Hochschule Reutlingen
Reutlingen University

Parallel and Distributed Computing Group
Department of Computer Science
Reutlingen University

TOSCA-based Container Orchestration on Mesos

Two-Phase Deployment of Cloud Applications using Container-based Artifacts

Stefan Kehrer, Wolfgang Blochinger
Reutlingen University
{stefan.kehrer, wolfgang.blochinger}@reutlingen-university.de

```
@Article{Kehrer2018,
       author="Kehrer, Stefan and Blochinger, Wolfgang",
       title="TOSCA-based container orchestration on Mesos",
       journal="Computer Science - Research and Development",
       year="2018",
       month="Aug",
       day="01",
       volume="33",
       number="3",
       pages="305--316",
       issn="1865-2042",
       doi="10.1007/s00450-017-0385-0",
       url="https://doi.org/10.1007/s00450-017-0385-0"
}
```

The final publication is available at Springer via https://doi.org/10.1007/s00450-017-0385-0

Original Publication Reference:
https://link.springer.com/article/10.1007/s00450-017-0385-0

Computer Science - Research and Development manuscript No.
(will be inserted by the editor)

TOSCA-based Container Orchestration on Mesos

Two-Phase Deployment of Cloud Applications using Container-based Artifacts

Stefan Kehrer · Wolfgang Blochinger

Received: date / Accepted: date

Abstract Container virtualization evolved into a key technology for deployment automation in line with the DevOps paradigm. Whereas container management systems facilitate the deployment of cloud applications by employing container-based artifacts, parts of the deployment logic have been applied before to build these artifacts. Current approaches do not integrate these two deployment phases in a comprehensive manner. Limited knowledge on application software and middleware encapsulated in container-based artifacts leads to maintainability and configuration issues. Besides, the deployment of cloud applications is based on custom orchestration solutions leading to lock-in problems.

In this paper, we propose a two-phase deployment method based on the TOSCA standard. We present integration concepts for TOSCA-based orchestration and deployment automation using container-based artifacts. Our two-phase deployment method enables capturing and aligning all the deployment logic related to a software release leading to better maintainability. Furthermore, we build a container management system, which is composed of a TOSCA-based orchestrator on Apache Mesos, to deploy container-based cloud applications automatically.

Keywords Container Orchestration · Two-Phase Deployment · TOSCA · Container-based Artifacts · Apache Mesos · DevOps

Stefan Kehrer
Department of Computer Science, Reutlingen University, Alteburgstr. 150, 72762 Reutlingen, Germany
E-mail: stefan.kehrer@reutlingen-university.de

Wolfgang Blochinger
Department of Computer Science, Reutlingen University, Alteburgstr. 150, 72762 Reutlingen, Germany
E-mail: wolfgang.blochinger@reutlingen-university.de

1 Introduction

Fast software release cycles are an essential business requirement today. To this end, the DevOps paradigm [1] proposes close collaboration of development and operations personnel. In this context, deployment automation is crucial to avoid error-prone, manual processes [2]. Tools to support deployment automation typically rely on DevOps artifacts (e.g., scripts or templates), which encapsulate the required deployment logic. Basically, two classes of DevOps artifacts have been identified [3]: *Node-centric artifacts* contain deployment logic executed on a single node, whereas *environment-centric artifacts* specify deployment logic comprising multiple nodes. Both types of artifacts have to be integrated to deploy complex multi-node application topologies.

To foster the DevOps paradigm, container virtualization gained momentum during the last years. Implementations such as Docker[1] provide a new node-centric artifact: Node-centric deployment logic is specified (in a Dockerfile) and employed to build a container-based artifact (Docker image). Subsequently, multiple of those artifacts are used to deploy an application topology.

This trend led to the development of *container management systems*, which are used to deploy container-based cloud applications across a pool of (virtual) machines. Prominent open-source implementations are Marathon[2] on Apache Mesos[3], Kubernetes[4], and Docker Swarm[5]. Moreover, cloud services such as Amazon EC2 Container Service[6] are available.

[1] https://www.docker.com.
[2] https://mesosphere.github.io/marathon.
[3] https://mesos.apache.org.
[4] https://kubernetes.io.
[5] https://github.com/docker/swarm.
[6] https://aws.amazon.com/de/ecs.

For deploying multi-node application topologies, current container management systems and services utilize custom orchestration solutions. These systems operate on container-based artifacts and thus are unaware of node-centric deployment logic encapsulated in these artifacts. This leads to two major problems: (1) Orchestration solutions employ custom environment-centric artifacts, e.g., templates based on domain-specific languages, to orchestrate containers leading to lock-in problems [4]. (2) These systems do not provide comprehensive solutions how to align and integrate node-centric and environment-centric deployment logic. Developers are responsible for mapping the deployment requirements of encapsulated components to environment-centric artifacts (e.g., templates) and tooling [5]. Limited knowledge on the components inside a container leads to maintainability and configuration issues.

The emerging standard TOSCA (Topology and Orchestration Specification for Cloud Applications) [6] enables portable orchestration of automated deployment operations. TOSCA defines a modeling language to specify application topologies and related management operations to describe portable cloud applications.

In this paper, we propose a two-phase deployment method to integrate TOSCA-based orchestration and deployment automation using container-based artifacts. We address lock-in problems inherent to current container management systems by leveraging the TOSCA standard. Further, we elaborate on the integration of node-centric and environment-centric deployment logic to support software developers in line with the DevOps paradigm. Our approach integrates automated deployment of container-based cloud applications and the maintainability of deployment logic required to enable automation. In particular, our contributions are as follows:

- We introduce a *two-phase deployment method* and corresponding concepts to integrate node-centric and environment-centric deployment. In this context, we propose novel TOSCA-based modeling constructs.
- We present a *TOSCA-based container management system* to support two-phase deployment. It is implemented as a prototypical TOSCA-based orchestrator on top of Apache Mesos.

