Future Generation Computer Systems 56 (2016) 317–332

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel

Johannes Wettinger*, Uwe Breitenbücher, Oliver Kopp, Frank Leymann

Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany

HIGHLIGHTS

- Classification of DevOps artifacts and their usage.
- Integrated, standards-driven modeling & runtime framework based on TOSCA.
- Discovery, transformation, and APIfication of DevOps artifacts.
- Enrichment and deployment of application topologies.
- Evaluation of artifact transformation and comprehensive case study.

ARTICLE INFO

Article history:
Received 1 March 2015
Received in revised form
17 June 2015
Accepted 30 July 2015
Available online 10 August 2015

Keywords:
DevOps
Deployment automation
Transformation
TOSCA
Cloud standards
Cloud computing

ABSTRACT

DevOps as an emerging paradigm aims to tightly integrate developers with operations personnel. This enables fast and frequent releases in the sense of continuously delivering new iterations of a particular application. Users and customers of today's Web applications and mobile apps running in the Cloud expect fast feedback to problems and feature requests. Thus, it is a critical competitive advantage to be able to respond quickly. Besides cultural and organizational changes that are necessary to apply DevOps in practice, tooling is required to implement end-to-end automation of deployment processes. Automation is the key to efficient collaboration and tight integration between development and operations. The DevOps community is constantly pushing new approaches, tools, and open-source artifacts to implement such automated processes. However, as all these proprietary and heterogeneous DevOps automation approaches differ from each other, it is hard to integrate and combine them to deploy applications in the Cloud using an automated deployment process. In this paper we present a systematic classification of DevOps artifacts and show how different kinds of artifacts can be discovered and transformed toward TOSCA, which is an emerging standard. We present an integrated modeling and runtime framework to enable the seamless and interoperable integration of different approaches to model and deploy application topologies. The framework is implemented by an open-source, end-to-end toolchain. Moreover, we validate and evaluate the presented approach to show its practical feasibility based on a detailed case study, in particular considering the performance of the transformation toward TOSCA.

1. Introduction

The split between developers and operations, which is traditionally found in many organizations today is a major obstacle for fast and frequent releases of applications. Different goals, contrary mindsets, and incompatible processes are followed by these two groups. For instance, developers aim to push changes into production quickly, whereas the operations personnel's main goal is to keep production environments stable [1]. For this reason, collaboration and communication between developers and operations personnel are mainly based on slow, manual, and error-prone processes. As a result, it takes a significant amount of time to make changes, new features, and bug fixes available in production environments. However, especially users and customers of Web applications and mobile apps expect fast responses to their changing and growing requirements. Thus, it becomes a competitive advantage to implement automated processes that enable quick and

* Corresponding author.
E-mail addresses: wettinger@iaas.uni-stuttgart.de (J. Wettinger), breitenbuecher@iaas.uni-stuttgart.de (U. Breitenbücher), kopp@iaas.uni-stuttgart.de (O. Kopp), leymann@iaas.uni-stuttgart.de (F. Leymann).

http://dx.doi.org/10.1016/j.future.2015.07.017
0167-739X/

frequent releases. This can only be achieved by closing the gap between development and operations. *DevOps* [2] is an emerging paradigm to bridge this gap between these two groups, thereby enabling efficient collaboration. Besides organizational and cultural challenges to eliminate the split, the *deployment process* needs to be highly automated to enable continuous delivery of software [3]. The constantly growing DevOps community supports this by providing a huge variety of individual approaches such as tools and artifacts to implement holistic deployment automation. Reusable DevOps artifacts such as scripts, modules, and templates are publicly available to be used for deployment automation. Juju charms[1] as well as Chef cookbooks[2] are examples for these [4,5]. In addition, Cloud computing [6] is used to provision the underlying resources such as virtual servers, storage, network, and databases. DevOps tools and artifacts can then configure and manage these resources. Thus, end-to-end deployment automation is efficiently enabled by using the DevOps approaches in Cloud environments.

However, DevOps artifacts are usually bound to certain tools. For instance, Chef cookbooks require a Chef runtime, whereas Juju charms need a Juju environment to run. This makes it challenging to reuse different kinds of heterogeneous artifacts in combination with others. Especially when systems have to be deployed that consist of various types of components, typically multiple approaches have to be combined because they usually focus on different kinds of middleware and application components. Thus, there is a variety of solutions and orchestrating the best of them requires to integrate the corresponding tools, e.g., by writing workflows or scripts that handle the individual invocations. However, this is a difficult, costly, and error-prone task as there is no means to do this integration in a standardized manner, supporting interoperability of the artifacts involved. Therefore, the goal of our work is to enable the seamless integration of different kinds of DevOps artifacts based on the emerging OASIS standard *TOSCA* (Topology and Orchestration Specification for Cloud Applications) [7]. In this paper we present the major contributions of our work:

- We propose an initial classification of DevOps artifacts and outline how they are used.
- We present an integrated and comprehensive modeling and runtime framework based on TOSCA, covering the discovery, transformation, and APIfication of artifacts as well as the enrichment and deployment of application topologies.
- We discuss concrete artifact transformation processes for Chef cookbooks and Juju charms.
- We evaluate these transformation processes by measuring and analyzing the transformation performance.
- We conduct a comprehensive case study based on the motivating scenario to validate our framework.

Our work presented in this paper is based on previous research that has been published in the paper entitled *Standards-based DevOps Automation and Integration Using TOSCA* [8] at the 7th International Conference on Utility and Cloud Computing (UCC 2014). For reasons of better comprehension we repeat what we did previously to present the additional work and research that has been done on top. The remaining part of this paper is structured as follows: Section 2 describes the problem statement and presents a motivating scenario that is used as a running example. The fundamentals to understand our work are shown in Section 3, including a classification of DevOps artifacts and the explanation of Chef, Juju, and TOSCA. The core of our work, namely the TOSCA-based integrated modeling and runtime framework is discussed in Section 4. Based on the motivating scenario, Section 5 presents an evaluation and validation of our framework and its implementation. Sections 6 and 7 present related work and conclude the paper.

2. Problem statement and motivating scenario

The DevOps community actively shares open-source artifacts (i.e., *DevOps artifacts*) such as scripts, modules, and templates to deploy middleware and application components. Their portability and community support make them a predestined means to be reused to automate the deployment of different kinds of applications, especially Web applications and Web-based backends for mobile apps. As long as only a single type of artifacts is used, the artifacts are typically interoperable and a corresponding tooling may be utilized, e.g., a Chef runtime [9] for Chef cookbooks. However, using and combining artifacts of different kinds such as Chef cookbooks and Juju charms in a seamless manner is a major challenge. Extra effort is required to learn and integrate all their peculiarities such as invocation mechanisms, state models, parameter passing, etc. In addition, the orchestration of different artifacts and tools must be implemented using workflows or scripts that integrate them on a very low-level of abstraction. This requires deep technical insight in the corresponding technologies and the overall orchestration approach. Typically, a lot of glue code is required that makes the overall orchestration hard to understand and maintain for non-experts. In the following we discuss a motivating scenario as a concrete example to confirm the necessity of integrating different kinds of artifacts.

