Journal of Grid Computing (2021) 19: 5
https://doi.org/10.1007/s10723-021-09549-z

TORCH: a TOSCA-Based Orchestrator of Multi-Cloud Containerised Applications

Orazio Tomarchio · Domenico Calcaterra · Giuseppe Di Modica · Pietro Mazzaglia

Received: 10 July 2020 / Accepted: 18 January 2021 / Published online: 18 February 2021

Abstract The growth in the number and types of cloud-based services offered to IT customers is supported by the constant entry of new actors in the market and the consolidation of disruptive technologies such as AI, Big Data and Micro-services. From the customer's perspective, in a market landscape where the cloud offer is highly diversified due to the presence of multiple competing service providers, picking the service that best accommodate their specific needs is a critical challenge. Once the choice is made, so called "cloud orchestration tools" (orchestrators) are required to take care of the customer application's life-cycle. While big players offer their customers proprietary orchestrators, in the literature quite a number of open-source initiatives have launched multi-cloud orchestrators capable of transparently managing applications on top of the most representative cloud platforms. In this paper, we propose TORCH, a TOSCA-based framework for the deployment and orchestration of cloud applications, both classical and containerised, on multiple cloud providers. The framework assists the cloud customer in defining application requirements by using standard specification models. Unlike other multi-cloud orchestrators, adopts a strategy that separates the provisioning workflow from the actual invocation of proprietary cloud services API. The main benefit is the possibility to add support to any cloud platforms at a very low implementation cost. In the paper, we present a prototypal implementation of TORCH and showcase its interaction with two different container-based cluster platforms. Preliminary performance tests conducted on a small-scale test-bed confirm the potential of TORCH.

Keywords Cloud orchestration · Automated deployment and provisioning · Containerised applications · TOSCA · BPMN

O. Tomarchio (✉) · D. Calcaterra · G. Di Modica · P. Mazzaglia
Department of Electrical, Electronic and Computer Engineering, University of Catania, Catania, Italy
e-mail: orazio.tomarchio@unict.it

D. Calcaterra
e-mail: calcaterra.domenico@gmail.com

G. Di Modica
e-mail: dimodica@unict.it

P. Mazzaglia
e-mail: pietromazzaglia@gmail.com

1 Introduction

In the last decade, cloud computing has established itself as a new paradigm of distributed computing that allows the sharing of resource pools on an on-demand basis model. For IT industry, this leads to several benefits in terms of availability, scalability and costs, lowering the barriers to innovation [1]. Moreover, cloud technologies encourage a larger distribution of services across the internet [2]. As reported in the Flexera 2020 State of the Cloud Report [3], many companies and organisations have successfully adopted the cloud computing paradigm worldwide, while more

and more are approaching it, as they see a real opportunity to grow their business. Since cloud has emerged as a dominating paradigm for application distribution, providers keep on implementing new features to offer services which are not limited to just infrastructure provisioning. This trend is depicted as "Everything as a Service", namely *XaaS* [4].

The increasing complexity of the cloud environments, along with the challenges posed by building and managing scalable applications, has drawn out the necessity of software that would simplify the deployment and monitoring processes for cloud applications. In this regard, *cloud orchestration* tools have increased their popularity in recent years, becoming a main topic for cloud research [5, 6]. Today, most of the commercial cloud providers offer an orchestration platform to end-users [7]: however, these products are proprietary and not portable. Besides, although modern configuration management solutions provide support for handling resource configuration over cloud services, all potential users are often required to understand various low-level cloud service APIs and procedural programming constructs in order to create and maintain complex resource configurations.

The advent of the multi-cloud computing and its wide adoption by enterprises [3] further exacerbates the orchestration issues. The *multi-cloud paradigm* is a recent technological trend within the cloud computing landscape, which gravitates towards the possibility to access services and resources provided by multiple providers [8]. With respect to other paradigms involving multiple cloud platforms, the multi-cloud one is characterised by a unified interface that allows to select which cloud to use for consuming services and resources, choosing among a collection of various platforms [9]. This brings in several advantages, such as reducing vendor lock-ins, enabling a fairer marketplace of cloud services, and introducing a smaller overhead compared to solutions that engage with multiple clouds at the same time.

In order to effectively enable the multi-cloud paradigm, it is essential to guarantee an easy *portability* of applications among cloud providers [10, 11]. This new requirement calls for more powerful cloud orchestration mechanisms capable of dealing with the heterogeneity of the underlying cloud resources and services. In the last few years, several efforts have been made to overcome these issues. Open Standards, such as OASIS CAMP [12] and OASIS TOSCA [13], have been proposed for modelling the application topology and the component life-cycles so as to facilitate the orchestration process and enhance portability across different providers. In particular, TOSCA stands out for the large number of literature works and tools that are based upon it [14].

Container-based applications have provided a solution to improve portability in the cloud deployment landscape [15]. 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, high reusability and near-native performances [16] 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. Container-centric orchestrators such as Docker Swarm [17], Kubernetes [18], and Apache Mesos [19] have appeared. They perform orchestration at container level by automating the provisioning and management of complex containerised deployments across multiple hosts and locations. 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 present the design and development of TORCH, a framework for the deployment and orchestration of classical and containerised cloud applications on top of different cloud providers. Our work leverages the TOSCA specification to build up a cloud service orchestrator capable of automating the execution of tasks and operations required for the provisioning of a multi-cloud application. The basic strategy adopted by TORCH is to transform a TOSCA application model into an equivalent BPMN workflow and dataflow model, which a BPMN engine leverages to enforce the operations specified in the model [20, 21]. The proposed approach clearly separates the orchestration of the provisioning tasks from the real provisioning services, easily allowing the deployment on different cloud providers. Based on our previous work [22], we provide integration with a variety of container-based technologies in order to close the loop. In this regard, the fault-aware orchestration of containerised applications is supported via new business process models and pluggable software connectors. We also enrich our web tool with features

for the modelling, deployment management and monitoring of both containerised and non-containerised applications.

In summary, TORCH main features include:

- description and modelling of the application topology using standard languages (namely, TOSCA);
- capability to deploy application components on different cloud providers, by means of pluggable "connectors";
- integration with different container-based cluster technologies;
- fault-aware orchestration based on a set of business process models;
- deployment management through a simple web tool.

In this work, after discussing the technical choices on which the TORCH design grounds, we provide implementation details of the TORCH prototype and show some performance results obtained from running the prototype on a small-scale test-bed. The remainder of the paper is organised as follows. In Section 2, a background of technologies exploited in this work is presented. In Section 3, we present the main design principles of TORCH, along with its overall architecture. The prototypal implementation is discussed in Section 4. In Section 5, the deployment and the performance results of a use case application on a small-scale test-bed are presented. Related work is discussed in Section 6. Finally, we conclude the work in Section 7.

2 Background

This work aims to provide synergy between cloud resource orchestration, portable topology specification and containerisation technologies. In this section, we provide a more in-depth background on these topics.

2.1 Cloud Orchestration

Cloud orchestration denotes various processes and services to select, describe, configure, deploy, monitor and control cloud services or resources across different cloud solutions in an automated way. The overall goal of orchestration is to guarantee successful hosting and seamless delivery of applications by meeting the Quality of Service (QoS) goals of both cloud application owners and cloud resource providers [23].

Many cloud industry players have developed cloud management platforms (CMP) to automate the provisioning of cloud services (e.g. Amazon CloudFormation [24], Flexera Cloud Management Platform [25], RedHat CloudForms [26], IBM Cloud Orchestrator [27]). The most advanced platforms also offer lifecycle management of cloud applications. These commercial products are neither open to the community nor portable across third-party providers.

As to open-source cloud orchestration frameworks, OpenStack provides a service to orchestrate composite cloud applications using a declarative template-based format, i.e. Heat Orchestration Template (HOT) [28], through both an OpenStack-native REST API and AWS CloudFormation-compatible API. Another notable example of orchestration platform in the open-source domain is Cloudify [29], which allows to model TOSCA-compliant applications and automate their lifecycle via a set of built-in workflows.

There are a number of neighbouring tool categories which share similarities with cloud orchestrators. Configuration management tools (Ansible [30], Chef [31], Puppet [32], Salt [33]) are mostly built to automate the development, delivery, testing and maintenance throughout the software life-cycle. These tools have recently moved towards orchestration functionalities, including virtual machine and infrastructure creation, which leverage Infrastructure as Code (IaC) [34] to change, configure, and automate infrastructure. Terraform [35] is one of the most notable IaC open-source solutions. Nevertheless, it includes no life-cycle management, while scaling and error-handling are only provided with external support.

Regarding standardising initiatives, BPEL [36] and BPMN [37] are the most applied standards for service composition and fault-aware orchestration [38]. Three basic fault handling concepts are supported by BPEL: *compensation handlers*, *fault handlers*, and *event handlers*. Nevertheless, BPEL only manages predefined faults specified by application designers. Similarly, BPMN supports *error events*, *cancel events* and *compensation events*.

2.2 Cloud Portability

Portability has been defined as the capability of a program to be executed on various types of data processing systems without converting the program to a different language and with little or no modification [39]. In respect to cloud computing, portability comprises three categories: data portability, application portability and platform portability [40]. In particular, *application portability* refers to the ability to define application features in a vendor-agnostic way.

Supporting *open standards* such as CAMP [12] and TOSCA [13] for modelling the application's topology and components facilitates the usage of cloud orchestrators and further increases the reusability of the topology definition, as it restricts the vendor lock-in issue to cloud provider level.

CAMP is a standard developed by OASIS which focuses on providing a management API for cloud platforms. It defines interfaces and artifacts for self-service provisioning, monitoring and control. A resource model for representing applications and their components is defined. CAMP was designed to be language, framework, and platform-neutral with the goal of covering a variety of cloud platforms, which is why it uses a REST-like protocol for manipulating the specified resources. Apache Brooklyn [41] is a notable open-source cloud orchestration framework implementing CAMP.