Our paper is structured as follows. Section 2 describes the state of the art. In Section 3 we present our two-phase deployment method. We derive integration concepts and specify requirements for a container management system in Section 4. Our TOSCA-based container management system is presented in Section 5. Section 6 discusses our approach. Section 7 deals with related work. Finally, we conclude in Section 8.

2 State of the art

In this section, we present the state of the art in container virtualization, container orchestration based on Mesos, and TOSCA-based orchestration.

2.1 Container virtualization and Docker

Conventional machine-level virtualization techniques virtualize the ISA system interface for sharing resources of a physical host between several guest virtual machines each executing an individual operating system. On the other hand, container virtualization establishes a sandboxed execution environment for a group of processes (typically representing an application) on the operating system level. A host can execute multiple containers independently. Isolation is enforced by operating system mechanisms like control groups and namespaces. While machine level virtualization provides a higher degree of isolation, container virtualization requires less resources (especially memory) such that the number of containers a machine can host is considerably higher compared to the number of possible virtual machines.

Docker, an open-source implementation of container virtualization, gained momentum during the last years. Docker is designed for application development and facilitates the DevOps paradigm by ensuring portability of applications. An application can be described using a *Dockerfile* representing the build configuration. The Docker build process creates a *Docker image*, which defines a self-contained artifact to deploy a *Docker container* running one or more application or middleware components. Essentially, this means that parts of the configuration (usually applied during deployment) are applied during the build process. The resulting image captures an ordered collection of filesystem changes and a runtime configuration for the container. Thus, the context of an application or middleware component in a container is the same in both development and production environments [7].

Employing container-based artifacts to deploy and operate cloud applications makes management operations throughout the whole lifecycle of a container instance necessary and requires advanced orchestration support [5].

2.2 Container orchestration based on Mesos

In this section, we describe how container orchestration can be implemented on top of Mesos. Mesos represents a new way of sharing computing resources across container-based applications [8]. It employs container

virtualization techniques to ensure isolation on operating system level. Since containers are used to define execution contexts for application components, computing resources provided by virtual machines can be assigned to applications without additional configuration efforts.

As a result, a new abstraction layer is introduced, which ensures virtual machine management in a transparent manner and provides an interface for resource scheduling. Additionally, Mesos ensures container runtime provisioning for given container images. Custom *Mesos Frameworks* can be built to allocate resources according to resource requirements of applications managed by Mesos Frameworks.

Marathon is an open-source Mesos Framework for container orchestration. It automates deployment and forwards application-specific resource requirements to Mesos. Container orchestration based on Mesos leads to the following benefits for cloud application deployment: (1) Virtual machines are managed transparently; (2) fast application provisioning by allocating available resource pools; (3) efficient utilization of resources by globally synchronized, multi-tenant resource sharing; (4) Resources are available at any granularity.

Marathon enables the deployment of cloud applications, which are specified using a JSON-based template. *Dependencies* can be used to specify relationships to other containers. Other orchestration solutions such as Kubernetes and Docker Swarm provide similar functionality. However, custom orchestration solutions lead to lock-in problems [4].

2.3 TOSCA-based orchestration

Portable orchestration of automated management operations is an important issue that needs to be addressed to prevent lock-in problems comprehensively. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [6] aims at standardizing a language for portable cloud applications that supports orchestration of automated management operations. In this context, the TOSCA Simple Profile in YAML [9] provides a human-readable modeling language based on YAML.

TOSCA enables the specification of cloud applications in a provider-independent manner to facilitate application migration between cloud providers. Therefore, cloud applications are captured as topology graphs and plans defining management tasks. Application topologies can be specified by means of a *service template*. The nodes of the topology graph represent components of a cloud application and the edges their relationships. These are the basic types of the TOSCA topology model and can be used to model a cloud application.

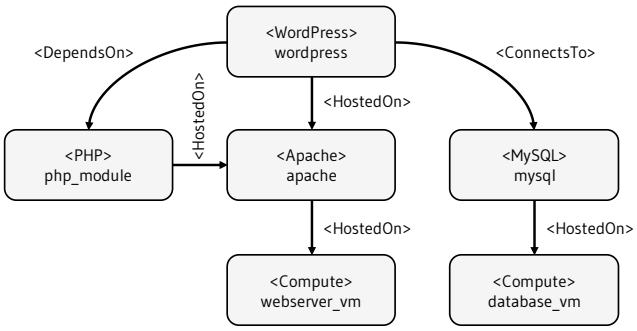

Fig. 1 Topology of an exemplary cloud application

Figure 1 shows the topology of an exemplary cloud application consisting of an Apache HTTP server running a WordPress installation and a MySQL database, hosted on a virtual machine each. WordPress makes use of the PHP Module hosted on Apache HTTP server and connects to the database. Nodes and relationships can be described in a *topology template* by using *node templates* and *relationship templates*. The TOSCA type system provides *node types* and *relationship types* that can be used as building blocks to model an application topology using the *topology template*. The *node types* *WebApplication.WordPress* and *Database.MySQL* may be used to model the nodes depicted in Figure 1. Furthermore, the normative *relationship types* *HostedOn*, *DependsOn*, and *ConnectsTo* exist.

The resulting *topology template* is part of the *service template*. Since a *topology template* is an abstract description of a cloud application topology, *deployment artifacts* such as VM images are linked to *node types* as depicted in Figure 2. Both, *node types* and *relationship types* define *lifecycle operations*. The Simple Profile in YAML [9] defines a set of normative *lifecycle operations* such as *create*, *configure*, and *delete*. *Lifecycle operations* are implemented by *implementation artifacts* such as shell scripts (cf. Figure 2) and used to automate management tasks.