Fig. 1 shows a part of the topological structure of the Web shop application inspired from [10]. The application itself is hosted on an Apache HTTP server and depends on the PHP module; the database of the application is hosted on a MySQL master/slave environment to improve the application's scalability and to enable high availability of the database: data that are written to the master instance are consistently replicated to the slave instances, so reading requests can be load-balanced between slave instances. In case the master instance breaks, a slave instance can be selected to be the new master instance. The underlying infrastructures and/or platforms could be chosen depending on certain requirements or preferences. For instance, the Apache and MySQL servers could be hosted on virtual machines provided by Amazon Web Services.[3] To implement deployment automation for this application, we aim to reuse existing DevOps artifacts, especially to deploy the middleware components. For instance, Chef cookbooks may be used to deploy the Apache HTTP server and the PHP module, assuming that these are running on a single virtual machine. However, there is no Chef cookbook to deploy a complete MySQL master/slave environment out of the box. Moreover, the required scaling mechanisms to manage such an environment are missing. Consequently, using the MySQL charm[4] shared by the Juju community to deploy and dynamically scale such a MySQL setup is a better option to meet all relevant requirements. This adds another kind of artifact to the deployment automation implementation, implying the learning, usage, and orchestration of additional tooling to handle and execute corresponding artifacts. Finally, we may have implemented custom Unix shell scripts to meet specific deployment and operations requirements of our Web shop application and its database. These scripts are used to deploy the application-specific parts of the topology. Consequently, there is yet another kind of artifact involved in the

[1] Juju Charm Store: http://jujucharms.com.
[2] Chef Supermarket: https://supermarket.getchef.com.
[3] Amazon Web Services: http://aws.amazon.com.
[4] MySQL charm: http://jujucharms.com/mysql.

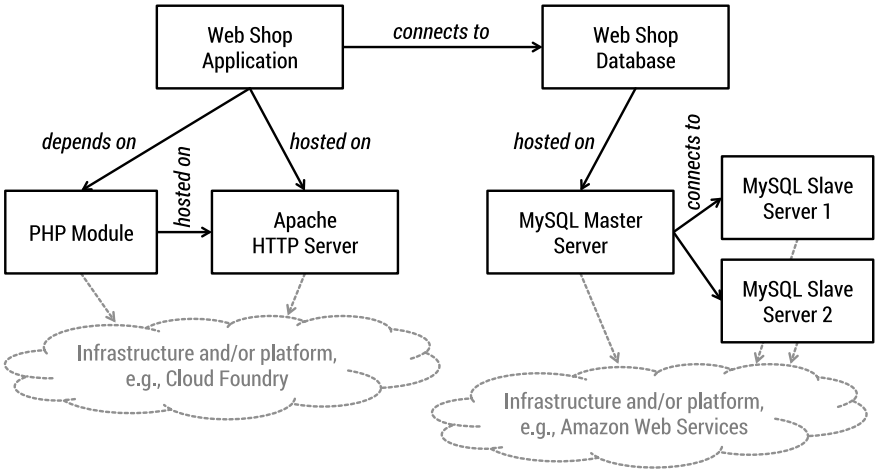

Fig. 1. Web shop application and database topology.

deployment automation process that needs to be integrated, too. Thus, three different kinds of deployment technologies must be combined to deploy the application efficiently in a fully automated manner by reusing existing artifacts that are optimally suited. This happens already for a relatively simple application topology as the one outlined in Fig. 1. Consequently, a monolithic approach by just using one single deployment solution in a one-size-fits-all manner is not suitable for many scenarios.

Cloud standards such as TOSCA[5] [7] tackle this challenge of seamlessly integrating and combining different kinds of artifacts by introducing a uniform metamodel. TOSCA artifacts can be used and composed seamlessly to create application models that can be deployed automatically. TOSCA is an emerging standard, but it still lacks an ecosystem of communities and reusable artifacts. However, an ecosystem based on open-source communities is key to establish TOSCA in practice [11,12]. Because the DevOps community provides such an ecosystem but is mostly based on individual and proprietary approaches, the goal of our work is to *discover and transform existing DevOps artifacts toward TOSCA to make them reusable and composable in a seamless and interoperable manner. This is to support the enrichment of TOSCA-based application topologies with executable DevOps artifacts to enable their automated deployment for maintaining instances of the application*. To provide the required background information for understanding our approach, the following Section 3 provides a classification of DevOps artifacts, outlines Chef and Juju as two concrete DevOps tooling approaches, and explains the basic constructs of TOSCA.

3. Fundamentals

In this section we discuss the fundamentals on which our work is based. Section 3.1 provides an initial classification of DevOps artifacts to clarify their conceptual and technical differences. We consider a representative of each class in our further discussions. TOSCA's most important concepts and constructs are presented in Section 3.4 as a foundation to discuss the core of our work in Section 4, namely the integrated modeling and runtime framework.

3.1. Classification of DevOps artifacts

As discussed in Section 2, there is a huge and ever-growing amount and variety of tools and artifacts shared by the DevOps community to deploy middleware and application components. Because these artifacts differ in how they are designed and how they are used, a classification enables their systematic categorization as, for instance, proposed in previous research [13]. Fig. 2 presents further classification aspects for DevOps artifacts, which are relevant for the work presented in this paper, based on two major classes:

1. *Node-centric artifacts (NCAs)* are scripts, virtual machine images, modules, declarative configuration definitions, etc., that are *executed on a single node* such as a physical server, a virtual machine (VM), or a container [14,15]. Cross-node relations such as an application component connecting to a database running on another node are not explicitly expressed and implemented. Consequently, NCAs are not meant to be used to deploy complete application topologies but only to install and configure single nodes.
2. *Environment-centric artifacts (ECAs)* are scripts, bundles, templates, etc., that are *executed in an environment, potentially consisting of multiple nodes*. Cross-node relations are explicitly expressed and implemented. Consequently, ECAs can be used to deploy complete application topologies.

Unix shell scripts, Chef cookbooks, Puppet modules,[6] SaltStack modules,[7] and Docker images[8] are a few prominent examples for NCAs. In terms of ECAs, Juju charms and bundles, Amazon CloudFormation templates,[9] and OpenStack Heat templates[10] are representatives of this class of artifacts. However, node-centric and environment-centric artifacts are not meant to be used exclusively. In fact, ECAs typically use and orchestrate NCAs. For instance, Juju charms utilize Unix shell scripts to install, configure, and wire software components on VMs. Moreover, Amazon CloudFormation templates[11] can utilize Chef cookbooks to implement deployment logic. Besides the orchestration of NCAs using ECAs, additional management tooling can be utilized to enable the environment-centric usage of NCAs. As an example, Marionette Collective[12] can manage the distribution and execution of Chef cookbooks in large-scale environments, providing consistent, environment-specific

[5] Topology and Orchestration Specification for Cloud Applications (TOSCA).

[6] Puppet Forge: https://forge.puppetlabs.com.

[7] SaltStack modules: http://goo.gl/l1mcnr.

[8] Docker Hub Registry: http://index.docker.io.

[9] Amazon CloudFormation templates: http://goo.gl/NzOe3.

[10] OpenStack Heat: http://wiki.openstack.org/wiki/Heat.

[11] Integration of Amazon CloudFormation with Chef: http://goo.gl/pOsU0.

[12] Marionette Collective: http://docs.puppetlabs.com/mcollective.

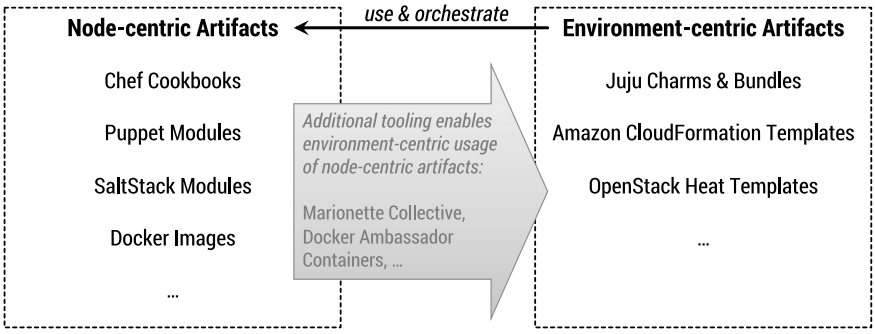

Fig. 2. Initial classification of DevOps artifacts.

context information to the nodes and NCAs involved. Docker ambassador containers[13] are another means to distribute and host containers based on Docker images [16] in a node environment, e.g., consisting of several VMs.