TOSCA is a standard designed by OASIS to enable the portability of cloud applications and the related IT services. The structure of a cloud application is described 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 topology node can also contain information such as the corresponding *requirements*, the operations to manage it (*interfaces*), the *attributes* and the *properties* it features, the *capabilities* it provides, and the software *artifacts* it uses. Cloud applications are typically packaged in TOSCA using *Cloud Service Archive (CSAR)* files.

TOSCA Simple Profile is an isomorphic rendering of a subset of the TOSCA specification in the YAML language [42]. It defines a few normative workflows 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.

2.3 Containerisation

Container-based virtualisation [43] is a key approach for sharing the host operating system kernel across multiple guest instances, while keeping them isolated. Environment-level containers provide resource isolation with little overhead compared to OS-level hypervisors [44]. Docker [45] represents the leading Linux-based technology for container runtimes [46]. Among competitors, containerd [47], CRI-O [48] and Containerizer [49] are worth mentioning.

In recent times, container-centric cluster solutions have grown in popularity regarding container deployment and management. Most of them perform orchestration at container level by automating the provisioning of complex containerised deployments across multiple hosts and locations, leading to greater scalability, improved reliability and sophisticated management.

Kubernetes [18] currently constitutes the most widespread ecosystem for the deployment, scaling and management of containerised workloads. Docker Swarm [17] offers a native solution to integrate cluster management into Docker. Apache Mesos [19] is an open-source project to manage computer clusters, which natively supports Docker containers and can be used in conjunction with Marathon [50], a container orchestration platform.

Most cloud providers, such as Amazon AWS, Microsoft Azure and Google Cloud have built-in services to operate containers and clusters. OpenStack, serving as an open-source alternative to control large pools of resources, supports container orchestration by means of *Magnum* [51] and *Heat* [28] services. The former allows clustered container platforms (Kubernetes, Mesos, Swarm) to interoperate with other OpenStack components. The latter is a service to orchestrate composite cloud applications.

The great availability 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 one requires several manual operations to be performed, and the learning curve might not be entirely negligible. These shortcomings have led to systems to automate deployment and management operations while interfacing with multiple container technologies, clusters and cloud providers.

3 TORCH Design

The objective of this work is to propose an open-source platform that offers user-friendly and reliable services to deploy user applications on top of any cloud provider infrastructure. In this section, we discuss inspiring design principles and delve into technical details of the framework.

3.1 Design Principles

Two simple principles guided the design of the proposed system. The first principle concerns cloud users and their need to easily specify the requirements of the application to be deployed on the Cloud. We make the basic assumption that application owners are completely agnostic as to how their applications are handled on the Cloud. Nevertheless, they know very well their application structure, i.e., the number and type of software modules that the application is composed of, modules inter-dependencies, functional and non-functional requirements, etc. Application owners should be supported to supply a representation of such application requirements in a way that is as simple as unambiguous. In the remainder of the paper we will interchangeably use the terms *Cloud user* and *Customers* to refer to the application owner that buys cloud resources (computing capacity, storage, etc.) to deploy their application on top of them.

The second principle regards the application of a provisioning strategy that clearly separates the provisioning workflow from the actual invocation of cloud services that enforce the provisioning. We argue that, despite cloud providers expose proprietary APIs, the activities underlying any application provisioning process follow a common and API-independent pattern, that basically consists in instantiating the required resource(s) on the cloud provider infrastructure, deploying the application components on the instantiated resources, making the necessary software configuration, and finally running the application. By isolating the provisioning workflow from the actual cloud service invocation, several benefits derive in terms of:

- *Coding effort*. Once designed, the provisioning workflow can be fed to any off-the-shelf workflow engine that will transparently take care of pipelining the activities. Implementation effort will mainly focus on coding the plugins ("Connectors", from now on) responsible for connecting the workflow tasks to platform-specific APIs.
- *Maintainability*. As long as system responsibilities are distributed among multiple components (namely, the workflow engine and the connectors), system maintenance keeps easier and less time-consuming.
- *Expansibility*. Adding support for a new cloud provider is as easy as creating a new Connector.
- *Scalability*. No particular effort is requested to enforce the system scalability, since the responsibility of scaling up to provisioning requests is upon the workflow engine tool.

As far as modelling application deployment requirements is concerned, we choose to comply with the *OASIS TOSCA* standard, and more specifically, the *TOSCA Simple Profile* rendering. The latter offers a human-readable language to describe both the application topology and the artifacts needed by the application itself. We make use of BPMN [37] to model provisioning activities. BPMN is a widespread process modelling language which exhibits a rich expressiveness and is supported by many reliable workflow engines. The connection between the workflow engine activities and the proprietary cloud provision API offered by the multitude of cloud providers is realised by means of ad-hoc plugins, whose implementation follows a specified development pattern as better detailed in Section 4. The proposed framework will enable a scenario of an open cloud service market, where cloud providers can participate and customers are presented with a variety of services to choose from.

3.2 Framework Architecture

In this section, the high-level architecture of the TORCH orchestration framework is presented.

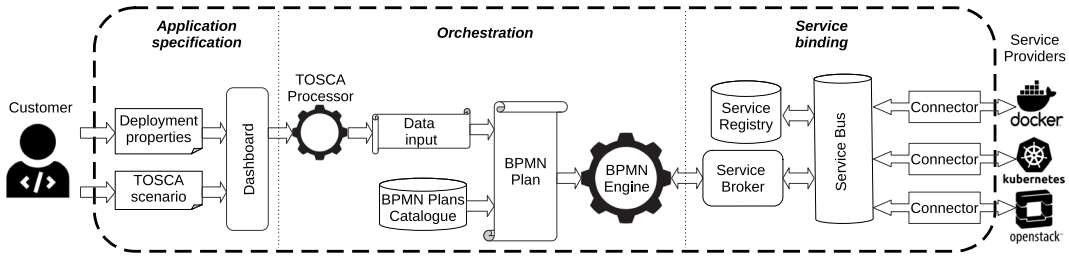


Fig. 1 TORCH provisioning scenario

The framework offers tools and services that facilitate the scenario depicted in Fig. 1. On the one hand, the framework offers Customers user-friendly tools to specify application provisioning requirements; on the other hand, it allows Cloud Providers to plug their deployment services by means of some service binding tools. In between, the framework automatically pipelines and executes the activities leading to the provisioning of the Customer's application on the desired cloud infrastructure.

The overall provisioning process is composed of three distinct sequential phases. The process begins with the *Application specification* phase, during which the Customer is in charge of modelling and submitting the application requirements. The submitted input triggers the *Orchestration* phase, that in turn consists of the transformation of the requirements into a workflow plan, composed of a combination of pre-modelled workflow models that get customised based on input data, and its subsequent enactment performed by a workflow engine. Finally, in the *Service binding* phase, ad-hoc service Connectors transform generic provisioning actions into concrete service deployment invocations.

Figure 2 shows the multi-layered framework architecture enabling the provisioning scenario shown in Fig. 1. The *Application Specification Layer* consists of two components: the Dashboard and the TOSCA Modeller. The Dashboard is the front-end component implementing the interaction with the Customer and the displaying views on the provisioning process status. The TOSCA Modeller guides the Customer to sketch application requirements. The *Orchestration Layer* is populated by the TOSCA Processor component, which is in charge of validating, parsing and converting TOSCA application scenarios into BPMN Data Inputs, and the BPMN Engine component, which is responsible for instantiating and orchestrating BPMN Plans. The *Service Binding Layer* comprises all the components necessary to manage the interaction between provisioning tasks and provisioning services. Invocation of cloud provisioning services is implemented in a SOA fashion (hence the presence of Service Broker, Service Bus and Service Registry components) and is mediated by Connector components.

3.3 Application Specification Layer

One of the main strengths of TORCH is the possibility to specify cloud and platform-agnostic applications using a fully-compliant TOSCA YAML description.

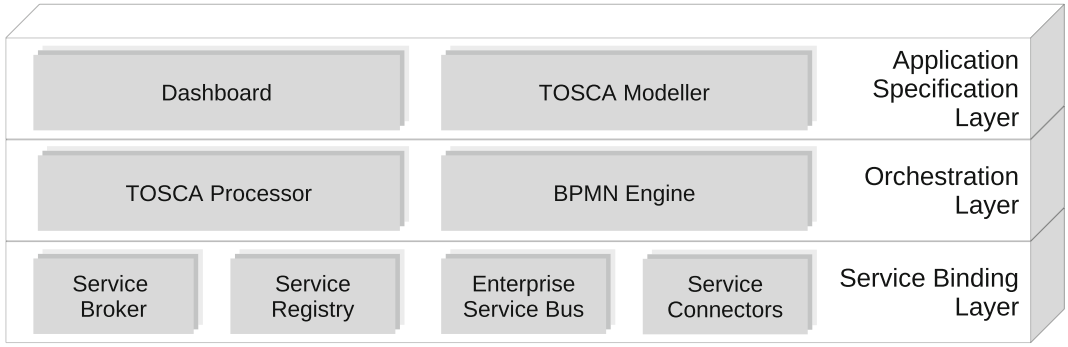


Fig. 2 TORCH framework architecture

```
tosca.nodes.Container.Runtime:
  derived_from: >
    tosca.nodes.SoftwareComponent
  capabilities:
    host:
      type: >
        tosca.capabilities.Container
    scalable:
      type: >
        tosca.capabilities.Scalable
```

(a)

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

(b)

Fig. 3 *Container.Runtime* (**a**) and *Container.Application* (**b**) node types

TOSCA Simple Profile provides a variety of types to describe VM-based and containerised applications in a generic and portable fashion. The application specification approach herein adopted is solely based on such normative types, avoiding any redefinition of standard entities or the employment of ad-hoc types as basic blocks of an application.

The standard presents two main node types to describe containerised applications: *Container.Runtime* and *Container.Application*, which are illustrated in Fig. 3. The former (Fig. 3a) constitutes the software component where a container application runs, while the latter (Fig. 3b) is used to describe a container entity. In TORCH, single or multiple Applications running on a Runtime represent a *Deployment Unit* (DU).

As discussed in [22], a DU establishes a common representation for deployable entities across different container cluster technologies, allowing to specify container's properties, volumes and connections through a generic approach. Later, during the DU instantiation stage, each of these features would be mapped to the appropriate resources, according to the user's specification. The possibility to map TORCH's DU to several existing tools enables container cluster interoperability within the framework.