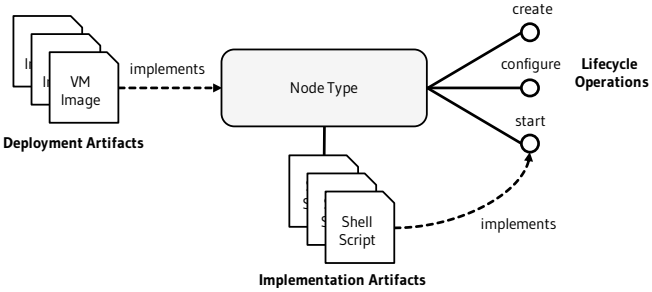

Fig. 2 Deployment and implementation artifacts in TOSCA

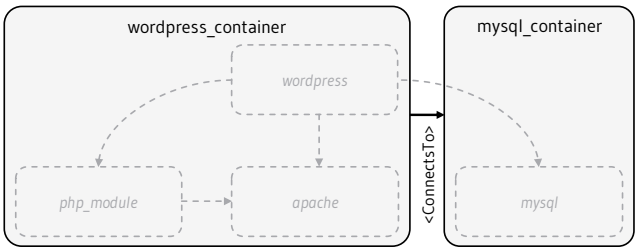

Fig. 3 Topology of a container-based example application

A TOSCA *orchestrator* processes a *service template* by instantiating and managing nodes and their relationships by invoking the *lifecycle operations* defined in the *topology template*. The processing of *service templates* can be either imperative or declarative. Imperative processing relies on *management plans* that can be executed automatically and orchestrate *lifecycle operations*. On the contrary, declarative processing utilizes the *topology template* to derive necessary management tasks and their execution sequence based on *node* and *relationship templates* [10].

TOSCA-based cloud applications are packaged as a self-contained, portable service model in form of a *Cloud Service ARchive (CSAR)* that can be used to deploy and manage service instances in all TOSCA-compliant environments.

In the Simple Profile in YAML V1.0 [9], TOSCA provides modeling constructs for containers. The non-normative *node type Container.Application.Docker* represents a Docker application component. Furthermore, Docker images can be defined as non-normative *deployment artifact type* called *Deployment.Image.Container.Docker*. Whereas these modeling constructs may be used to integrate Docker containers into holistic topology models (including virtual machines), we aim at enabling container orchestration supporting the abstraction provided by container management systems.

3 Two-phase deployment method

Container-based artifacts allow the provisioning of application software and middleware. Therefore, these application components are composed and packaged by using a set of containers. Referring to the example application (cf. Figure 1), both the application software WordPress and the middleware, an Apache HTTP server hosting a PHP module, should be bundled into a single container image, since the middleware establishes a runtime for WordPress. Figure 3 shows this single node called *wordpress_container* on the left. The MySQL database is provided in a separate container node called *mysql_container*. Both container nodes are connected by a *ConnectsTo* relationship (cf. Figure 3). Note that the appropriate partitioning of application components into containers is highly application-specific and requires non-trivial architectural decisions.

Essentially, using container-based artifacts leads to smaller topologies, but treats containers as black boxes. Current approaches employ templates only to specify environment-centric deployment logic. In fact, parts of the deployment logic have been moved into containers leading to *two-phase deployment*.

In the *node-centric phase* application software and middleware components of a node are composed by defining a build specification. All node-related configurations are applied in this phase and result in a container-based artifact. Furthermore, software developers are able to reuse existing container-based artifacts as basis for their build specification. Thus, a build specification only captures additional changes that have to be applied to the underlying container-based artifact. This leads to reusability and thus to faster software release cycles. Nevertheless, knowledge on the components and deployment requirements of reused container-based artifacts is required. In the *environment-centric phase*, multiple container-based artifacts are employed to deploy a defined topology of containers using a container management system.

Referring to our example application, the coarse-grained topology used in the environment-centric phase consists of only two nodes and a single relationship (cf. Figure 3). On the other hand, a more fine-grained model is required for the node-centric phase, where application components of a node are assembled and configured to create a container. The internal structure of a container is specified as a set of layers each described by a corresponding build specification such as a Dockerfile. Different *views* are required to ensure the abstraction provided by container-based artifacts for the environment-centric phase, while also providing the internal structure of a container for the node-centric phase.

We aim at providing both views to ease the mapping of deployment requirements of internal components to environment-centric artifacts (e.g., templates). The general idea is to synchronize both views on the deployment logic of a software release to support automated deployment and the maintainability of deployment logic required to enable automation. We propose a TOSCA-based approach to integrate both deployment phases and their corresponding views. In this respect, we employ the *service template* as standards-based environment-centric artifact. TOSCA *lifecycle operations* provide a generic concept for mapping deployment requirements of internal components to the *service template*.

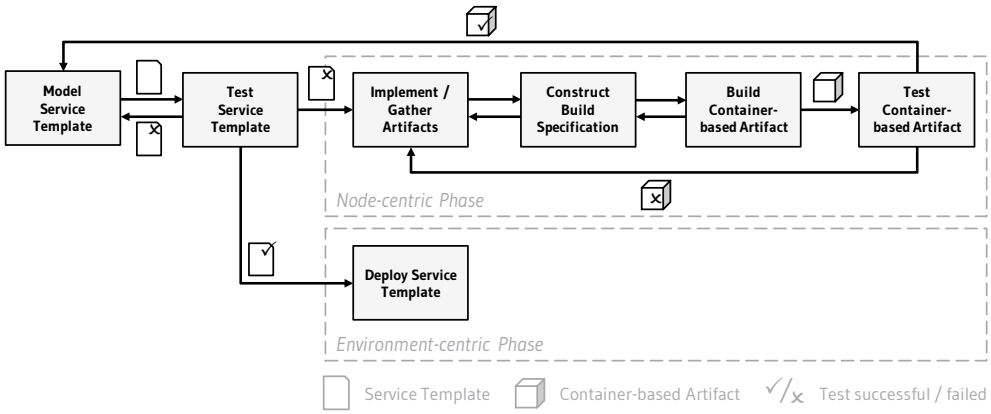

Fig. 4 Two-phase deployment method

Figure 4 outlines how both phases are integrated by means of a *service template*. Modeling and testing the *service template* are the central steps in our two-phase deployment method. The *service template* is the central artifact, which contains all required information with respect to deployment. In this context, testing the service template can be implemented depending on the requirements ranging from syntax testing to system integration testing. The node-centric phase encapsulates the construction of container-based artifacts. This phase is executed per container and requires manual effort such as gathering required application software and middleware artifacts and constructing the build specification. However, this phase is only required if a new container-based artifact has to be constructed or an existing one has to be updated. Iterating among the steps of the node-centric phase may be required if more then one layer should be used to construct the final container-based artifact. The environment-centric phase is executed in a fully automated manner by submitting a *service template* to a TOSCA-based container management system (cf. Section 5). We explain both phases, the corresponding concepts, and related modeling constructs in the next section.