The presented classification based on NCAs and ECAs is the foundation for our transformation approach. There are several other aspects that may be considered when choosing corresponding artifacts to implement deployment automation. These aspects include:

- *Level of dependencies*: some artifacts are *provider-dependent* such as CloudFormation templates, i.e., they can only be used in combination with a certain provider such as Amazon in this case. Other artifacts are *tooling-dependent* such as Chef cookbooks or Juju charms: they can be used in conjunction with different providers, but require certain tooling such as a Chef runtime or a Juju runtime.
- *Level of virtualization*: artifacts may depend on certain virtualization solutions such as *hypervisor-based virtualization* (e.g., Amazon machine images) or *container virtualization* (e.g., Docker images) [15].
- Especially environment-centric artifacts can be distinguished in *infrastructure-centric* vs. *application-centric* artifacts. Infrastructure-centric artifacts such as CloudFormation templates focus on the configuration and orchestration of infrastructure resources such as VMs, storage, and network. Juju bundles are much more application-centric by focusing on the configuration and orchestration of middleware and application components and transparently managing the underlying infrastructure.
- There are *definition-based* artifacts such as Chef cookbooks and Puppet modules, defining the configuration of resources such as VMs or containers. On the other hand *image-based* artifacts such as Docker images capture the state of a certain resource to create new instances by restoring the persisted state on demand.
- Definition-based artifacts can be created in a *declarative*, in an *imperative*, or in a combined manner. For instance, Chef cookbooks typically define the desired state of a resource using a declarative domain-specific language [17,4]. However, imperative statements can also be part of such artifacts. Unix shell scripts typically consist of a set of imperative command statements.

In the following we explain Chef and Juju in more detail to highlight how NCAs (Chef cookbooks) and ECAs (Juju charms) are used in practice. Chef and Juju are representatives for their corresponding class. Moreover, we refine the positioning of these two approaches regarding the classification aspects discussed in this section. This is the foundation for later discussions of concrete transformation processes toward TOSCA based on Chef and Juju (Sections 4.2.1 and 4.2.2).

3.2. Chef

Chef [4,5] is a configuration management framework that provides a domain-specific language *(Chef DSL)* based on Ruby. The Chef DSL is used to define configurations of resources such as VMs. These configuration definitions are called *recipes*. Multiple recipes are bundled in *cookbooks*. For instance, a *MySQL* cookbook may provide the recipes *install_server*, *install_client*, and *install_all* to deploy the corresponding components of MySQL. The definition of declarative expressions such as *ensure that MySQL server is installed* is the recommended way to define portable configurations in recipes. However, imperative expressions such as system command statements (e.g., "`apt-get install mysql-server`") can be used, too. Fig. 3 provides an overview of the core concepts of Chef: the Chef server acts as a central management instance for the recipes, run lists, and attributes. Each node has a run list and a set of attributes assigned. The run list of a particular node specifies which recipes have to be executed on this node. Because most recipes are created to be used in different ways, they have some variability points such as the version number of the *mysql* package as shown in the sample recipe in Fig. 3. To resolve these variability points at runtime, attribute definitions such as *mysql_ver* = '5.5' can be assigned to a node. These attributes can be read during execution time. Finally, a Chef workstation runs *knife*[14] to control the Chef server as well as all nodes and data that are registered with the Chef server. To sum it up: according to our classification aspects presented in Section 3.1, Chef is a *tooling-dependent* but not provider-dependent solution that supports *different levels of virtualization*: Chef recipes can be executed on physical servers, VMs, and containers. Chef is *infrastructure-centric* because it focuses on the distribution and execution of recipes on infrastructure resources such as VMs. Because Chef recipes are typically *declarative* scripts, although they may include *imperative* statements as well, Chef follows the idea of creating and maintaining *definition-based* artifacts. Chef recipes are *node-centric artifacts* because they run in the scope of a single node.

3.3. Juju

In contrast to Chef, Juju[15] follows an environment-centric approach to implement configuration management and deployment automation. Charms contain scripts to implement a well-defined lifecycle of a certain component such as a MySQL server. For instance, scripts to *install*, *start*, and *stop* a MySQL server are contained in the MySQL charm. The lifecycle scripts can be implemented in an arbitrary language (Python, Ruby, Unix shell, etc.),

[13] Docker ambassador containers: http://goo.gl/PzGfcH.

[14] Chef knife: http://docs.opscode.com/knife.html.

[15] Ubuntu Juju: https://juju.ubuntu.com.

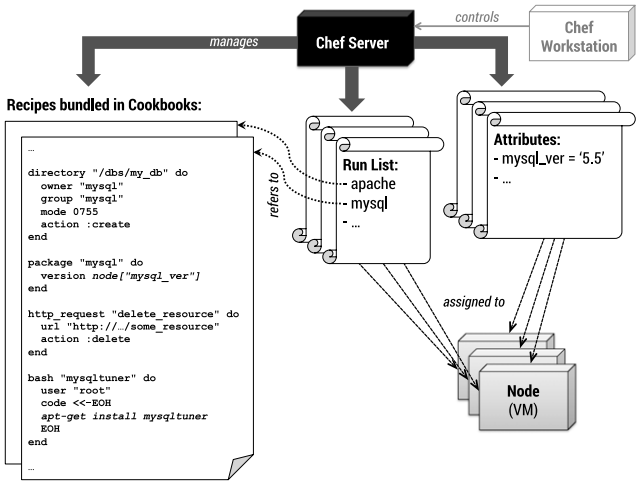

Fig. 3. Chef overview: artifacts, components, and interrelations.

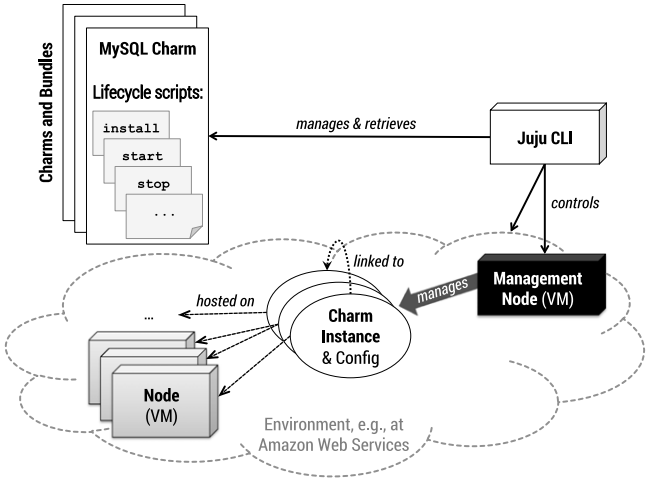

Fig. 4. Juju overview: artifacts, components, and interrelations.

as long as the resulting scripts are executable on the target nodes. Juju does not prescribe the usage of any domain-specific language to create these scripts. Fig. 4 denotes the environment-centric focus of Juju: charm instances and their configurations live in a certain environment such as an interconnected set of VMs hosted at Amazon Web Services. Charm instances can be linked such as an application component that is connected to a database. These links are explicitly expressed and can be configured. In contrast to Chef, charms and charm instances are the main entities; the underlying nodes and the scripts that are executed on them provide the required infrastructure to host the charm instances. It is important to denote that a charm instance is not limited to the scope of a single node. For instance, to deploy a multi-node MySQL database server in a master/slave setup, a single charm instance can be used that is scaled in and out by adding and removing *units*. An example is shown in Fig. 5, where the MySQL charm instance is hosted on three VMs. By default, one unit is one VM. When a new charm instance is created, it initially consists of one unit.