While normative types provide sufficient capabilities to define containerised scenarios, we found that, in the current version, the TOSCA standard neglects the possibility to distinguish different kinds of containers in an application. Indeed, no specific entities, such as *Database* or *WebApplication*, are derived from the basic container node type *tosca.nodes.Container.Application*, as opposed to the node types referring to VM-based pieces of software. In order to fill this gap in the standard, we propose to define nodes that derive from the *Container.Application* node type and provide containerised versions of the normative role-specific types.

Figure 4 provides an example of the way this extension is operated. The main difference lies in the *derived_from* field. While the original *WebApplication* node (Fig. 4a) derives from the *Root* node, the *Container.WebApplication* node (Fig. 4b) derives from the *Container.Application* node. Then, properties and capabilities of the original type are copied into the new containerised version. There is no need to replicate the requirements since containers represent self-contained pieces of software that, in contrast with common application software, have no pre-installation requirements.

We make clear that the employment of the derived nodes does not affect the orchestration process at all. Derived nodes work exactly the same way as the standard *Container.Application* node, given the fact that they simply add properties and capabilities but no additional requirements. This is significant for portability, as any TOSCA-compatible runtime would be able to execute the derived nodes as standard container applications. However, we believe that the possibility

```
tosca.nodes.WebApplication:
  derived_from: tosca.nodes.Root
  properties:
    context_root:
      type: string
  capabilities:
    app_endpoint:
      type: >
        tosca.capabilities.Endpoint
  requirements: ...
```

(a)

```
tosca.nodes.Container.WebApplication:
  derived_from: >
    tosca.nodes.Container.Application
  properties:
    context_root:
      type: string
  capabilities:
    app_endpoint:
      type: >
        tosca.capabilities.Endpoint
```

(b)

Fig. 4 Standard TOSCA *WebApplication* node type (**a**) and our containerised extension, *Container.WebApplication* (**b**)

to use role-specific containers increases the expressiveness of an application description.

In order to establish connections between containers in our application description, two DU nodes are required: the "source node", which has a requirement specifying a tosca.relationships.ConnectsTo relation, and a "target node", with endpoint capabilities. A link is created when the source's requirement points to the target node. This link, which is first established in the application specification, would later be interpreted as a port to expose, for the target node, and as a container link, for the source node.

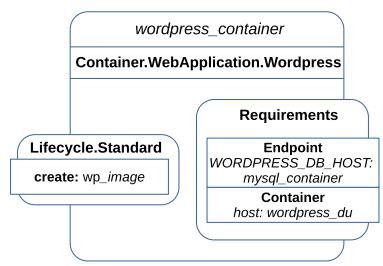
A DU can also be equipped with persistent storage capabilities through the support of a volume. In our application description, volumes work similarly to container's connections. The source node, which is the one being provided with the additional storage, specifies a tosca.capabilities.Storage requirement, which must be fulfilled by the target node. The resource instantiation for the storage and the volume bindings are automatically managed by TORCH at a later stage of the orchestration process.

An example of our approach is presented in Fig. 5. The *Wordpress* container node type (Fig. 5a) is defined using the extended *Container.WebApplication* type and can be employed to describe a Wordpress container in a scenario (Fig. 5b). The derivation from *Container.WebApplication* helps to specify the role that the container would play in the overall application. The example also presents a *WORDPRESS_DB_HOST* requirement that could be used to establish a connection to a database. Such requirement could be naturally satisfied either by a Container.Database node or by any Container.Application node possessing the tosca.capabilities.Endpoint.Database capability.

A TOSCA YAML application can be sketched using any text editor. To facilitate the process of creating a scenario and reduce the framework learning curve, TORCH provides a graphical tool to sketch YAML applications. The *TOSCA Modeller*, which is accessible from the Dashboard, allows to draw an application topology as a graph and, then, automatically convert it into a TOSCA Simple Profile template and download the resulting file(s).

```
tosca.nodes.Container.WebApplication.Wordpress:
  derived_from: >
    tosca.nodes.Container.WebApplication
  requirements:
    - WORDPRESS_DB_HOST:
        capability: >
          tosca.capabilities.Endpoint.Database
        relationship: >
          tosca.relationships.ConnectsTo
```

(a)



(b)

Fig. 5 *Wordpress* container node type definition (**a**) and instance example (**b**)

The Dashboard is the main endpoint for the cloud users to interact with the framework. After completing the registration process, every user gets access to a management panel which allows to create/upload new applications and monitor the deployment of the existing ones.

In conformance with the standard, uploading a new application expects a YAML or CSAR file which is straightaway validated by TORCH before being pushed to the user applications' database. Then, it is possible to deploy the application from a dedicated panel. At the moment of starting the deployment, the cloud user is required to provide a few deployment properties, such as cloud provider, container technology (where necessary), and retry settings.

The TOSCA YAML application specification, along with the deployment properties, are sent to the TOSCA Processor, which will process them and start the orchestration process. Finally, once the deployment process is launched, the provisioning status is monitorable from the application-dedicated panel.

3.4 Orchestration Layer

The TOSCA Processor (see Fig. 2) allows to convert TOSCA YAML templates into BPMN Data Inputs to configure pre-modelled BPMN models that the BPMN engine executes. These models define a workflow of fault-tolerant provisioning activities that can detect faults and, consequently, react in order to preserve the continuity of the provisioning process. All recoverable faults are autonomously managed by the process tasks, resulting in a minimised recourse to human intervention (referred to below as *escalation*).

In our previous work [20], we reviewed the potential faults which may occur at both provider side and client side. Depending on their nature, while some faults might be transient and hence recoverable by retrying the faulty operation, some others might be permanent thus requiring human intervention.

Based on the requirements for fault management services, we devised a set of BPMN provisioning models and extended them to support application deployments on container clusters. In Fig. 6, the overall service provision workflow is depicted. The input to the diagram is the set of all TOSCA *nodes* as produced by the TOSCA Processor. Originally, a TOSCA node was either a cloud resource or a software package. Afterwards, we expanded the BPMN plans for our purpose, modelling a workflow path for deployment unit nodes.

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. Depending on the TOSCA node type, a sub-process can proceed to a "create cloud resource", a "create deployment unit", or a "deploy package" sub-process. In these sub-processes, whenever an error is detected, an escalation is thrown by the relative "escalation end event" ("cloud resource error", "deployment unit error", or "package error") in the parent sub-process ("Instantiate Node"). The selective escalation was conceived to end only the faulty "Instantiate Node" sub-process and keep all

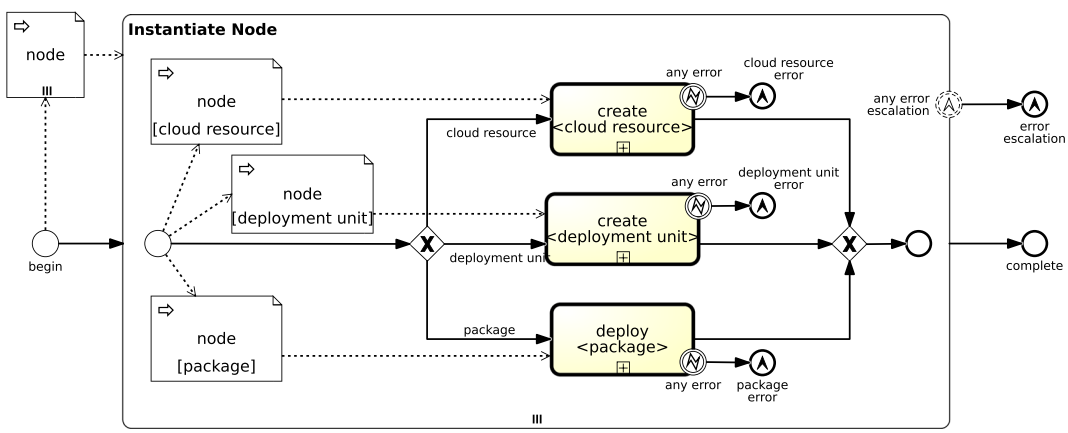


Fig. 6 Overall provision workflow

other sub-processes alive and running, while the faulty sub-process is being recovered.

For the sake of brevity, the workflows of the "create cloud resource" and "deploy package" sub-processes were omitted. We refer the reader to our previous work [20, 21] for further details. In addition to reporting the BPMN provisioning models for cloud resources and packet-based services, we presented the BPMN Choreography and Collaboration diagrams for a simple WordPress use case showing how individual components interact with each other.

In Fig. 7, 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 with the "create deployment unit" sub-process being analysed. The bottom pool called "Container Cluster Service Connectors" represents the pool of the software connectors deployed on the Service Bus. In the middle pool, the sequence of tasks carried out to create and instantiate a deployment unit are depicted.

The creation of a deployment unit starts with a task awaiting notifications 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 Service Bus. 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 Service Bus 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 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 request. 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 triggering an escalation event. Moreover, a compensation mechanism ("dispose deployment unit" task) allows to dispose of the deployment unit, whenever a fault has occurred.

Lastly, 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.

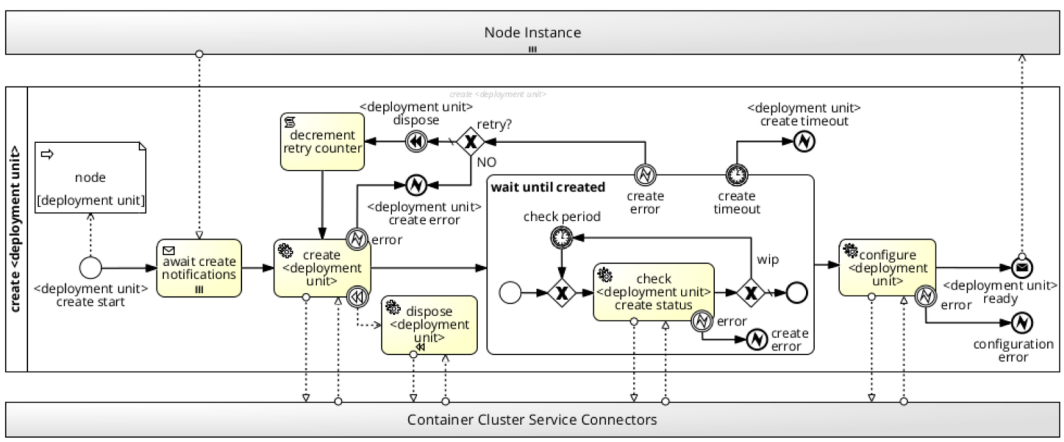


Fig. 7 Deployment Unit provision workflow

3.5 Service Binding Layer

The *Service binding layer* manages the provisioning of all the resources and services needed to deploy an application. It consists of four components: Service Bus, Service Registry, Service Broker, and Service Connectors.