4 TOSCA-based integration

In this section, we elaborate on integration concepts to support two-phase deployment. This section is structured according to the deployment requirements, which we identify in both deployment phases of our example application (cf. Figure 3).

4.1 Environment-centric phase

In the environment-centric phase a *service template* is processed by a container management system. Enabling automated orchestration of container-based cloud applications leads to several requirements with respect to a *service template*, which we discuss in the following based on the environment-centric deployment of our example application (cf. Figure 5). Further, we derive requirements for a TOSCA-based container management system, which is used to deploy the *service template*. These requirements are denoted as (RQx). In Section 5 we show how we designed our TOSCA-based container management system based on these requirements.

Fig. 5 Exemplary environment-centric deployment

A container management system deploys a specified topology based on a *service template* and manages virtual machines transparently (RQ1). As a result, virtual machines do not have to be specified as part of application topologies, while the container management system ensures optimized resource utilization by scheduling container-based applications across a set of virtual machines (RQ2).

The first step to specify the application topology depicted in Figure 3 is modeling containers as TOSCA nodes. We use Docker as container format and employ the *deployment artifact type Deployment.Image.Container.Docker* to define Docker images. Listing 1 shows our *node template* for the *mysql_container* based on the Simple Profile in YAML [9]. The Docker image *mysql/mysql-server* is specified as *artifact* called *my_image* (cf. Listing 1, line 12–16). This Docker image is stored in the Docker Hub[7], which is a public repository

[7] https://hub.docker.com.

```
1  mysql_container:
2    type: cst.nodes.Docker.MySQL
3    properties:
4      cpu_shares: 0.5
5      mem_size: 512 MB
6      disk_size: 500 MB
7    capabilities:
8      db_endpoint:
9        properties:
10         protocol: tcp
11         port: 3306
12   artifacts:
13     my_image:
14       file: mysql/mysql-server
15       type: tosca.artifacts.Deployment.Image.
           ↪ Container.Docker
16       repository: docker_hub
17   interfaces:
18     Standard:
19       create:
20         implementation: my_image
21         inputs:
22           MYSQL_ROOT_PASSWORD: my-root-pw
23           MYSQL_USER: my-user
24           MYSQL_PASSWORD: my-user-pw
25           MYSQL_DATABASE: my-db
```

Listing 1: mysql_container node template in YAML

for Docker images. We utilize repositories to store and retrieve *deployment artifacts* (RQ3).

Container management systems provide an abstraction layer for scheduling containers across virtual machines. Therefore, each *node template* defines the resources it requires, e.g., CPU shares, memory size, and disk size, while a container management system ensures that these resource requirements are met (RQ4). Listing 1 shows the specification of container resources, i.e., *cpu_shares*, *mem_size*, and *disk_size*, in form of *properties* (cf. Listing 1, line 3–6).

The *lifecycle operation create* is used to create a node. Therefore, the *create* operation of *mysql_container* specifies the *deployment artifact my_image* and the required *inputs* (cf. Listing 1, line 19–25). The container management system provides the runtime for the node including setting the specified *inputs* as environment variables and instantiates the node based on the given *deployment artifact* (RQ5). The *wordpress_container* described in Listing 2 is constructed in an analogous manner.

This far, we know how to create our nodes. Next, we have to establish a relationship. For this purpose, we rely on the generic TOSCA concept of *endpoints*. Referring to our example application, the *mysql_container* provides a *capability* named *db_endpoint* (cf. Listing 1, line 7–11), which is used by the *wordpress_container* to connect to the database. To this end, the *wordpress_container* specifies a *requirement db_endpoint*, which

```
1  wordpress_container:
2    type: cst.nodes.Docker.WordPress
3    ...
4    requirements:
5      - db_endpoint:
6          node: mysql_container
7          relationship: connect_to_db
8    artifacts:
9      wp_image:
10       file: wordpress-custom
11       type: tosca.artifacts.Deployment.Image.
           ↪ Container.Docker
12       repository: custom_repository
13   interfaces:
14     Standard:
15       create:
16         implementation: wp_image
```

Listing 2: wordpress_container node template in YAML

relates to *mysql_container* with a custom relationship *connect_to_db* (cf. Listing 2, line 5–7).

Additional configuration is required to set up this relationship, since the *wordpress_container* requires the IP address of the *mysql_container* to connect to the corresponding *endpoint*. Therefore, *implementation artifacts* are used in TOSCA.

We propose *embedded implementation artifacts* for container-based cloud applications. We define *embedded implementation artifacts* as *implementation artifacts*, which are provided by container-based *deployment artifacts* (i.e., container images). This leads to a single container-based artifact with several benefits: (1) additional distribution of *implementation artifacts*, e.g., by using remote access protocols such as SSH / SCP, is not necessary; (2) the container image specifies the execution environment to execute *implementation artifacts*; (3) We can use repositories to store and retrieve container-based artifacts. We identified two types of *embedded implementation artifacts*:

- *Node-triggered implementation artifacts* are triggered by the container itself during the creation of the corresponding node. They are used to configure a node directly after its instantiation. All input values are not dependent on runtime information or already have been resolved before node instantiation.
- *Environment-triggered implementation artifacts* are triggered by orchestration solutions and can be used to configure a node that has been created before. They are indispensable if required input values are dependent on runtime information that cannot be resolved before node creation.