Besides the VMs that host units for the charm instances, a management node runs in each environment to deploy, manage, and monitor the charm instances. Conceptually, the management node can be compared to a Chef server. However, each environment requires its own management node, so different environments are independent of each other. Interconnections between management nodes are currently not supported, so multi-cloud applications are difficult to deploy and manage by Juju. A Juju

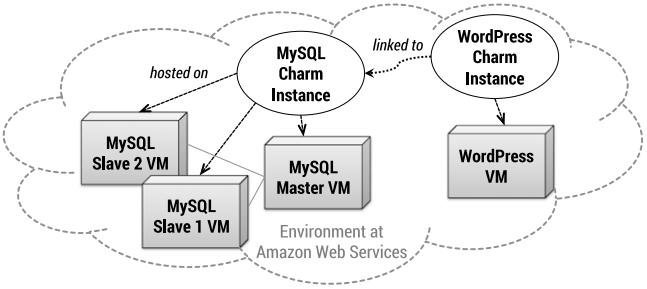

Fig. 5. Juju sample: MySQL and WordPress running at Amazon.

command-line interface (CLI) is provided to control both the management node and the environment itself, e.g., to configure security aspects such as firewalls. Different environments can be managed based on different infrastructures (OpenStack [18], Amazon Web Services, Microsoft Azure, HP Cloud, etc.). Moreover, the Juju CLI tool is used to manage and retrieve charms from public and private charm stores. This is the foundation for creating charm instances in any environment. Based on our classification aspects outlined in Section 3.1, Juju is a *tooling-dependent* solution, mostly focused on *hypervisor-based virtualization* because VMs are the unit of currency. Contrary to Chef, Juju can be considered as *application-centric* because the underlying infrastructure and scripts are abstracted by charms and charm instances. Furthermore, charms are *environment-centric artifacts* because links between them are explicitly expressed and can be configured. Charms are *definition-based* because of their lifecycle scripts that are typically implemented using *imperative* scripting languages. However, links between charms are expressed in a *declarative* manner by defining requirements that can be met by certain capabilities provided by other charms.

In terms of integration, Chef recipes could be reused as NCAs to implement the lifecycle scripts of Juju charms, which are ECAs. However, the integration requires deep technical knowledge of both approaches (state models, parameter passing, invocation mechanisms, etc.) and couples them tightly. If even further approaches and technologies come into play, all these have to be integrated separately. Consequently, a common, standardized intermediate metamodel is required to efficiently and seamlessly integrate different approaches and technologies. In the following Section 3.4 we introduce the Topology and Orchestration Specification for Cloud Applications (TOSCA) [7] that fulfills these metamodel requirements.

3.4. TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [7] is an emerging standard, supported by several companies in the industry.[16] Its main goal is to enhance the portability and management of Cloud applications. Technically, TOSCA is specified using an XML schema definition. *Topology templates* are defined as graphs consisting of nodes and relationships to specify the topological structure of an application as, for instance, shown in Section 2 (Fig. 1). As a foundation for defining such templates, *node types* and *relationship types* are defined as shown in Fig. 6. These are used to create corresponding *node templates* and *relationship templates* based on them in the topology template. A comprehensive type system can be introduced because types may be derived from other existing types in the sense of inheritance as it is used, for instance, in object-oriented programming. As an example, an abstract *Java Servlet Container* node type can be defined, which has a

16 TOSCA technical committee: https://www.oasis-open.org/committees/tosca.

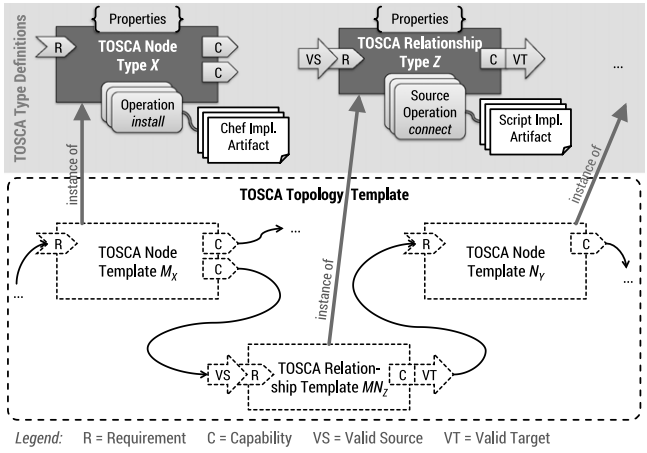

Fig. 6. Type definitions and templates in TOSCA.

child node type *Apache Tomcat*. There could be further node types derived from this one, such as *Apache Tomcat 6.0, Apache Tomcat 7.0*, etc.

Types consist of further sub-elements: *operations* are attached to nodes and relationships, for instance, to cover their lifecycle (*install, start, stop*, etc.). In addition, further management operations may be defined such as *backup_database* and *restore_database*. These operations are implemented by *implementation artifacts (IAs)*, which could be, for instance, Chef recipes or Unix shell scripts. An IA is executed when the corresponding operation is invoked, e.g., by the TOSCA runtime environment. Operations belonging to relationships are distinguished in *source operations* and *target operations* because a relationship links a source node with a target node. Source operations are executed on the source node, target operations on the target node. To enable the linking of nodes and relationships, they expose *requirements* and *capabilities* that are used for matchmaking purposes. For instance, a Java application node may expose a *Java Servlet Runtime* requirement, whereas the *Apache Tomcat* node provides a matching *Java Servlet Runtime* capability. A relationship specifying the *Java Servlet Runtime* requirement as valid source and the *Java Servlet Runtime* capability as valid target can be used to wire the two nodes. *Properties* can be defined as arbitrary data structures in XML schema to make nodes and relationships configurable. As an example, the *Apache Tomcat* node may provide a property to specify the directory where the log files should be written to. All properties are exposed to the operations and their IAs, so they can be considered during execution. Finally, TOSCA specifies the structure of *Cloud Service Archives (CSAR)* as a portable, self-contained packaging format for Cloud applications. Not only the XML definitions of types and templates are part of a CSAR; it also contains all scripts and files that are referenced, e.g., as IAs. Consequently, a CSAR is self-contained, enabling a TOSCA runtime environment to process it by traversing the topology template to create application instances. Each node provides lifecycle operations such as *install* that are called when the topology template gets traversed to create instances of the nodes and relationships on a real infrastructure. TOSCA supports the deployment and management of applications modeled as topology templates by two different flavors: (i) imperative processing and (ii) declarative processing [19]. The *imperative approach* employs so called *management plans* that orchestrate the management operations provided by node and relationship types. Thus, they execute the implementation artifacts that are attached to the corresponding operation. These plans can be executed automatically and are typically implemented using workflow languages such as *BPEL* [20], *BPMN* [21], or the BPMN extension *BPMN4TOSCA* [22]. To enable completely self-contained CSARs, management plans can be stored directly in the corresponding CSAR. Thus, imperative TOSCA runtime engines run these plans to consistently execute management tasks. In contrast to the imperative approach, the *declarative approach* does not require any plans: a declarative TOSCA runtime engine derives the corresponding logic automatically by interpreting the topology template.

4. Integrated modeling and runtime framework

To address the challenges outlined in Section 2, we present an integrated and comprehensive modeling and runtime framework. The framework is completely standards-driven and thus centered around the TOSCA standard, which was discussed in Section 3.4. Fig. 7 provides an overview of the conceptual framework, which covers multiple phases. The initial phase of *artifact discovery*, which is described in Section 4.1 in detail, utilizes a crawling approach to discover and retrieve DevOps artifacts from public repositories. These repositories typically contain artifacts such as scripts, templates, and modules with metadata attached in the form of semi-structured data. The discovery and crawling results are stored in a set of *DevOps knowledge repositories*, which contain the crawled artifacts in the form of structured data using a normalized data format. During the second phase of *artifact transformation* (Section 4.2), the crawled DevOps artifacts of different kinds are packaged and wrapped as TOSCA artifacts to utilize a uniform, standards-based metamodel. The sum of these transformed artifacts are then stored in a *standards-based knowledge base*, which holds TOSCA artifacts. This knowledge base can then be used in the *topology enrichment* phase (Section 4.3) to attach deployment logic to plain application topologies that are modeled in TOSCA. As a result of the enrichment, the application topology becomes deployable by including the deployment logic required to create instances of the application. This happens in the *topology deployment* phase (Section 4.5).