The *Enterprise Service Bus* (ESB) is responsible for connecting the requests coming from the provisioning tasks with the provisioning services. The *Service Registry* is responsible for the registration and discovery of the service connectors. The *Service Broker* is in charge of taking care of the requests coming from the provisioning tasks.

Service Connectors (SC) are software modules that include the logic to provision a specific resource or service, interacting with the external providers. They provide unified interface models for the invocation of services, which allow to achieve *service location transparency* and *loose coupling* between provisioning BPMN plans and provisioning services. Each connector implements one of the three interfaces of SC that are presented in Fig. 8.

Cloud Resource Connectors (Fig. 8a) enable the provisioning of computational, networking and storage resources from cloud providers, such as AWS and OpenStack. For containerised applications, the *Instantiate Cluster* connector interface provides an endpoint to deploy different kinds of container clusters on the cloud. Other generic interfaces comprise the *AddStorage* and the *InstantiateVM* connectors. The first is a generic connector to cloud storage services, while the second is used for VM services.

Container Cluster Connectors (Fig. 8b) concern the deployment of containerised units on different container cluster platforms. The *Instantiate DU* interface contains methods to interpret, deploy and configure a DU on specific container-management platforms, such as Kubernetes or Docker Swarm. The process of interpreting, parsing and processing a DU is facilitated by an additional software component, the DU Translator, that is further described in Section 4.

Packet-based Connectors (Fig. 8c) implement interactions with all service providers that provide packet-based applications. These applications are software which should be executed on top of preconfigured runtime environments. Packets may also require installation, configuration and starting routines, which are reflected in the generic deployment life-cycle of a packet, which is organised into *create*, *configure* and *start* operations.

In general terms, all the connectors implementing a certain interface hold the same provisioning

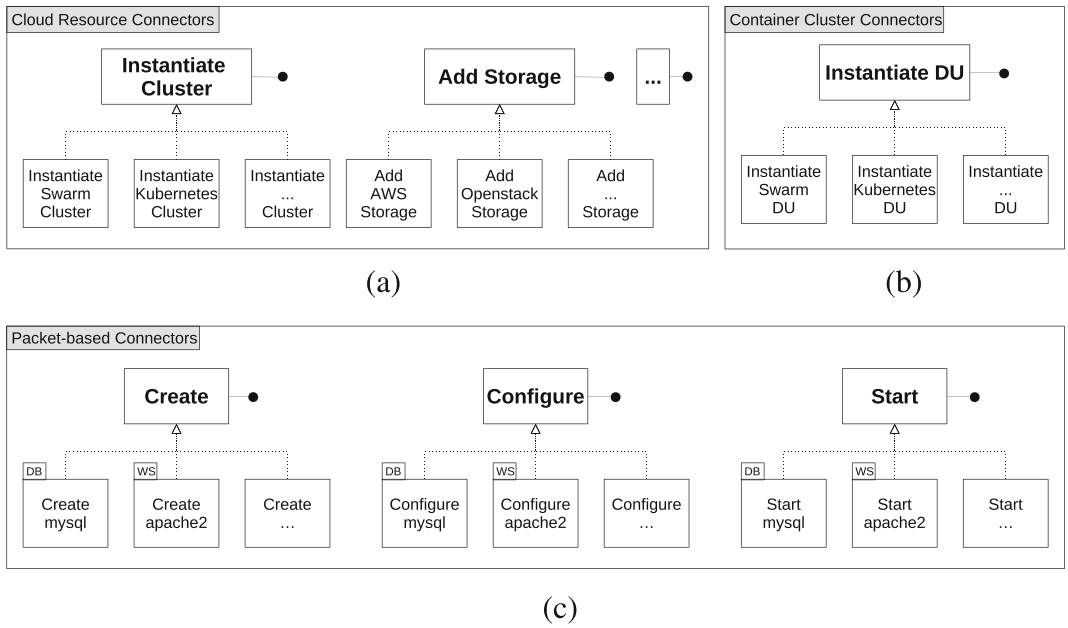


Fig. 8 Service connectors

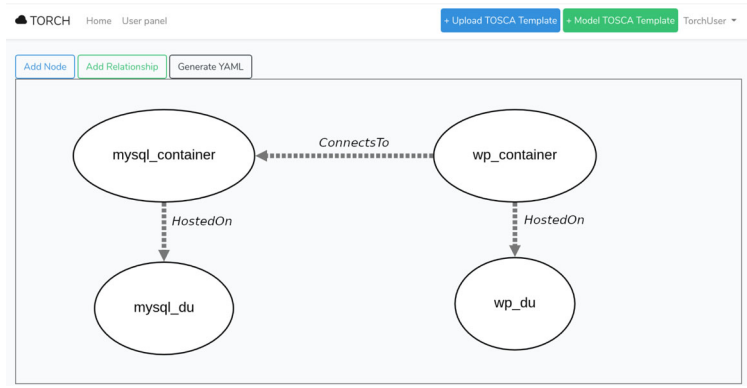


Fig. 9 Modeller sketching a simple Wordpress-MySQL containerised app

logic. The difference among them lies in the implementation of the interface methods that should match the correct cloud, container-cluster or packet-service API. For example, in order to instantiate a DU on Kubernetes a connector should implement the *create*, *check* and *configure DU* methods (see Section 4 for more details). These would include the code that communicates with the Kubernetes API to accomplish the deployment of the DU, according to the "create deployment unit" BPMN workflow.

At registration time, each Connector has to provide the Service Registry with all the information regarding the service being provided, such as a description, functional and non-functional properties of the service, and the URL. At the provisioning stage, the Service Broker attempts to meet the expectations of the requestors by querying the Service Registry and selecting the best-fitting Connectors. Finally, when a SC is invoked to provision a resource, the implemented methods from the connector interface are sequentially called, according to the BPMN workflow.

4 TORCH Prototype Implementation

The aim of this section is to describe the details of the implementation, providing a technical overview of the different components of TORCH. The code for the implementation is publicly available on GitHub.[1]

4.1 Application Specification Layer

The Dashboard is an interactive tool that simplifies the management of the deployments. It comprises two main components: a VueJS-based front-end GUI, which provides graphical tools to operate with the framework, and an SQL database, where it stores users and their respective templates and deployed applications. After registering into the system, a user gains access to several services, which include the possibility to model a TOSCA template, deploy a new application or monitor a former deployment.

As for the template modelling, it is possible to develop a new TOSCA template using the Modeller, which is a web-based tool that allows to graphically create, manipulate and, finally, generate a TOSCA Simple Profile customised template. As depicted in Fig. 9, the Modeller includes a graph visualiser to display the topology of the sketched application in real-time. Simple buttons are used to invoke forms to add either a node or a relationship between existing nodes to the topology. Finally, the *Generate YAML* button invokes the processing of the graph and the sketched template YAML is outputted.

After a YAML application is uploaded into the system, it is possible to visualise its topology, in the format it would be deployed, and provide some deployment properties that characterise the provisioning process. These features are shown in Fig. 10, where the deployment panel for a Wordpress-MySQL scenario is shown. On the right, some deployment properties are visible: the parameters 'Retry counter', 'Check period' and 'Create timeout' represent the retry settings to control the BPMN Engine behaviour (see Section 3.4), while the 'Cluster Platform' parameter allows to choose the desired container cluster technology among the available ones. This setting is applied to all the Deployment Units in the application scenario.

[1] https://github.com/unict-cclab/TORCH

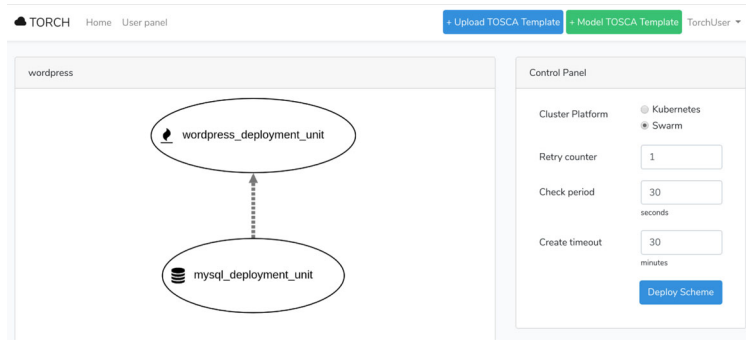


Fig. 10 Deployment panel for a Wordpress-MySQL containerised app

After starting the deployment of an application, the Dashboard allows to monitor the deployment process. In order to do so, it communicates with the BPMN Engine through the REST APIs and displays the data to the user, as in Fig. 11. For each entity of the scenario to be deployed, one of the following states is associated:

- *Success*: the entity is successfully deployed.
- *InProgress*: the entity is being deployed.
- *Waiting*: the entity needs to wait for some entities before being deployed.
- *Failed*: it was not possible to successfully deploy the entity.

4.2 Orchestration Layer

All the requests from the Dashboard that require processing or validating a YAML template are fulfilled by the TOSCA Processor. This component has a dual responsibility: the validation of the scenario, which must comply with all the TOSCA-standard requirements, and the parsing of the corresponding TOSCA graph, which comprehends defining the deployment order of the DUs based on template dependencies and extracting the properties of each DU. The implementation of the TOSCA Processor heavily relies on the OpenStack TOSCA-Parser[2], but introduces several novelties: an ad-hoc parsing of the properties and the artifacts of a node, the capability to work with encoded-file formats (i.e. Base64Encoded), and the generation of a JSON output.