We already used *node-triggered implementation artifacts* to create our nodes without explicitly mentioning them. When providing inputs to the *create* oper-

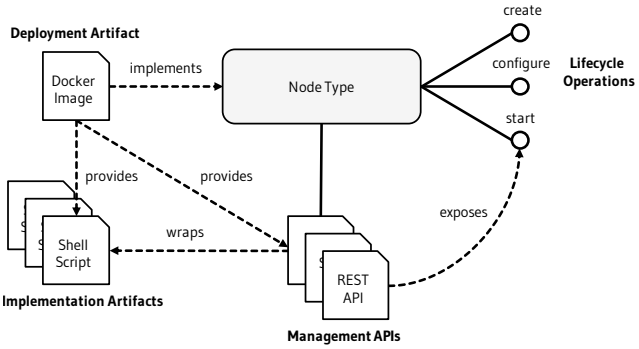

Fig. 6 Management API concept

ation, not only an instance of *mysql_container* is created, but also the configuration of login credentials is executed. On the other hand, *environment-triggered implementation artifacts* enable us to set up our custom relationship *connect_to_db*, which requires the configuration of an existing *wordpress_container* instance.

To access *environment-triggered implementation artifacts*, we employ the concept of *management APIs* [11, 12]. In this context, management APIs are standards-based interfaces provided by nodes. Figure 6 shows how we use these management APIs to wrap *environment-triggered implementation artifacts* provided by the *deployment artifact* of a *node type*.

Node and *relationship templates* specify the information necessary to invoke a management API (RQ6). Therefore, the relationship *connect_to_db* specifies its normative *pre_configure_source* operation, which refers to an *implementation artifact* named *api_artifact* and requires four input values (cf. Listing 3, line 5–11). This *implementation artifact* specifies all required information to access the management API (cf. Listing 4). In this case, the *implementation artifact* does not refer to a file. However, *file* is a required keyname [9].

TOSCA functions allow referencing values inside the *service template*, which have to be resolved during runtime [9]. For example, *connect_to_db* specifies the input *WORDPRESS_DB_HOST* using the TOSCA function *get_attribute* to get the *ip_address* of the *TARGET* of this relationship (cf. Listing 3, line 8). In this case, *mysql_container* is the *TARGET* and the TOSCA function will resolve to the corresponding IP address, which is not known before *mysql_container* has been created.

Our integration concepts influence the way how container management systems have to be built. Table 1 summarizes the derived requirements for a TOSCA-based container management system, which automates the environment-centric phase by using a *service template*.

```
1  connect_to_db:
2    type: ConnectsTo
3    interfaces:
4      Configure:
5        pre_configure_source:
6          implementation: api_artifact
7          inputs:
8            WORDPRESS_DB_HOST: { get_attribute:
                ↪  [TARGET, ip_address] }
9            WORDPRESS_DB_USER: my-user
10           WORDPRESS_DB_PASSWORD: my-user-pw
11           WORDPRESS_DB_NAME: my-db
```

Listing 3: ConnectsTo relationship template in YAML

```
1  api_artifact:
2    file: none
3    type: tosca.artifacts.Implementation.API.REST
4    properties:
5      protocol: HTTP
6      format: JSON
7      path: /api/configure
8      port: 8080
```

Listing 4: Management API artifact in YAML

Table 1 Requirements for a TOSCA-based container management system

ID	Requirement
RQ1	Manage virtual machines (hosts) transparently.
RQ2	Enable optimization of resource utilization by scheduling container-based applications across virtual machines.
RQ3	Fetch container-based artifacts from repositories.
RQ4	Acquire computing resources for nodes as described by their *node templates*.
RQ5	Provide container runtime environment for instantiating nodes based on container-based artifacts.
RQ6	Invoke *lifecycle operations* by calling management APIs specified in *node* and *relationship templates*.

4.2 Node-centric phase

The node-centric phase addresses the construction of container-based artifacts. Therefore, the internal structure of the corresponding containers has to be known. This structure is given by a set of layers hosted on top of each other. Further, each layer is described by a build specification. We employ the *service template* to model these layers. Based on the TOSCA standard it is not possible to model both the internal structure of a container and the container itself. Thus, additional modeling constructs are required. However, these modeling constructs are only used in the node-centric phase of two-phase deployment. In a preprocessing step, we au-

tomatically remove them before submitting the service template to a container management system.

The topology of our example application is comprised of two containers. The node-centric phase is executed for each container separately. In the following, we explain the node-centric phase of the *wordpress_container* (cf. Listing 2). At first, we gather or implement artifacts required for application and middleware components as well as *implementation artifacts* (cf. Figure 4, "implement / gather artifacts"). The node-centric deployment logic is specified in a build specification (cf. Figure 4, "construct build specification").

In case of the *wordpress_container*, an existing base container image might be used, which already contains an Apache HTTP server with a PHP module installed. On top of this layer, a new *wordpress* layer is constructed by creating a Dockerfile, which configures the base container image according to the requirements of WordPress, e.g., download WordPress and unzip files.

These two layers, namely *wordpress* and *apache_php*, represent the internal structure of *wordpress_container* and are modeled as nodes. Further, we define the internal structure by means of a new construct to model contained nodes (cf. Listing 5, line 3). Note that this is an exemplary internal structure. More contained nodes can be added thus capturing all layers down to the Docker scratch image. However, this depends on the documentation requirements in an individual setting.

```
1  wordpress_container:
2    type: cst.nodes.Docker.WordPress
3    contains: [wordpress, apache_php]
4    ...
```

Listing 5: Contained nodes in YAML

Furthermore, all deployment requirements of encapsulated components have to be mapped to the *node template* of *wordpress_container*. Whereas we already defined *wordpress_container* in Section 4.1 (cf. Listing 2), all *lifecycle operations* of *wordpress_container* have to be supported by its contained nodes or layers, respectively. In this case, the only *lifecycle operation* specified is *create* and requires no inputs.