The TOSCA deployment engine used in the deployment phase must deal with different kinds of TOSCA artifacts that were derived from existing DevOps artifacts during the transformation phase, i.e., wrapped DevOps artifacts. Despite the fact they are packaged using a uniform metamodel (TOSCA), which is an efficient means for abstraction and unification at modeling time, the deployment engine needs to know how to invoke the different kinds of artifacts at runtime (invocation mechanism, parameter passing, result retrieval, etc.). To further improve the portability of application topologies, focusing on the previously mentioned runtime aspects, the *artifact APIfication* phase (Section 4.4) aims to generate proper APIs for the TOSCA artifacts that are involved. The kind of APIs must be compatible with the deployment engine to enable the engine the interaction with these APIs, hiding and abstracting the underlying, lower-level technical details of the involved artifacts. The following sections discuss the different aspects and phases of our framework in more detail. As described in the following, we technically compose and integrate various existing works centered around TOSCA and DevOps approaches.

4.1. Artifact discovery

The initial phase of our integrated modeling and runtime framework is the discovery of reusable DevOps artifacts. Therefore, we follow a crawling approach [23] as outlined in Fig. 8. Different kinds of DevOps artifacts are crawled from public repositories containing executables such as Chef cookbooks[17] and Juju charms[18] with metadata attached in the form of semi-structured data. As an example, each Chef cookbook typically owns a `metadata.rb` file,[19] which contains important information on

[17] Chef Supermarket: https://supermarket.chef.io/cookbooks-directory.
[18] Juju Solutions: https://jujucharms.com/solutions.
[19] Chef's `metadata.rb` file: https://docs.chef.io/config_rb_metadata.html.

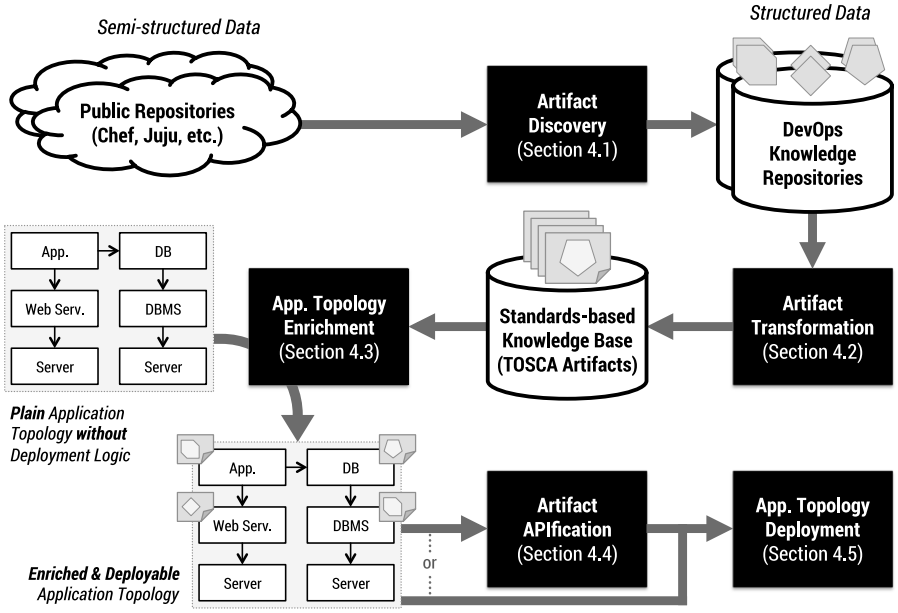

Fig. 7. Framework overview.

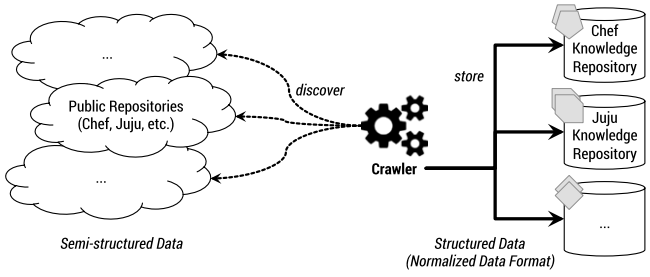

Fig. 8. Knowledge repositories containing different kinds of DevOps artifacts [23].

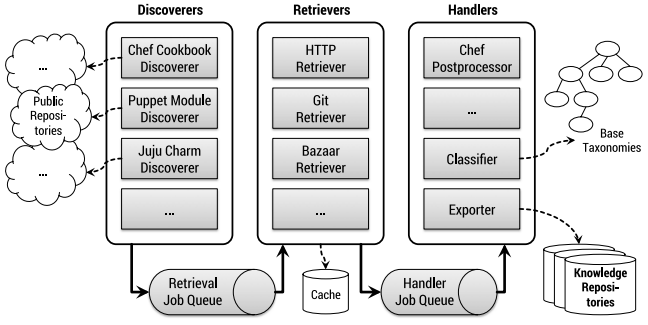

Fig. 9. Architecture overview of DevOps artifact crawler [23].

which platforms the cookbook runs and which input parameters are expected. The crawler retrieves discovered artifacts and stores them in corresponding knowledge repositories using a normalized data format. Technically, in our current prototype implementation, YAML[20] files are stored using a proper directory structure in Git repositories.

Fig. 9 outlines an architecture overview of the DevOps artifact crawler. The architecture is completely modular, decoupled, and thus extensible. It comprises three kinds of modules that are integrated using a plugin mechanism, namely (i) discoverers, (ii) retrievers, and (iii) handlers. A discoverer module such as the *Chef cookbook discoverer* is a specialized piece of code to find existing artifacts (Chef cookbooks) in public repositories. In case an artifact is discovered, a retrieval job is put into the *retrieval job queue*. This job is then processed by a corresponding retriever module depending on where the artifact is stored. If, for example, the discovered artifact is stored in a Git repository, the *Git retriever* is utilized. The retrieved files may be cached to avoid retrieving the same artifact multiple times. This helps to save time and traffic costs. After retrieving an artifact, a handler job is put into the *handler job queue* for postprocessing purposes. Depending on the metadata attached to the handler job, the crawler determines which handlers have to be invoked in which order. At least one handler has to deal with the postprocessing of the artifact. Assuming the discovered artifact is a Chef cookbook, the *Chef postprocessor* is used to process and normalize Chef-specific metadata attached to the artifact. In addition, the *classifier* may be utilized in conjunction with base taxonomies in the background to automatically determine how the given artifact can be classified. Fig. 10 shows an example for a base taxonomy to be used for classifying middleware artifacts. Similarly, further base taxonomies are defined for infrastructure entities such as operating systems as well as 'DevOpsware' entities such as DevOps tooling (Chef, Juju, etc.) for automating the deployment and management of applications. These taxonomies are used to identify an appropriate category for the given artifact, e.g., using document classification techniques [24]. Finally, the *exporter* is used to store the artifact and all related metadata using a normalized data format in corresponding knowledge repositories as outlined in Fig. 8. The following YAML snippet outlines an example for how a Chef cookbook can be represented in the knowledge repository:

```
Middleware/Database/Relational/MySQL/MySQL Chef Cookbook:
  properties:
    versions:
    - "5.0"
    - "5.1"
    - "5.5"
    - "5.6"
    requires:
      all_of:
      - DevOpsware/Deployment/Chef
      - ...
    hosted_on:
```

20 YAML Ain't Markup Language: http://yaml.org.

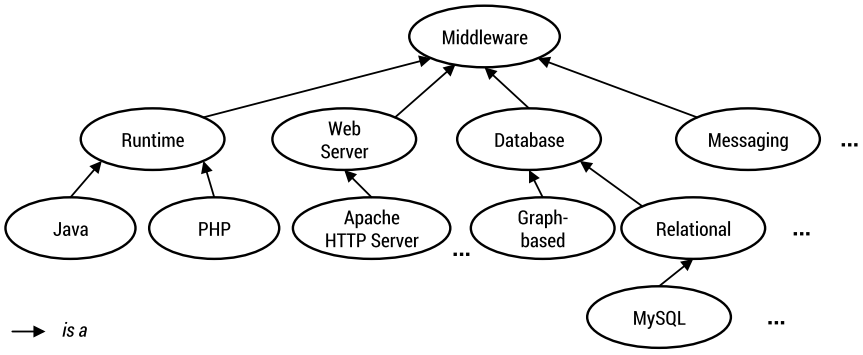

Fig. 10. Base taxonomy for middleware artifacts [23].