The introduction of such novelties enables the integration of this component as a web-based service in our framework. The JSON output generated by the Processor encapsulates all the information of a YAML template in a compact format. One example of this approach is illustrated in Figs. 12 and 13, where we present the YAML template for a Wordpress Deployment Unit and its JSON processed version, respectively.

The *wordpress_deployment_unit* node is translated into a JSON object of type *du* which encloses the *wordpress_container* object. The properties, the capabilities and the artifacts adopted in the container YAML are mapped into the container object properties in the JSON. Finally, the external requirements of the Wordpress container are interpreted and copied in the *create* requirements section of the JSON DU and in the properties of the container object. If any volume is expected in the YAML, that would be included in the *volumes* list of the JSON. The Processor-generated description can then be fed into a BPMN plan as a Data Input, along with the deployment properties.

Nowadays, most workflow engines directly support the execution of BPMN processes. However, workflow engines usually implement only a subset of the language features and they do it differently. We chose Flowable[3] as BPMN engine. The following adjustments were necessary for the BPMN workflows to be executable by Flowable. Firstly, each process execution must be associated with a business key, which allows to identify and filter all tasks that are part of a process instance. Then, all the information concerning the process instance is usually stored and managed through process variables.

By way of illustration, Fig. 14 shows how the provision workflow for a deployment unit (depicted in Fig. 7) was modelled in Flowable. Since multiple installation instances may be involved in a provision process, synchronisation is necessary as TOSCA

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

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

Fig. 11 Deployment monitoring for a Wordpress-MySQL containerised app

nodes may have precedence constraints. For that reason, the creation of a deployment unit starts with the "await notifications" subprocess modelled as a call activity (see Fig. 15a).

This subprocess comprises, in turn, a parallel multi-instance subprocess waiting for notifications from the preceding instances. Depending on the number of DU requirements, each notification is collected by means of the "receive message" call activity referencing the process in Fig. 15b. Here, an intermediate message event catches messages with a specified name. In addition to notifications, process variables are also collected via the "merge" script task in Fig. 15a.

Once all the DU requirements are satisfied, an HTTP task invokes an appropriate Connector on the ESB in order to set off the instantiate process. Whenever the creation is faulty, an error event is thrown and then caught in the parent process. If no faults occur, recurrent checks on the DU are performed until the latter becomes available for use. Once again, an HTTP task invokes the Connector on the ESB to check the resource status on the selected container cluster platform. Upon successful completion of the DU status check, the configuration step can take place whenever configuration operations need to be executed on the deployed containers. In the absence of any errors, a

```
wordpress_container:
  type: tosca.nodes.Container.WebApplication.Wordpress
  requirements:
    - host: wordpress_deployment_unit
    - WORDPRESS_DB_HOST: mysql_container
  capabilities:
    app_endpoint:
      properties:
        port: 80
  artifacts:
    wp_image:
      file: wordpress
      type: tosca.artifacts.Deployment.Image.Container.Docker
      repository: docker_hub
  interfaces:
    Standard:
      create:
        implementation: wp_image
        inputs:
          port: { get_input: wp_host_port }
          WORDPRESS_DB_PASSWORD: { get_input: [mysql_pswd, password] }

wordpress_deployment_unit:
  type: tosca.nodes.Container.Runtime
```

Fig. 12 YAML template for a Wordpress deployment unit

```
{
   "name":"wordpress_deployment_unit",
   "type":"du",
   "requirements":{
      "create":[ "mysql_deployment_unit.configure" ],
      "configure":[]
   },
   "containers":[
      {
         "name":"wordpress_container",
         "category":"wa",
         "image":"wordpress",
         "volumes":[],
         "properties":{
            "port":"80:80",
            "WORDPRESS_DB_PASSWORD":"root",
            "WORDPRESS_DB_HOST":"mysql_deployment_unit"
         },
         "configuration_script":"no.configuration.script"
      }
   ],
   "properties":{}
}
```


Fig. 13 Processor-generated JSON for a Wordpress deployment unit

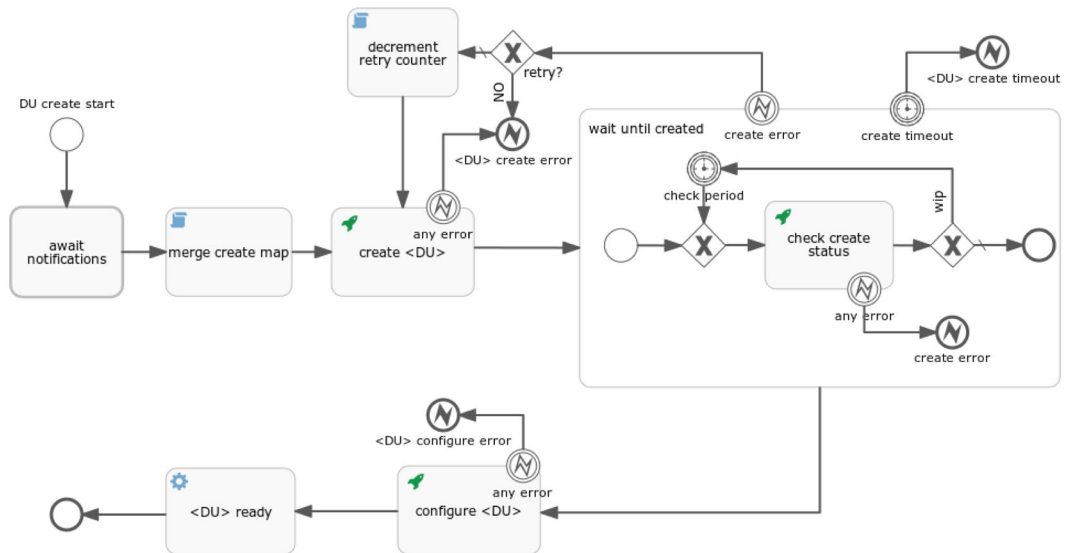


Fig. 14 Deployment Unit provision workflow in Flowable

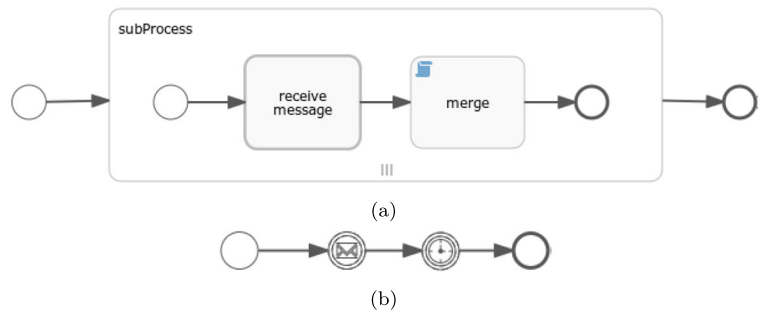


Fig. 15 (**a**) Await Notifications and (**b**) Receive Message workflows in Flowable

```
1  import it.unict.vertx.esb.du.InstantiateDU;
2  import it.unict.vertx.esb.common.Translator;
3  import io.kubernetes.client.ApiClient;
4
5  public class InstantiateDUKubernetes implements InstantiateDU
6  {
7      @Override
8      public void createDU(String nodeJson)
9      {
10         String duSpecification = Translator.translate(nodeJson,
11                                                 K8sTranslatorPlugin.class);
12         ApiClient k8sClient = initialiseClient(...);
13         String deploymentId = deployDu(k8sClient, duSpecification);
14         return deploymentId;
15     }
16     @Override
17     public void checkDU(...) { ... }
18     @Override
19     public void configureDU(...) { ... }
20 }
```

Fig. 16 Skeleton of the *InstantiateDU* connector for Kubernetes

service task sends a notification to all of the blocked instances.

4.3 Service Binding Layer

Service Connectors are Java-based pieces of software that communicate with the rest of the architecture using the Vert.x[4] REST API. Each of them implements one of the three main interfaces presented in Section 3.5. The interfaces follow the flow of the BPMN plans orchestrated by the Engine, defining the methods that would be invoked at provisioning time.

In order to grant *loose coupling* between provisioning BPMN plans and provisioning services, Connectors are provider-specific. From a practical perspective, this means that different implementations of the same interface would be needed to work on different providers.

For instance, to deploy a DU on the Kubernetes platform, an implementation of the generic *InstantiateDU* is needed. Figure 16 shows the skeleton of a potential implementation. The three methods *createDU*, *checkDU* and *configureDU*, which conform with the HTTP tasks of the *create DU* workflow (Fig. 14), need to communicate with the Kubernetes API to perform the provisioning operations. To deploy a DU on a different container cluster technology, a new Service Connector should be implemented, having the same three methods just seen, but including mechanisms to communicate with the chosen container cluster API. For the sake of clarity, a potential implementation of the *createDU* method is provided in the Figure.

In order to work with the generic Deployment Unit entity, a platform-specific Container Cluster Service Connector needs to translate from the BPMN Engine data inputs to the platform-specific application description format. To furnish a reusable asset, we provide a generic `TranslatorPlugin` Java interface. This interface contains the `translateDU` method that must be implemented to generate a DU specification in the corresponding Service Connector format.

To further facilitate the creation of translation plugins, we also included a DU *Translator* that can parse the data inputs provided by the BPMN Engine into a *DeploymentUnit* Java object. This enables to implement a translation plugin with no need to know the JSON data-input specification format. An example of how to use the Translator and translator plugins is presented at line 10 of Fig. 16, where the JSON description of a node is parsed and translated to the Kubernetes application specification format.

5 Use Case

The application modelling use case taken into consideration is Sock Shop[5], which is an open-source web

[4] https://vertx.io/

[5] https://microservices-demo.github.io/

application simulating the user-facing part of an e-commerce website that sells socks. It is a well-known multi-component demo application in the microservice landscape, which is intended to aid the demonstration and testing of microservice and cloud native technologies.