Moreover, *environment-triggered implementation artifacts* have to be provided by containers. The relationship *connect_to_db* (cf. Listing 3) specifies the *lifecycle operation pre_configure_source*, which defines a management API for *wordpress_container*. This management API wraps a shell script, which is used for connecting to the MySQL database. In this case, we assume a lightweight Java-based REST API. All required files are added to the Dockerfile and configured accordingly.

The node-centric deployment logic of *wordpress_container* is captured in the set of all build specifications of the internal layers. It is especially required if changes to an existing container-based artifact are necessary, since internal deployment requirements have to be mapped to *lifecycle operations* of *wordpress_container*. To integrate both views, we append the build specification to contained nodes. Thus, the Dockerfile is specified by using a custom *artifact type cst.artifacts.Deployment.BuildSpec.Docker*, i.e., in the node-centric phase a build specification is considered as *deployment artifact* of a contained node. Besides, *image_name* and *repository* of the resulting container-based artifact are specified as *properties* (Listing 6, line 5–10).

```
1  wordpress:
2    type: cst.nodes.DockerInternal
3    ...
4    artifacts:
5      build_spec:
6        file: wp/Dockerfile
7        type: cst.artifacts.Deployment.BuildSpec.
           ↪ Docker
8        properties:
9          image_name: wordpress-custom
10         repository: custom_repository
11   interfaces:
12     Standard:
13       create:
14         implementation:
15           primary: build_spec
16           dependencies:
17             - wp/pre_configure.sh
18             - wp/managementAPI-rest.jar
```

Listing 6: WordPress build specification in YAML

More artifacts may be required for the build process, e.g., the shell script and its management API implementation. These artifacts can be specified as shown in Listing 6 (line 16–18). All files have to be provided in the *CSAR*.

The *apache_php* node is modeled in an analogous manner. Further, both contained nodes can be connected by a generic *HostedOn* relationship.

After building the container-based artifact, a test is applied to check whether a container behaves as expected (cf. Figure 4). In case of failure, we fix observed bugs and retry the test after building the container-based artifact again. If the test succeeds, we push it to the corresponding repository. After updating the *service template* (cf. Figure 4, "model service template"), we retry the *service template* test (cf. Figure 4, "test service template"). If it succeeds, the *service template* is prepared for automated deployment with a TOSCA-based container management system.

5 TOSCA-based container management system

In this section, we describe and evaluate our prototype of a TOSCA-based container management system. It automates the environment-centric phase and addresses the requirements derived (cf. Table 1).

We selected Mesos as abstraction layer for resource management and runtime provisioning due to its modular ecosystem [13], which is adaptable with respect to internal components and integration of third party code. For example, *Isolator* modules enable the implementation and integration of custom resource isolation mechanisms such as solutions for software-defined networking, external storage, or GPU hardware support. Mesos covers our requirements with respect to infrastructure abstraction, i.e., it manages virtual machines transparently (RQ1), enables optimization of resource utilization (RQ2), and provides an extensible container runtime environment (RQ5). Mesos is designed as master/worker architecture with a *Mesos Master*, which is replicated to ensure failure recovery, and multiple workers, which are called *Mesos Agents*. We make use of the *Mesos Frameworks* concept to implement a TOSCA-based orchestrator on top of Mesos.

The basic building blocks of a Mesos Framework are the *Scheduler* and the *Executor*. A Scheduler communicates with the Master and manages the scheduling of tasks. Therefore, the Master sends resource offers to Mesos Frameworks. While the Master decides how many resources to offer, a Mesos Framework decides which resources to accept and which tasks to run [8]. The Scheduler implements a predefined callback API to receive updates from the Master. An Executor is responsible for executing tasks and runs on each Agent. Tasks may be shell commands or Docker containers. We use Docker containers as unit of deployment and employ the default Executor for our prototype.

Our prototype is built on top of Mesos 1.3.0 and implemented in Java. Figure 7 shows the architecture of our TOSCA-based container management system. We describe the depicted components in the following.

TOSCA Processor: Our TOSCA Processor expects a *service template* as input. Thus, the *CSAR* created formerly by adding node-centric artifacts has to be stripped of all node-centric modeling constructs and related artifacts. We process the *service template* and remove all node-centric modeling constructs such as contained nodes and their relationships before submitting the *service template* to our container management system. The TOSCA Processor translates the given *service template* into an internal Java object.

Deployment Generator: We chose declarative processing of the *service template* due to its simplicity. Therefore, our orchestrator deploys *node templates* according to dependency chains described by their *relationships*. The Deployment Generator applies topological sorting based on Kahn's algorithm [14] to generate deployment plans. Therefore, we derive a graph of deployment operations and their dependencies among each other. Topological sorting is employed to sort these deployment operations according to their dependencies. Note that our prototype does not generate deployment plans optimized for parallel execution of tasks [10].

Instance Manager: The Instance Manager manages an instance model representing the currently deployed nodes and relationships. This model can be accessed by using well-defined interfaces. The Instance Manager ensures a globally consistent state of the managed instance model and tracks the current state, e.g., if a node is running.

Deployment Manager: The Deployment Manager uses an instance model provided by the Instance Manager to deploy a *topology template* incrementally [9]. The Deployment Manager deploys nodes according to the *properties* specified in the *node template* (RQ4) and forwards deployment requests to the Scheduler. Furthermore, required inputs are specified and also forwarded to the Scheduler (RQ5). Some inputs may have to be resolved dynamically during runtime, e.g., the IP address of a node. In this case, TOSCA functions can be used in the *service template*. The Deployment Manager accesses the instance model provided by the Instance Manager to evaluate TOSCA functions before executing the corresponding deployment operation. The generated deployment plan ensures that only deployment operations are processed that require evaluable TOSCA functions. Further, it uses the API Invoker to call management APIs if necessary.