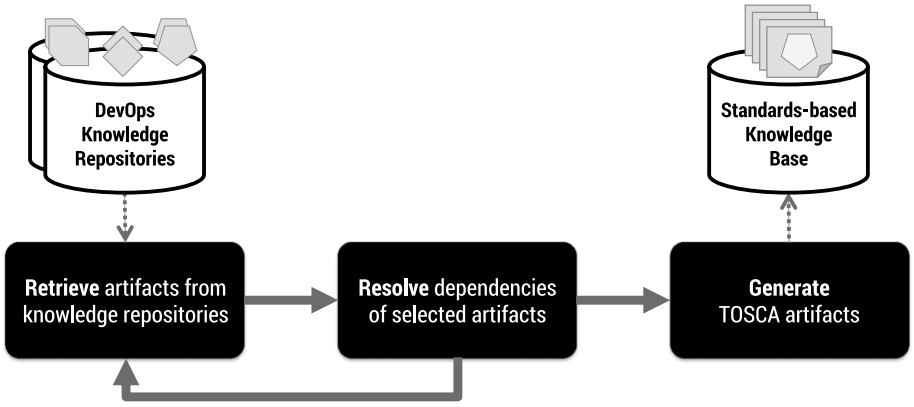

Fig. 11. Overview of DevOps artifact transformation toward TOSCA.

```
one_of:
- Infrastructure/Operating System/Linux/Ubuntu
- Infrastructure/Operating System/Linux/Debian
- Infrastructure/Operating System/Linux/RHEL
- Infrastructure/Operating System/Linux/CentOS
- Infrastructure/Operating System/Linux/Fedora
- ...
```

Each artifact is identified by a qualified name including a classification for the solution it provides such as `Middleware/.../MySQL Chef Cookbook`. Properties are attached to indicate the context of its potential usage. Dependencies are expressed using the `requires` property, which is represented as a list of references pointing to other artifacts or solutions stored in knowledge repositories. Furthermore, the `hosted_on` property holds a list of references to infrastructure entities such as operating systems that can be utilized to host the solution provided by the given artifact such as a MySQL database server. After the crawled artifacts are stored as structured data in corresponding knowledge repositories, they need to be transformed toward TOSCA in order to establish a consolidated, standards-based knowledge base comprising TOSCA artifacts. This transformation is discussed in the following Section 4.2.

4.2. Artifact transformation

In order to transform arbitrary DevOps artifacts of different kinds (considering the classification presented in Section 3.1) toward TOSCA to make them usable and composable in a seamless and interoperable manner, we present a transformation framework and method as shown in Fig. 11. Artifacts are retrieved from DevOps knowledge repositories (e.g., Chef knowledge repository). Then, all dependencies of the retrieved artifacts are checked automatically. As a result, further artifacts (i.e., further Chef cookbooks) may have to be retrieved to resolve these dependencies. These artifacts may have further dependencies that need to be resolved afterward. Therefore, the process iterates these two steps until all dependencies are resolved. In the next step, TOSCA artifacts (i.e., node types and relationship types as described in Section 3.4) are generated based on the retrieved artifacts to store them in the consolidated, standards-based knowledge base. This is the technical transformation of DevOps artifacts toward TOSCA. The actual transformation logic is highly specific to the individual DevOps approaches. In the following Sections 4.2.1 and 4.2.2, we explain the technical transformation of node-centric artifacts (NCAs) and environment-centric artifacts (ECAs) applied to Chef and Juju in more detail.

4.2.1. Transformation of NCAs applied to Chef cookbooks

In this section we present the concepts of a technical transformation of Chef cookbooks that bundle Chef recipes as outlined in Section 3.2. The goal is to generate TOSCA node types for existing Chef cookbooks. We provide two different alternatives to perform the transformation of Chef cookbooks: (i) the *fine-grained transformation* and (ii) the *coarse-grained transformation*. As shown in Fig. 12 for the fine-grained approach, a node type is created for each selected cookbook. Each recipe is attached to the generated node type as a TOSCA implementation artifact that implements a certain operation as discussed in [25]. A single node type capability is generated for each cookbook consisting of the cookbook name. Each cookbook owns a `metadata.rb` file that contains information such as name, version, and maintainer of the cookbook. Moreover, all dependencies on other cookbooks are defined inside the `metadata.rb` file. For each dependency a corresponding requirement is generated. Relationship types are not required to be generated because Chef does not support the expression of relationships explicitly as discussed in Section 3.1 in the context of node-centric artifacts. Thus, a generic *depends*

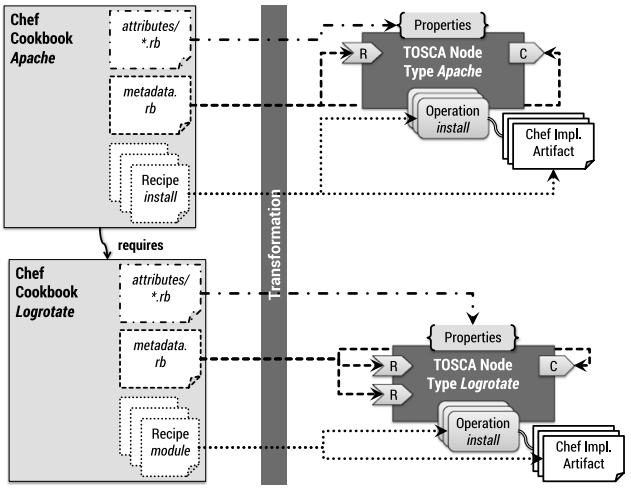

Fig. 12. Fine-grained transformation: one node type per Chef cookbook.

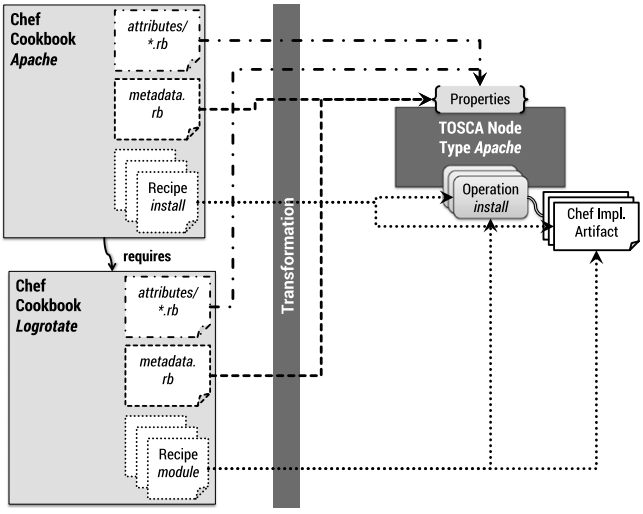

Fig. 13. Coarse-grained transformation: dependencies packaged as implementation artifacts.

on relationship can be used to wire nodes derived from node types that were generated from cookbooks. Finally, the attributes definitions (`*.rb` files) stored in the `attributes` sub-directory of a cookbook is used to derive the node type properties. These are used to configure node templates derived from the node type, influencing the execution of the operations attached to the node type.