The Sock Shop application is intentionally polyglot, packaged in Docker containers and composed of 13 components. The *Front-end* displays a graphical user interface for e-shopping socks. Pairs of services and databases are used for storing and managing the catalogue of available socks (i.e., *Catalogue* and *Catalogue-DB*), the users (i.e., *Users* and *Users-DB*), the users' shopping carts (i.e., *Carts* and *Carts-DB*), and the users' orders (i.e., *Orders* and *Orders-DB*). *Payment* and *Shipping* services simulate the payment and shipping of orders, respectively. The *Queue* is a message queue that is filled with shipping requests by the Shipping service. The shipping requests are then consumed by the *Queue-Master* to simulate the actual shipping of orders.

Figure 17 displays how the Sock Shop application has been represented in TOSCA. All the containers are implemented using the *Container.Application* type or an extension of it. In particular, a few nodes take advantage of the expressiveness of the Container.Application extensions proposed in Section 3.3: the Front-end, which adopts the *Container.WebApplication* type, and the database-typed containers, which extend from the proposed *Container.Database* type. Furthermore, each of the nodes presents one or more dependencies: all of them have a *HostedOn* relationship with respect to a dedicated *Container.Runtime* node, which in the Figure is depicted as part of the node itself, and some nodes have *DependsOn* relationships, which are used to describe the links between the nodes and define a deployment order.

We adopted the Sock Shop scenario as an exemplary use case to assess the capabilities and the performance of TORCH. Our testbed consists of a machine running the full framework, along with the

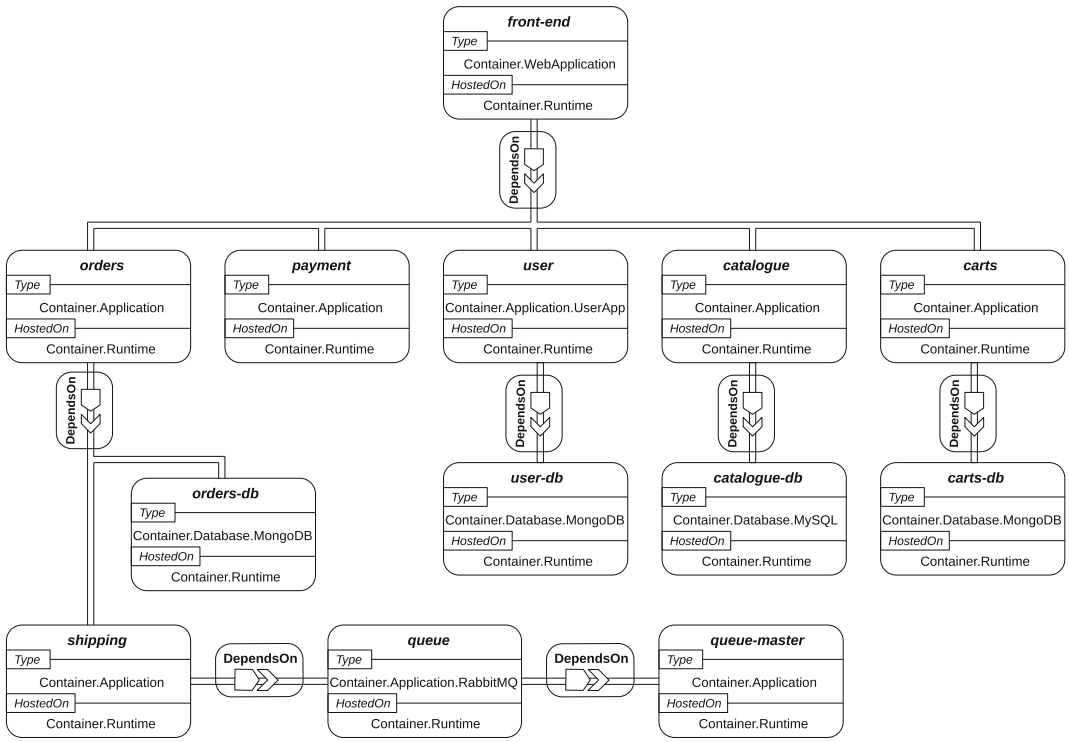


Fig. 17 Sock Shop container-based application representation

Table 1 Steady state resource statistics about the testing system

	CPU avg	MEM avg	NET IN avg	NET OUT avg
no activity	0.39%	1.53 GiB	32.9 kbps	0 kbps
TORCH running	0.94%	4.89 GiB	33.4 kbps	0kbps

monitoring tools to collect some metrics about the system performance.

We used the Prometheus[6] toolkit together with the Node Exporter[7] 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 x86-64 Linux distribution.

Table 1 displays the average values for the metrics when the machine is found in a steady state with and without the TORCH framework running. The metrics have been collected on a 5-minutes time frame and averaged over time, to prevent occasional system processes from interfering with our analysis. The table shows that TORCH 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 TORCH in an end-to-end fashion. The testbed machine, hosting the whole framework, is accessed to upload the Sock Shop 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.

The OpenStack cluster is deployed on off-the-shelf PCs 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 x86-64 Linux distribution.

The deployment of Sock Shop is tested on two different container cluster technologies: Kubernetes and Docker Swarm, whose cluster deployments are available through the OpenStack Magnum service. All the clusters created for the tests were composed of a master node and a slave node, both equipped with 4 GB RAM and 100 GB of storage.

To communicate with all these platforms, four Service Connectors have been implemented: two OpenStack *InstantiateCluster* connectors to create each of the clusters, using the OpenStack4J library[8], the Kubernetes *InstantiateDU* connector, using the official Kubernetes Java client[9], and the Swarm *InstantiateDU* connector, using the Spotify Docker client[10].

The Sock Shop application has been correctly deployed on both Kubernetes and Swarm. The deployment times are shown in Table 2. The table displays the average times $\pm$ 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 of Kubernetes. We detail the testbed machine resource usage in Fig. 18. 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.

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.

[6] https://prometheus.io/

[7] https://github.com/prometheus/node_exporter

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

[9] https://github.com/kubernetes-client/java

[10] https://github.com/spotify/docker-client

Table 2 Deployment average times

	Kubernetes		Docker Swarm	
	Value	Percentage	Value	Percentage
create cluster	5min 31s ± 3s	50.22%	4min 3s ± 15s	53.29%
create du	5min 28s ± 8s	49.78%	3min 33s ± 9s	46.71%
total	10min 59s ± 13s	100%	7min 36s ± 5s	100%

As for the plots, this stage implies a small increase in the CPU usage and a high peak in the network transmitted traffic.

The cluster creation phase entails an increase in the CPU usage, reaching peaks of 2-2.5%. There are no heavy network requests, but steady 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 the largest CPU increase over time. This is mainly due to the parallelism capabilities of TORCH, 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.

The results of our test show that TORCH is a computationally inexpensive framework. 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, TORCH 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. Finally, we noticed a regular network usage during all the provisioning phases.

Because of hardware constraints, we were able to test the framework only on single-cluster and single-scenario use cases. However, looking at the achieved performance, TORCH promises to efficiently scale also for the deployment of several applications, concurrently. Anyway, extensive scalability tests will be part of our future work.

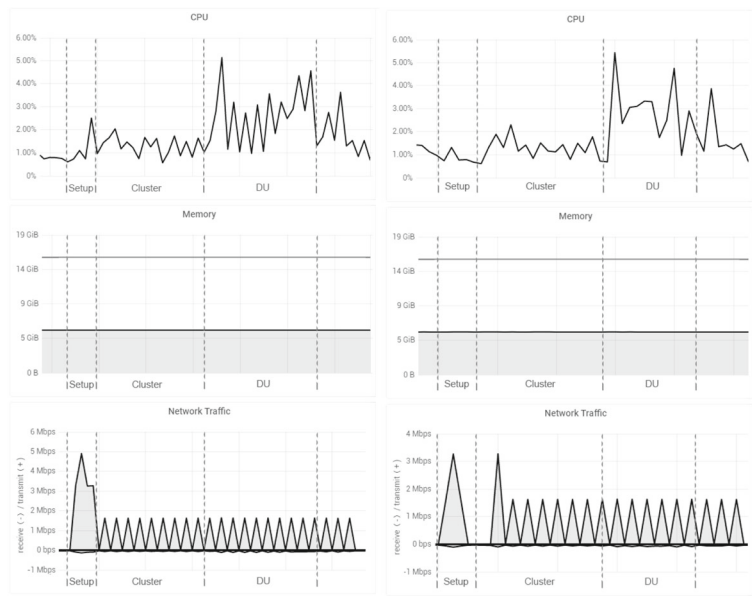


Fig. 18 Testbed machine resource usage. (**a**) Kubernetes deployment. (**b**) Swarm deployment

6 Related Work

In this section, we position TORCH in respect to other business-oriented and research projects for orchestrating multi-component cloud applications with containers and/or TOSCA.

Cloudify [29] delivers container orchestration integrating multiple technologies and providers. Despite graphical tools for sketching and modelling an application, its data format is based on the TOSCA standard. Alien4Cloud [52] 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 works implement the interoperability between different clusters and providers by defining complex sets of nodes in a technology-specific way. Their TOSCA implementations rely on Domain-Specific Languages (DSLs) which, despite sharing the TOSCA template structure, divert from the node type hierarchy defined in the standard. With respect to Cloudify, TORCH focuses on TOSCA-compliant application descriptions, making no prior assumptions regarding the technology stack to be established.

Apache ARIA TOSCA [53] 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 support. Compared to TORCH, the derived node types still lack the possibility to use role-specific containers (see Section 3.3). In addition, no cluster orchestrators are natively supported.

Apache Brooklyn [41] is an open-source framework for modelling, deploying, and managing distributed applications using declarative blueprints written in Brooklyn's DSL. Brooklyn's YAML format follows the CAMP specification, but it uses a few custom extensions. With respect to TORCH, Brooklyn provides no native support for TOSCA yet, even though some work [54] has been done over the past years. Containerisation is not supported out of the box, but it can be integrated via separate projects (e.g, Cloudsoft Clocker [55]).

Apache Stratos [56] is an open-source PaaS framework which allows developers to build distributed applications and services. It defines configurations and applications in a specific JSON format, and leverages Kubernetes as a cluster orchestration framework in order to provide containerisation. As compared with TORCH, Stratos does not support either TOSCA or any containerised clusters other than Kubernetes.