API Invoker: TOSCA *lifecycle operations* are accessed by using management APIs exposed by nodes. As a result, our orchestrator is relieved of the burden of executing many different *implementation artifacts* [15]. The API Invoker provides an invoker implementation for every supported interface type (RQ6). We chose REST/HTTP for our prototypical implementation. However, more invokers can be added easily.

Artifact Repository: The Artifact Repository is used to retrieve container-based artifacts. We implemented an Artifact Repository in form of a private Docker registry[8]. More Artifact Repositories may be used or implemented easily (cf. Section 4.1).

Scheduler: The Scheduler connects the orchestrator and Mesos. The Deployment Manager is responsible for passing resource requirements to the Scheduler, which handles resource allocation by accepting or de-

[8] https://hub.docker.com/_/registry.

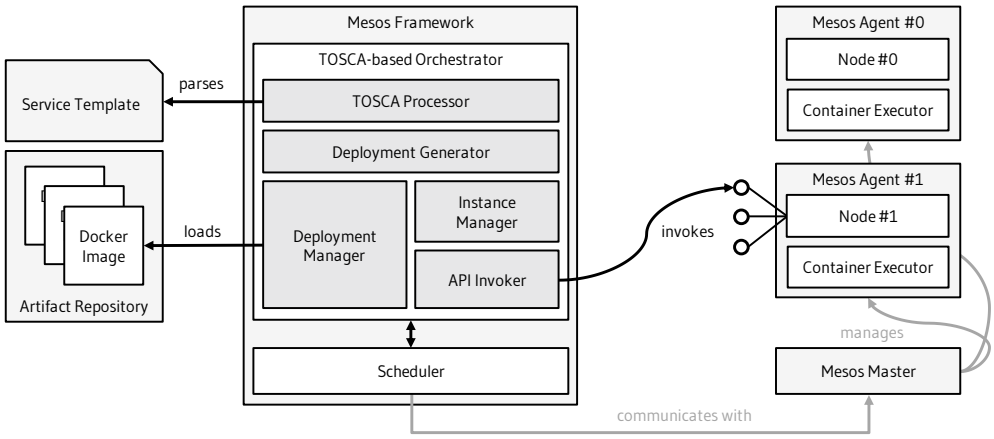

Fig. 7 Architecture of our TOSCA-based container management system

clining resource offers from the Master. Mesos manages container executors on Agents, which are responsible for creating the node instances (RQ5). To create a node instance, *deployment artifacts* are retrieved from an Artifact Repository (RQ3).

Evaluation. For evaluation purposes we executed 10 independent deployment runs by submitting the *service template* of our example application to the TOSCA-based container management system. We measured the total deployment time in seconds for every run. The total deployment time is defined as the elapsed time from the *service template* submission to the point, where all containers are instantiated, configured, and running. To avoid time variations related to network bandwidth and stability, we ran our TOSCA-based container management system as single node cluster. The sizes of the container-based artifacts required for deployment are 215 MB for *mysql/mysql-server* (cf. Listing 1) and 1077 MB for *wordpress-custom* (cf. Listing 2). Note that the size of *wordpress-custom* is not optimized for a real world scenario and is mainly related to the Java Virtual Machine (JVM), which is required for our Java-based management API. Using another technology for implementing the API or running an optimized JVM would improve the size significantly.

Our measurements result from a CentOS 7 virtual machine with 2 vCPUs clocked at 2.6 GHz, 4 GB RAM, and 40 GB disk running in our OpenStack-based cloud environment. Based on these measurements, we calculated an average total deployment time of 45 ± 5 seconds. However, the deployment time depends on a large number of factors such as the network bandwidth, the location of the Artifact Repository, the size of container-based artifacts, and host-based caching mechanisms. Thus, the reported values may be different in a real world scenario.

Table 2 DevOps artifacts supporting two-phase deployment

Deployment Phase	DevOps Artifacts
Node-centric Phase	– Build Specification – Node-triggered Implementation Artifact – Environment-triggered Implementation Artifact – Container-based Artifact
Environment-centric Phase	– Service Template – Container-based Artifact

6 Discussion

Our two-phase deployment method integrates two different views. Whereas the node-centric phase is concerned with the assembly and configuration of application components, the environment-centric phase aims at orchestrating application topologies in an automated manner. We integrate both phases of the deployment method by means of container-based artifacts and a *service template*. This approach enables capturing all deployment logic related to a software release within a single *CSAR* and thus leads to better maintainability. Further, the method supports the construction of deployment pipelines for container-based cloud applications, which enable fast feedback as well as automated deployment in the sense of continuous delivery [2, 5]. Composite cloud applications can be constructed as a set of self-contained and portable components with their dependencies and lifecycle operations in mind. Container virtualization prevents causes of failure, which originate from heterogeneous test and production environments. Table 2 summarizes the artifacts, which we employed for two-phase deployment.

Environment-triggered implementation artifacts also support use cases beyond flexible deployment: If input

values change during runtime, reinjection of dependencies is required, e.g., the *add_target* operation of a load balancer is invoked whenever a new target is added and the *remove_target* operation is invoked whenever a target is removed. Moreover, container-based cloud applications can support all normative *lifecycle operations* of TOSCA nodes and relationships by applying our integration concepts: *Create* and *delete* operations are supported by container management systems by default, *node-triggered implementation artifacts* ease the initial configuration of nodes, and *environment-triggered implementation artifacts* support custom management operations such as *start* and *stop* for nodes or *add_target* and *remove_target* for relationships.

One might argue that our approach is technology-dependent because we rely on Docker. However, the presented integration concepts are not limited to Docker-specific functionality. We assume that in the future standards for container virtualization will be supported by container management systems. The Open Container Initiative (OCI)[9] is an open governance structure with the purpose of creating open industry standards around container formats and a runtime, which can be implemented for the Mesos ecosystem.