As an alternative, the coarse-grained transformation of Chef cookbooks is shown in Fig. 13: node types are only generated for the given cookbooks. Cookbooks that have dependencies among each other are packaged inline as implementation artifacts. Consequently, the dependencies do not appear as separate node types, so a single node type may wrap multiple cookbooks. The main motivation for this is the fact that cookbooks and their dependencies may be very fine-grained. For instance, the *Apache HTTP server* cookbook[21] depends on the cookbooks *logrotate*, *iptables*, and *pacman*. These are operating system-level software packages that are not supposed to appear in the application topology as individual nodes (i.e., node templates). This is because an application topology typically models more coarse-grained

21 Apache cookbook: https://supermarket.getchef.com/cookbooks/apache2.

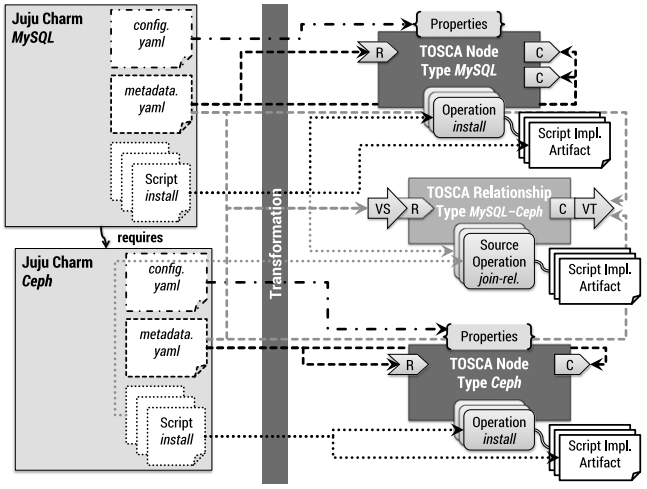

Fig. 14. Transformation of Juju charms.

middleware and application components such as Web servers and databases. The coarse-grained transformation skips the creation of lower-level node types, avoiding a pollution of topologies. We can also combine the two transformation approaches by specifying which cookbooks are too low-level to be exposed as separate node types. Consequently, these are transparently packaged as implementation artifacts without generating individual node types.

Other node-centric solutions and their artifacts following the classification of node-centric artifacts (NCAs) as discussed in Section 3.1 such as Puppet [26] and CFEngine [27] work very similar to how Chef works. Consequently, the technical transformations discussed in this section can be transferred to be applied to these artifacts, too. In the following Section 4.2.2, we discuss how Juju charms as environment-centric artifacts (ECAs) can be technically transformed to produce TOSCA node types and relationship types.

4.2.2. Transformation of ECAs applied to Juju charms

The transformation of environment-centric artifacts such as Juju charms is slightly more complex than what we discussed before for Chef cookbooks. This is because relationships have to be considered as separate entities with individual operations and implementation artifacts to link different nodes in an application topology. In case of Juju, a separate node type is generated for each charm (Fig. 14). The operations and their implementation artifacts are derived from the charm's lifecycle scripts. These lifecycle scripts do not only cover the installation and configuration of application components, but also the required scaling mechanisms, i.e., adding, wiring, and removing instances accordingly. Each charm owns a `metadata.yaml` file describing its interfaces that it requires and provides. Based on these interface descriptions, requirements and capabilities are attached to the generated node types in the TOSCA type definitions. Furthermore, relationship types are generated on the TOSCA side for each requirement-capability pair on the Juju side because charms typically contain further lifecycle scripts that need to be executed when a relationship is established or changed in order to target the lifecycle of the relationship. Lifecycle scripts that need to be executed on the source node are mapped to source operations in the generated relationship type in TOSCA; scripts that are supposed to be executed on the target node result in target operations. Each relationship type defines the corresponding requirement (exposed by a source node) as valid source and the matching capability (exposed by a target node)

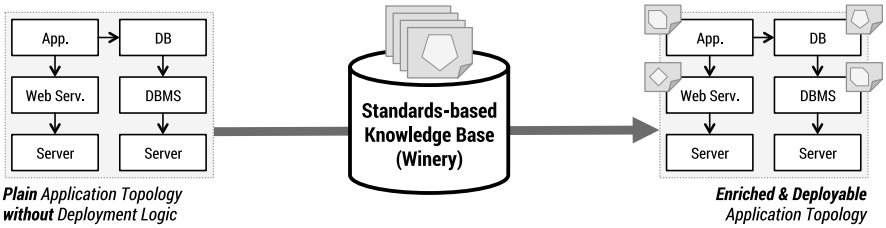

Fig. 15. Making plain application topologies deployable by means of enrichment.

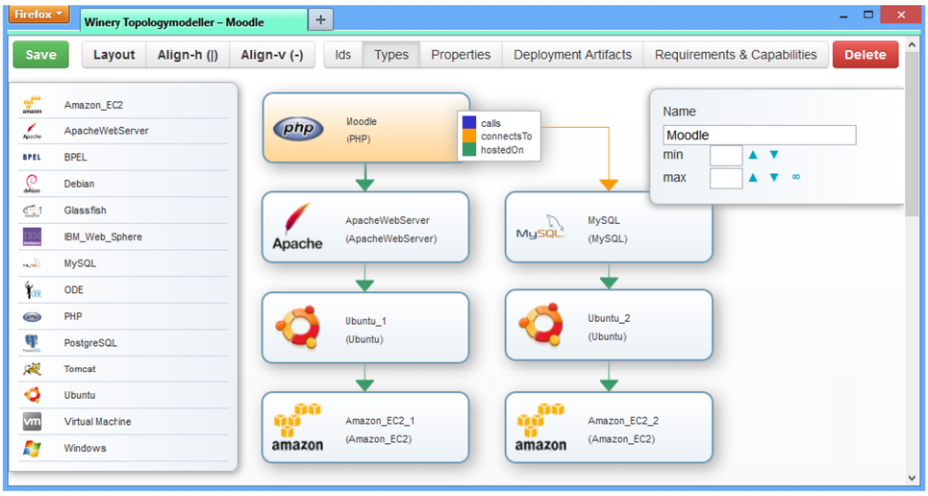

Fig. 16. Screenshot of TOSCA modeling tool *Winery* [28].

as valid target. Consequently, the generated relationship type can be used to connect two nodes by matching the requirement of the source node with the capability of the target node. Similar to Chef's attributes the `config.yaml` file of each charm provides a description of its configuration options in the form of key-value pair definitions including the data type of the value and optionally a default value. These are used to attach corresponding property definitions to the generated types. The discussed concepts to transform Juju charms to TOSCA types may be transferred to other environment-centric artifacts such as Amazon CloudFormation templates or OpenStack Heat templates.

4.3. Topology enrichment

The previous phases of the presented framework focus on how executable DevOps artifacts of different kinds can be discovered, stored in DevOps knowledge repositories, and how they are transformed into a standards-based descriptions using TOSCA as uniform metamodel to store them in a standards-based knowledge base. This knowledge base contains various TOSCA node types and relationship types including executable TOSCA artifacts to deploy them. To ease the composition of these types toward deployable application topologies that contain all artifacts required to deploy application instances, the *topology enrichment* phase is utilized to automatically retrieve and package appropriate TOSCA artifacts. These are stored in the standards-based knowledge base as deployment packages that can be executed to deploy components of an application and wire them. The approach is shown in Fig. 15 and enables developers to model an application as a *plain TOSCA topology model* that does not contain any deployment logic at all. The plain topology model is then enriched by executable TOSCA artifacts to eventually produce a deployable, self-contained package.

To recap, TOSCA enables the modeling of application topologies that consist of node templates representing the components of an application and relationship templates representing the relations between these components. Both are typed by referring to a corresponding node type or relationship type. Thus, based on the specified types of node and relationship templates in the topology model, the corresponding node and relationship type definitions including all associated deployment executables can be retrieved from the knowledge base by looking them up based on their name. The topology model as well as all retrieved type definitions including their executables are packaged as a CSAR, which contains all required information to deploy each component and relation of the application topology.

We implemented this approach by extending the open-source TOSCA modeling tool *Winery* [28]. This tool supports the modeling of plain topology models by a graphical editor, which is shown in Fig. 16. To enrich these models, they get analyzed fully automatically in terms of employed node and relationship types, which are then fetched from the Winery repository that serves as the knowledge base. Finally, the topology model as well as the retrieved artifacts and definitions are exported in the CSAR format containing all required files and metadata. The execution of these CSARs that comprise deployable application topologies is described in the following sections.