In [57] 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 [58] presents an approach which leverages the TOSCA standard for the deployment of Docker-based applications. TosKer approach is very different from the one proposed in this paper, 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. Clustered scenarios are also out of the picture, even though some recent work [59] has been done to deploy applications on top of existing Docker-based container orchestrators.

In [60] 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 cluster deployment.

In [61] the authors present a cloud resource deployment framework which leverages a unified configuration knowledge-base, where a process modelling notation, implemented by extending BPMN, is used to describe configurations and orchestration of federated cloud resources. Our approach differs from this work in two ways. On the one hand, resource modelling is based on TOSCA; on the other one, resource orchestration is grounded on native BPMN.

In [62] the authors introduce Cafe, a framework to model, configure and automatically provision composite cloud applications. Based on the application and variability models, cloud applications can be provisioned via a generic provisioning flow. Both TORCH and Cafe exploits pre-modelled configurable workflows for deployment orchestration. Whereas the former employs TOSCA-based application modelling and BPMN-based orchestration, the latter employs XML-based application modelling and BPEL-based orchestration.

Other approaches worth mentioning are OpenTOSCA [63], Cloudiator [64, 65], Roboconf [66], Indigo-DataClouds [67, 68], MODAClouds [69, 70], and SeaClouds [71, 72].

OpenTOSCA is a famous open-source ecosystem for modelling, deploying and managing TOSCA-based applications. It mainly consists of the following tools: Winery [73], OpenTOSCA Container [74] and Vinothek [75]. Winery is a graphical modelling tool to create both TOSCA topology models and management plans using the workflow language BPMN4TOSCA [76]. The OpenTOSCA Container is a runtime environment for the provisioning and management of TOSCA applications packaged as CSARs. Applications can be automatically deployed based on the contained artifacts, the Plan Generator [77] and the Management Bus [78]. Vinothek is a self-service portal which enables to list, select and instantiate TOSCA-based applications. Our work bears some similarities with OpenTOSCA. Both TORCH's Dashboard and Winery allow to model and/or import/export TOSCA applications as CSARs, but the former also enables to configure, trigger and monitor application deployments. Both TORCH and OpenTOSCA separate management plans (BPMN plans for the former, BPEL plans for the latter) from management operations by means of a bus hiding all the details of the employed technologies. While TORCH is extensible by adding new Connectors to the Service Binding Layer, OpenTOSCA is extensible by adding new plugins to the Management Bus.

Cloudiator is an open-source cross-cloud orchestration framework supporting many public and private cloud platforms. The application description consists of individual components, which are assembled to form a full application and run within Docker containers. Roboconf is an open-source scalable orchestration framework for multi-cloud platforms. Many IaaS providers as well as Docker containers are supported by using special plugins. Roboconf describe applications and their execution environments in a hierarchical way by means of a CSS-inspired DSL. However, in contrast to TORCH, both Cloudiator and Roboconf do not comply with either TOSCA or any other modelling standards.

INDIGO-DataCloud is an open-source data and computing platform, targeted at scientific communities, 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 it does not support other containerised clusters.

MODAClouds is an open-source design-time and run-time platform for developing and operating multi-cloud applications with guaranteed QoS. Despite leveraging a Model Driven Engineering (MDE) approach in order to support interoperability between cloud providers, no support is provided for TOSCA. SeaClouds is an open-source middleware solution for deploying and managing multi-component applications on heterogeneous clouds. As opposed to MODAClouds, SeaClouds fully supports TOSCA. However, both works lack a support for Docker containers, which makes them not suitable for orchestrating the management of multi-component applications including Docker containers.

In summary, all the current works achieve container cluster interoperability, or partial interoperability, either by linking platform-specific information to the nodes of the topology template or by redefining the TOSCA standard 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. Our work differ from all existing works in terms of high interoperability between different technologies and providers, no overturning of the standard-defined types, and no prior assumptions about the technology stack to be established.

7 Conclusion and Future Work

The ever-growing popularity of cloud computing among both research and industry communities has fostered a business landscape populated by quite a number of cloud providers, competing with each other but offering similar services in terms of functionality. Customers usually entrust cloud service brokers to pick the cloud services that best suit their own needs and to properly manage the life-cycle of their cloud applications. In the last decade, a number of cloud orchestration tools have appeared on the scene promising to facilitate the operations underlying the entire application's life-cycle management.

In this paper we have introduced TORCH, a TOSCA-enabled cloud orchestrator capable of seamlessly interfacing with multiple providers of cloud resources and provisioning services. With respect to other cloud orchestrators, TORCH stands out for its innovative approach that abstracts the provisioning workflow away from the actual invocation of the provisioning services. This separation enables the "design once, deploy anywhere" pattern, according to which the application provisioning workflow is decoupled from the specific cloud provider's API, while ad-hoc developed software connectors are responsible for invoking actual proprietary API. Exploiting the separation of concerns, we were able to entrust the provisioning activities to an off-the-shelf workflow engine.

The benefit of this approach is two-fold: code development efforts just need to focus on the implementation of connectors; increased workloads can be easily accommodated, since the workflow engine natively exhibits scaling capabilities. A software prototype of the framework was developed and experiments were conducted on a small-scale test-bed. Preliminary results prove the viability of the approach. In the future, tests will be extended to different cloud providers and carried out in a larger environment to assess the maturity of the solution at industrial scale.

Acknowledgments This work has been partially funded by University of Catania through the research project "MANGO - Piaceri 2020-2022 - Linea 2".

Funding Open Access funding provided by Universit degli Studi di Catania

Data Availability Data sharing not applicable to this article as no datasets were generated or analysed during the current study

References

1. Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., Ghalsasi, A.: Cloud computing, -the business perspective. Decis. Support. Syst. **51**(1), 176 (2011). https://doi.org/10.1016/j.dss.2010.12.006
2. Dikaiakos, M.D., Katsaros, D., Mehra, P., Pallis, G., Vakali, A.: Cloud Computing: Distributed Internet Computing for IT and Scientific Research. IEEE Internet Computing **13**(5), 10 (2009). https://doi.org/10.1109/MIC.2009.103
3. Flexera: Flexera 2020 State of the cloud report. https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020. Last accessed on 18-06-2020 (2020)
4. Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N.C., Hu, B.: Everything as a Service (XaaS) on the Cloud: Origins, current and future trends. In: 8th International Conference on Cloud Computing, pp. 621–628 (2015). https://doi.org/10.1109/CLOUD.2015.88
5. Ranjan, R., Benatallah, B., Dustdar, S., Papazoglou, M.P.: Cloud resource orchestration programming: Overview, issues, and directions. IEEE Internet Computing **19**, 46 (2015). https://doi.org/10.1109/MIC.2015.20
6. Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C.B., Domaschka, J.: Cloud Orchestration Features: Are Tools Fit for Purpose? In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC, 2015, pp. 95–101 (2015). https://doi.org/10.1109/UCC.2015.25
7. Bousselmi, K., Brahmi, Z., Gammoudi, M.M.: Cloud services orchestration: A comparative study of existing approaches. In: IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, (WAINA 2014), pp. 410–416 (2014). https://doi.org/10.1109/WAINA.2014.72
8. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: 2013 IEEE Sixth International Conference on Cloud Computing, pp. 887–894 (2013). https://doi.org/10.1109/CLOUD.2013.133
9. Petcu, D.: Consuming resources and services from multiple clouds. Journal of Grid Computing **12**(2), 321 (2014). https://doi.org/10.1007/s10723-013-9290-3
10. Petcu, D., Vasilakos, A.: Portability in Clouds: Approaches and Research Opportunities. Scalable Computing: Practice and Experience **15**(3), 251 (2014). https://doi.org/10.12694/scpe.v15i3.1019
11. Ferry, N., Rossini, A.: CloudMF: Model-driven management of multi-cloud applications. ACM Tran. Internet Technol **18**(2), 16 (2018). https://doi.org/10.1145/3125621
12. OASIS: Cloud application management for platforms version 1.1. http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html. Last accessed on 15-02-2017 (2014)
13. OASIS: 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 on 10-04-2018 (2013)
14. Bellendorf, J., Mann, Z.Á.: Cloud topology and orchestration using TOSCA: A systematic literature review. In: Kritikos, K., Plebani, P., de Paoli, F. (eds.) Service-Oriented

