computing: research directions for the next decade. ACM Comput. Surv. 51(5), 105:1–105:38 (2018). https://doi.org/10.1145/3241737
11. Caballer, M., Zala, S., García, Á.L., Moltó, G., Fernández, P.O., Velten, M.: Orchestrating complex application architectures in heterogeneous clouds. J. Grid Comput. 16(1), 3–18 (2018). https://doi.org/10.1007/s10723-017-9418-y
12. Calcaterra, D., Cartelli, V., Di Modica, G., Tomarchio, O.: 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), pp. 533–540. Funchal-Madeira (Portugal), March 2018. https://doi.org/10.5220/0006775605330540
13. Calcaterra, D., Di Modica, G., Mazzaglia, P., Tomarchio, O.: Enabling container cluster interoperability using a TOSCA orchestration framework. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020), pp. 127–137, May 2020. https://doi.org/10.5220/0009410701270137
14. Cloud Native Computing Foundation: containerd. https://containerd.io/. Accessed 10 July 2020
15. Cloud Native Computing Foundation: CRI-O. https://cri-o.io/. Accessed 10 July 2020
16. Docker Inc.: Docker. https://www.docker.com/. Accessed 10 July 2020
17. Docker Inc.: Docker Swarm. https://docs.docker.com/engine/swarm/. Accessed 10 July 2020
18. FastConnect: Alien4Cloud. https://alien4cloud.github.io/. Accessed 10 July 2020
19. GigaSpaces Technologies: Cloudify. https://cloudify.co/. Accessed 10 July 2020
20. Google: Google Kubernetes. https://kubernetes.io/. Accessed 10 July 2020
21. Kehrer, S., Blochinger, W.: TOSCA-based container orchestration on Mesos. Comput. Sci. Res. Dev. 33(3), 305–316 (2018). https://doi.org/10.1007/s00450-017-0385-0
22. Kiss, T., et al.: MiCADO—microservice-based cloud application-level dynamic orchestrator. Future Gener. Comput. Syst. 94, 937–946 (2019). https://doi.org/10.1016/j.future.2017.09.050
23. Mesosphere Inc.: Marathon. https://mesosphere.github.io/marathon/. Accessed 10 July 2020
24. OASIS: Web Services Business Process Execution Language Version 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed 10 July 2020
25. OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0, November 2013. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html. Accessed 23 Dec 2020
26. OASIS: TOSCA Simple Profile in YAML Version 1.2 (2019). https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html. Accessed 23 Dec 2019
27. OMG: Business Process Model and Notation (BPMN 2.0), January 2011. http://www.omg.org/spec/BPMN/2.0/. Accessed 10 July 2020
28. OpenStack: Build the future of open infrastructure. https://www.openstack.org/. Accessed 10 July 2020
29. OpenStack: Heat. https://wiki.openstack.org/wiki/Heat. Accessed 10 July 2020
30. OpenStack: Magnum. https://wiki.openstack.org/wiki/Magnum. Accessed 10 July 2020
31. Petcu, D., Vasilakos, A.: Portability in clouds: approaches and research opportunities. Scalable Comput. Pract. Experience 15(3), 251–270 (2014). https://doi.org/10.12694/scpe.v15i3.1019
32. Salomoni, D., Campos, I., Gaido, L., et al.: INDIGO-DataCloud: a platform to facilitate seamless access to E-Infrastructures. J. Grid Comput. 16(3), 381–408 (2018). https://doi.org/10.1007/s10723-018-9453-3
33. Singh, S., Singh, N.: Containers & Docker: emerging roles & future of Cloud technology. In: 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), pp. 804–807, July 2016. https://doi.org/10.1109/ICATCCT.2016.7912109
34. Sysdig: Sysdig 2019 container usage report (2019). https://sysdig.com/blog/sysdig-2019-container-usage-report/. Accessed 23 Dec 2019
35. Tosatto, A., Ruiu, P., Attanasio, A.: Container-based orchestration in cloud: state of the art and challenges. In: 9th International Conference on Complex, Intelligent, and Software Intensive Systems, (CISIS 2015), pp. 70-75 (2015)
36. Weerasiri, D., Barukh, M.C., Benatallah, B., Sheng, Q.Z., Ranjan, R.: A taxonomy and survey of cloud resource orchestration techniques. ACM Comput. Surv. 50(2), 26:1–26:41 (2017). https://doi.org/10.1145/3054177