The presented prototype constitutes the basis for portable cloud applications using container-based artifacts. TOSCA-based orchestration provides a uniform interface for container management systems. However, both more advanced functionality and application management beyond deployment have to be investigated in future research. Especially elastic workloads may benefit from the rapid provisioning of containers [4, 16].

7 Related work

Current container management systems do not support TOSCA-based orchestration and the integration of node-centric and environment-centric deployment logic is limited. Existing work identified the need for new methodological approaches and tool support to manage continuous development and deployment of container-based cloud applications [4, 5].

Each container management system provides its own custom solutions or has to be enhanced with additional components to consider deployment requirements stemming from internal components of a container [13]. On the contrary, TOSCA allows the standardized definition of custom *lifecycle operations* and thus provides flexibility. Combined with the concept of management APIs [11, 12], we allow an orchestrator to use *environment-triggered implementation artifacts*, which support all kind of *lifecycle operations*. However, by using *environment-triggered implementation artifacts* the orchestrator is responsible for lifecycle management of containers. Whereas this centralized approach supports complex orchestration scenarios, current container management systems typically employ custom solutions for distributed configuration of containers [13, 16]. It would be interesting if both approaches can be integrated, thus enabling distributed configuration where it can be applied and configuration managed by an orchestrator where it is required.

Existing work on the integration of TOSCA and DevOps artifacts is not focused on container-based artifacts as unit of deployment, but considers more generic approaches to implement deployment automation [3, 12, 15]. Moreover, it does not support modeling node-centric deployment logic in form of build specifications.

The use of APIs to wrap DevOps artifacts has been proposed in [3]. The presented framework supports portable orchestration using DevOps artifacts. The authors do not focus on container orchestration and use container-based artifacts to wrap *implementation artifacts*, whereas we use container-based artifacts as primary unit for both *deployment artifacts* and *embedded implementation artifacts*. However, we consider the investigation of automated API generation [3, 12] in conjunction with our approach as beneficial.

OpenTOSCA[10] is an open source runtime environment and implements the TOSCA standard. We proposed TOSCA-based orchestration as standardized interface for container management systems and services. As a result, our TOSCA-based orchestration layer is integrated with the resource provisioning, runtime, and isolation mechanisms of the corresponding container management system. This leads to both simple deployment of container-based cloud applications and optimized resource utilization. Whereas OpenTOSCA also allows automated deployment of container-based cloud applications, our prototype additionally invokes *management APIs* provided by container nodes.

Alien4Cloud[11] allows modeling TOSCA-based application topologies using container-based artifacts, but does not provide comparable concepts for integrating node-centric and environment-centric deployment logic.

8 Conclusion

In this paper, we address two major limitations of current container management systems and their application for deployment automation: Lock-in problems

[9] https://www.opencontainers.org.

[10] http://www.opentosca.org.

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

and limited knowledge on the internal structure of containers leading to maintainability and configuration issues. We applied a TOSCA-based approach to enable the portable specification of application topologies using container-based artifacts. Based on well-established concepts, we elaborated on the integration of node-centric and environment-centric deployment. Our two-phase deployment method fosters creating and maintaining all the deployment logic related to a software release. For automation purposes, we designed and implemented a prototypical TOSCA-based container management system. It contributes to the understanding of opportunities and challenges related to the usage of TOSCA for container orchestration.

Whereas automation is an important aspect of application deployment, in this context, some tasks cannot be fully automated. Future work may result in tools to facilitate manual work during the node-centric phase. In particular, the partitioning of application topologies to construct multi-node cloud applications using containers involves complex architectural decisions, which are supported by specific patterns [11]. We plan to investigate whether a higher degree of automation may be achieved for specific application classes.

Acknowledgment

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

References

1. Humble J, Molesky J (2011) Why enterprises must adopt devops to enable continuous delivery. Cutter IT Journal 24(8):6
2. Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley
3. Wettinger J, Breitenbücher U, Kopp O, Leymann F (2016) Streamlining devops automation for cloud applications using tosca as standardized metamodel. Future Generation Computer Systems 56(C):317–332
4. Kratzke N, Quint PC (2017) Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. Journal of Systems and Software 126:1–16
5. Pahl C, Brogi A, Soldani J, Jamshidi P (2017) Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing PP(99):1–1
6. OASIS (2013) Topology and orchestration specification for cloud applications (tosca) version 1.0, committee specification 01. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html
7. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J (2016) Borg, omega, and kubernetes. Queue 14(1):10:70–10:93
8. Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI), USENIX, pp 295–308
9. OASIS (2016) Tosca simple profile in yaml version 1.0, committee specification 01. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/cs01/TOSCA-Simple-Profile-YAML-v1.0-cs01.html
10. Breitenbücher U, Binz T, Képes K, Kopp O, Leymann F, Wettinger J (2014) Combining declarative and imperative cloud application provisioning based on tosca. In: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), IEEE Computer Society, pp 87–96
11. Burns B, Oppenheimer D (2016) Design patterns for container-based distributed systems. In: Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud), USENIX, pp 108–113
12. Wettinger J, Breitenbücher U, Leymann F (2016) Streamlining apification by generating apis for diverse executables using any2api. In: Cloud Computing and Services Science, Springer International Publishing, pp 216–238
13. Peinl R, Holzschuher F, Pfitzer F (2016) Docker cluster management for the cloud - survey results and own solution. Journal of Grid Computing 14(2):265–282
14. Kahn AB (1962) Topological sorting of large networks. Commun ACM 5(11):558–562
15. Wettinger J, Binz T, Breitenbücher U, Kopp O, Leymann F, Zimmermann M (2014) Unified invocation of scripts and services for provisioning, deployment, and management of cloud applications based on tosca. In: Proceedings of the 4th International Conference on Cloud Computing and Service Science (CLOSER), SciTePress, pp 559–568
16. Kang H, Le M, Tao S (2016) Container and microservice driven design for cloud infrastructure devops. In: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), IEEE Computer Society, pp 202–211