and Cloud Computing, pp. 207–215. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-99819-0_16
15. Pahl, C.: Containerization and the PaaS Cloud. IEEE Cloud Computing **2**(3), 24 (2015). https://doi.org/10.1109/MCC.2015.51
16. Ruan, B., Huang, H., Wu, S., Jin, H.: A performance study of containers in cloud environment. In: Wang, G., Han, Y., Martínez Pérez, G. (eds.) Advances in Services Computing, pp. 343–356. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-49178-3_27
17. Docker Inc.: Docker Swarm. https://docs.docker.com/engine/swarm/. Last accessed on 25-06-2020
18. C.N.C. Foundation. Kubernetes. https://kubernetes.io/. Last accessed on 25-06-2020
19. Apache Software Foundation. Mesos. http://mesos.apache.org/. Last accessed on 25-06-2020
20. 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) (Funchal-Madeira (Portugal)), pp. 533–540 (2018). https://doi.org/10.5220/0006775605330540
21. Calcaterra, D., Cartelli, V., Di Modica, G., Tomarchio, O.: Implementation of a fault aware cloud service provisioning framework. In: 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 9–16 (2018). https://doi.org/10.1109/FiCloud.2018.00010
22. 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 (2020). https://doi.org/10.5220/0009410701270137
23. Weerasiri, D., Barukh, M.C., Benatallah, B., Sheng, Q.Z., Ranjan, R.: Taxonomy and survey of cloud resource orchestration techniques. ACM Comput. Surv. **50**(2), 26:1 (2017). https://doi.org/10.1145/3054177
24. Amazon. CloudFormation. https://aws.amazon.com/cloudformation/. Last accessed on 25-06-2020
25. Flexera software. Cloud management platform. https://www.flexera.com/products/agility/cloud-management-platform.html. Last accessed on 25-06-2020
26. RedHat. CloudForms. https://access.redhat.com/products/red-hat-cloudforms. Last accessed on 25-06-2020
27. IBM. Cloud Orchestrator. https://www.ibm.com/us-en/marketplace/deployment-automation. Last accessed on 25-06-2020
28. OpenStack. Heat. https://wiki.openstack.org/wiki/Heat. Last accessed on 25-06-2020
29. GigaSpaces Technologies. Cloudify. https://cloudify.co/. Last accessed on 25-06-2020
30. Red Hat. Ansible. https://www.ansible.com/. Last accessed on 25-06-2020
31. Chef. Chef. https://www.chef.io/. Last accessed on 25-06-2020
32. Puppet. Puppet. https://puppet.com/. Last accessed on 25-06-2020
33. SaltStack Inc. Salt. https://www.saltstack.com/. Last accessed on 25-06-2020
34. Morris, K. Infrastructure as Code: Managing Servers in the Cloud, 1st edn. O'Reilly Media, Inc, Newton (2016)
35. HashiCorp. Terraform. https://www.terraform.io/. Last accessed on 25-06-2020
36. OASIS: Web services business process execution language version 2.0. https://www.oasis-open.org/committees/wsbpel/. Last accessed on 10-04-2018 (2007)
37. OMG: Business Process Model and Notation (BPMN 2.0). http://www.omg.org/spec/BPMN/2.0/. Last accessed on 10-04-2018 (2011)
38. Vargas-Santiago, M., Hernández, S.E.P., Rosales, L.A.M., Kacem, H.H.: Survey on web services fault tolerance approaches based on checkpointing mechanisms. JSW **12**(7), 507 (2017). https://doi.org/10.17706/jsw.12.7.507-525
39. Kolb, S., Wirtz, G.: Towards application portability in platform as a service. In: IEEE 8th International symposium on service oriented system engineering, pp. 218–229. https://doi.org/10.1109/SOSE.2014.26 (2014)
40. Oberle, K., Fisher, M.: ETSI CLOUD - Initial standardization requirements for cloud services. In: Proceedings of the 7th International Conference on Economics of Grids, Clouds, Systems, and Services, GECON'10, pp. 105–115. https://doi.org/10.1007/978-3-642-15681-6_8. Springer (2010)
41. Apache software foundation. Brooklyn. https://brooklyn.apache.org/. Last accessed on 25-06-2017
42. OASIS: TOSCA Simple Profile in YAML Version 1.3. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3. Last accessed on 26-02-2020 (2020)
43. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems 2007, EuroSys '07, pp. 275–287. https://doi.org/10.1145/1272996.1273025 (2007)
44. 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 (2016). https://doi.org/10.1109/ICATCCT.2016.7912109
45. Docker Inc. Docker. https://www.docker.com/. Last accessed on 25-06-2020
46. Sysdig: Sysdig 2019 container usage report. https://sysdig.com/blog/sysdig-2019-container-usage-report/. Last accessed: 2019-12-23 (2019)
47. Cloud native computing foundation. containerd. https://containerd.io/. Last accessed on 25-06-2020
48. Cloud native computing foundation. CRI-O. https://cri-o.io/. Last accessed on 25-06-2020
49. Apache Software Foundation. Mesos Containerizer. http://mesos.apache.org/documentation/latest/mesos-containerizer/. Last accessed on 25-06-2020
50. Mesosphere Inc. Marathon. https://mesosphere.github.io/marathon/. Last accessed on 25-06-2020
51. OpenStack. Magnum. https://wiki.openstack.org/wiki/Magnum. Last accessed on 25-06-2020
52. FastConnect. Alien4Cloud. https://alien4cloud.github.io/. Last accessed on 25-06-2020

53. Apache Software Foundation. ARIA TOSCA. https://ariatosca.incubator.apache.org/. Last accessed on 25-06-2020
54. Carrasco, J., Cubo, J., Durán, F., Pimentel, E.: Bidimensional cross-cloud management with TOSCA and Brooklyn. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp. 951–955 (2016). https://doi.org/10.1109/CLOUD.2016.0143
55. Cloudsoft. Clocker. http://www.clocker.io/. Last accessed on 25-06-2020
56. Apache Software Foundation. Stratos. https://stratos.apache.org/. Last accessed on 25-06-2020
57. Kiss, T., Kacsuk, P., Kovacs, J., Rakoczi, B., Hajnal, A., Farkas, A., Gesmier, G., Terstyanszky, G.: MiCADO—Microservice-based cloud application-level dynamic orchestrator. Futur. Gener. Comput. Syst. **94**, 937 (2019). https://doi.org/10.1016/j.future.2017.09.050
58. Brogi, A., Rinaldi, L., Soldani, J.: Tosker: A synergy between tosca and docker for orchestrating multicomponent applications. Software: Practice and Experience **48**(11), 2061 (2018). https://doi.org/10.1002/spe.2625
59. Bogo, M., Soldani, J., Neri, D., Brogi, A.: Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes. Software: Practice and Experience **50**(9), 1793–1821 (2020). https://doi.org/10.1002/spe.2848
60. Kehrer, S., Blochinger, W.: TOSCA-based container orchestration on Mesos. Comput. Sci. Res. Dev. **33**(3), 305 (2018). https://doi.org/10.1007/s00450-017-0385-0
61. Weerasiri, D., Benatallah, B., Barukh, M.C.: Process-driven configuration of federated cloud resources. In: Renz, M., Shahabi, C., Zhou, X., Cheema, M.A. (eds.) Database Systems for Advanced Applications, pp. 334–350. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-18120-2_20
62. Mietzner, R., Leymann, F.: A self-service portal for service-based applications. In: 2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–8 (2010). https://doi.org/10.1109/SOCA.2010.5707165
63. Breitenbücher, U., Endres, C., Képes, K., Kopp, O., Leymann, F., Wagner, S., Wettinger, J., Zimmermann, M.: The opentosca ecosystem - concepts & tools. In: European space project on smart systems, big data, future internet - Towards serving the grand societal challenges - volume 1: EPS Rome 2016,. INSTICC, pp. 112–130. SciTePress (2016). https://doi.org/10.5220/0007903201120130
64. Baur, D., Domaschka, J.: Experiences from building a cross-cloud orchestration tool. In: Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms, CrossCloud '16, pp. 4:1–4:6. https://doi.org/10.1145/2904111.2904116. ACM (2016)
65. Baur, D., Seybold, D., Griesinger, F., Masata, H., Domaschka, J.: A provider-agnostic approach to multi-cloud orchestration using a constraint language. In: 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 173–182 (2018). https://doi.org/10.1109/CCGRID.2018.00032
66. Pham, L.M., Tchana, A., Donsez, D., de Palma, N., Zurczak, V., Gibello, P.: Roboconf: A hybrid cloud orchestrator to deploy complex applications. In: IEEE 8th International Conference on Cloud Computing, pp. 365–372 (2015). https://doi.org/10.1109/CLOUD.2015.56
67. Caballer, M., Zala, S., García, Á.L., Moltó, G., Fernández, P.O., Velten, M.: Orchestrating complex application architectures in heterogeneous clouds. Journal of Grid Computing **16**(1), 3 (2018). https://doi.org/10.1007/s10723-017-9418-y
68. Salomoni, D., Campos, I., Gaido, L., et al.: INDIGO-DataCloud: a platform to facilitate seamless access to e-infrastructures. Journal of Grid Computing **16**(3), 381 (2018). https://doi.org/10.1007/s10723-018-9453-3
69. Ardagna, D., Di Nitto, E., Mohagheghi, P., Mosser, S., Ballagny, C., D'Andria, F., Casale, G., Matthews, P., Nechifor, C., Petcu, D., Gericke, A., Sheridan, C.: MODAClouds: a model-driven approach for the design and execution of applications on multiple clouds. In: 4th International Workshop on Modeling in Software Engineering (MISE), pp. 50–56 (2012). https://doi.org/10.1109/MISE.2012.6226014
70. Di Nitto, E., Matthews, P., Petcu, D., Solberg, A.: Model-driven Development and Operation of Multi-cloud Applications: The MODAClouds approach. Springer International Publishing, Berlin (2017)
71. Brogi, A., Carrasco, J., Cubo, J., D'Andria, F., Ibrahim, A., Pimentel, E., Soldani, J.: SeaClouds: Seamless adaptive multi-cloud management of service-based applications. In: 17th Conferencia Iberoamericana en Software Engineering (CIbSE 2014), pp. 95–108 (2014)
72. Brogi, A., Fazzolari, M., Ibrahim, A., Soldani, J., Wang, P., Carrasco, J., Cubo, J., Duran, F., Pimentel, E., Di Nitto, E., D'Andria, F.: Adaptive management of applications across multiple clouds: the SeaClouds approach. CLEI Electronic Journal **18**, 2 (2015). https://doi.org/10.19153/cleiej.18.1.1
73. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – a modeling tool for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) Service-Oriented Computing, pp. 700–704. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-45005-1_64
74. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S., OpenTOSCA –, A.: Runtime for TOSCA-Based Cloud Applications, pp. 692–695. Springer, Berlin (2013)
75. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F.: Vinothek - A self-service portal for TOSCA. In: Herzberg, N., Kunze, M. (eds.) Proceedings of the 6th Central-European Workshop on Services and their Composition, ZEUS 2014, Potsdam, Germany, February 20-21, 2014, CEUR Workshop Proceedings, vol. 1140, pp. 69–72. CEUR-WS.org (2014)
76. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications, pp. 38–52. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-33155-8_4
77. Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., Wettinger, J.: Combining declarative and imper-

ative cloud application provisioning based on TOSCA. In: 2014 IEEE International Conference on Cloud Engineering, pp. 87–96 (2014). https://doi.org/10.1109/IC2E.2014.56
78. Wettinger, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: Streamlining cloud management automation by unifying the invocation of scripts and services based on TOSCA. Int. J. Organ. Collect. Intell. **4**(2), 45 (2014). https://doi.org/10.4018/ijoci.2014040103