4.4. Artifact APIfication

To improve the interoperability and seamless integration of different kinds of DevOps artifacts when combining them in an application topology for deployment automation purposes, we performed the transformation toward TOSCA as a uniform metamodel as discussed in Section 4.2. This approach significantly helped at modeling time to enrich plain application topologies (without deployment logic) as described in Section 4.3 in order to make them eventually deployable. However, the TOSCA deployment engine used in the deployment phase later on must deal with different kinds of TOSCA artifacts that were derived

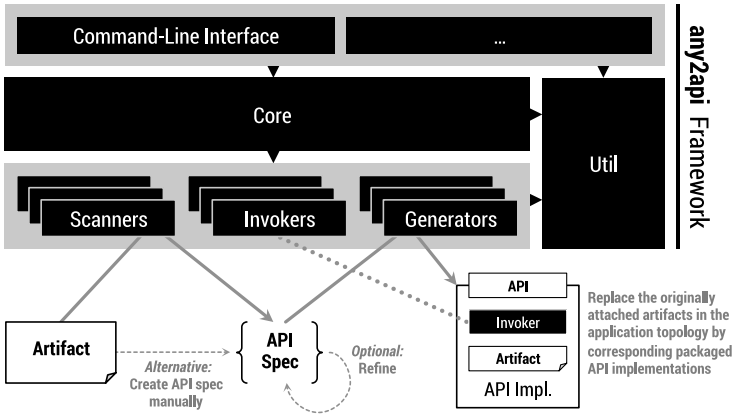

Fig. 17. Architecture overview of ANY2API.

Param. Name	Data Type	Default	Param. Mapping
version	string	"5.1"	CHEF_ATTR:mysql/version
port	number	3306	CHEF_ATTR:mysql/port
...			
invoker_config	*json_object*	*null*	*ENV:INVOKER_CONFIG*

Result Name	Data Type	Result Mapping
root_pw	string	FILE:/tmp/mysql-password
errors	string	STDERR
logs	string	STDOUT
...		

Fig. 18. API spec for an executable artifact to deploy a MySQL database server.

from existing DevOps artifacts during the transformation phase. The packaging of these artifacts using a uniform metamodel (TOSCA) does not fully address runtime-related challenges. For instance, the deployment engine needs to know how to invoke different kinds of artifacts (invocation mechanism, parameter passing, result retrieval, etc.). Therefore, we include the *artifact APIfication* phase in our framework to wrap the artifacts and expose their functionality transparently through a generated API (e.g., REST/HTTP or SOAP/HTTP), which is understood by the utilized deployment engine. The type of the generated APIs can be selected depending on the capabilities of the deployment engine. As an example, OpenTOSCA [29] utilizes BPEL workflows to run deployment plans (a subset of management plans). Therefore, HTTP-based APIs utilizing SOAP [30] and WSDL [31] would be appropriate to enable the interaction between plans and the underlying artifacts. The APIfication is not necessarily required and therefore an *optional* building block of our framework in case the utilized deployment engine can deal with all kinds of artifacts included in a certain application topology. However, the APIfication of artifacts helps to keep the deployment engine maintainable without the overhead to support a huge variety of kinds of artifacts.

Fig. 17 outlines the architecture and logical workflow of the APIfication framework ANY2API [13] that we are developing as open-source project[22] to generate APIs for arbitrary artifacts. The framework is fully modular and extensible. There are three kinds of modules:

- A scanner module (e.g., a *Chef scanner*) analyzes artifacts of a certain kind (e.g., Chef cookbooks) to automatically generate an API specification *(API spec)*, which technically specifies how to run a given artifact (invocation command, input parameters, results, etc.).
- An invoker module (e.g., a *Chef invoker*) provides and encapsulates the invocation logic for a certain kind of artifact.
- A generator module (e.g., a *REST API generator*) implements the functionality to generate and package an API implementation based on a given API spec. The packaging format could be a Docker container or a Vagrant virtual machine.

As shown in Fig. 17, an API spec is either produced by a scanner or it can be created manually, e.g., in case there is no appropriate scanner for a given artifact available. An API spec defines how to run a given artifact, which input parameters it expects, which results it produces, etc. Fig. 18 shows an example for both a parameters schema and a results schema as they appear in an API spec. This example refers to an executable artifact such as a Chef cookbook to deploy a MySQL database server. For each parameter (e.g., *version* and *port*) and result (e.g., *root_pw*), its name, data type, and mapping are specified. The mapping definition for a parameter is used for pushing the parameter value to a certain channel as expected by the artifact (e.g., a Chef cookbook attribute using the *CHEF_ATTR* prefix or a Shell environment variable using the *ENV* prefix). For a result, the mapping definition is used to collect the result value produced by the artifact through a certain channel (e.g., the console output using the *STDOUT* prefix or a file using the *FILE* prefix). In addition, a default value may be defined for parameters, which is used as fallback in case no value is given when invoking the generated API implementation. An example is the common default port 3306 for a MySQL database server. Optionally, an automatically produced API spec can be refined manually. Finally, a generator creates an API endpoint (e.g., a REST API) and bundles it together with the executable and the invoker that is required to run the executable. Fig. 19 outlines the structure of such a packaged API implementation. The packaging format could be, e.g., a Docker container [16] or a Vagrant VM [32].

As a result of the APIfication, the utilized deployment engine such as OpenTOSCA does not have to deal with the invocation of the packaged artifact, e.g., the MySQL Chef cookbook directly. The engine only needs to understand the utilized packaging format, e.g., Docker containers, to run the API implementation and to interact with the exposed API endpoint, thereby invoking the packaged artifact. Consequently, the packaged API implementation appears as a black box to the deployment engine, so it does not have to know anything about the internals of the API implementation and the wrapped artifact. To improve efficiency and reduce resource consumption at runtime, several artifacts of different kinds can be packaged together with the required invokers in a single API implementation, as long as they are exposed through the same kind of API (e.g., SOAP/HTTP). By

22 APIfication framework ANY2API: http://any2api.org.

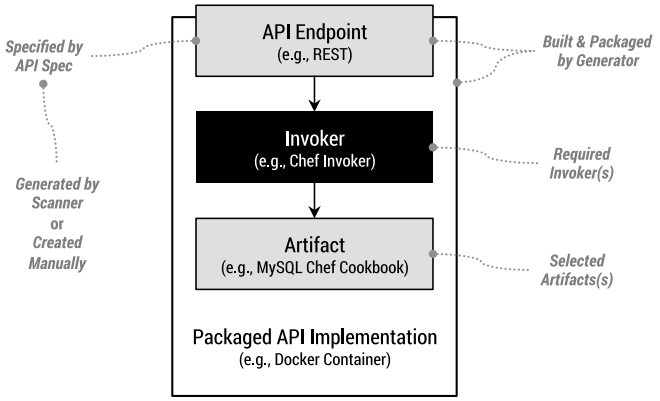

Fig. 19. Example for generated and packaged API implementation.

following this approach, it is not required to generate and run individual API implementations for each artifact included in the application topology. In case the artifact APIfication is used, the generated API implementations would replace the originally attached artifacts in the application topology.

4.5. Topology deployment

As the APIfication approach discussed in Section 4.4 bridges the gap between modeling time and runtime, the final phase of our framework targets the actual deployment of the application topology at runtime. We are using TOSCA as a standardized and uniform metamodel for efficiently integrating different kinds of DevOps automation approaches at modeling time and runtime. Consequently, we utilize the TOSCA-based deployment engine *OpenTOSCA* [29] to create and maintain instances of a given application topology. TOSCA proposes two alternative flavors of processing packaged application topologies, i.e., CSARs: either (i) the engine is able to declaratively process the structure of an application topology and derive the actions required to deploy the application, or (ii) an explicit deployment plan is provided, which imperatively tells the engine the ordered steps to be performed to make the deployment happen. OpenTOSCA supports both flavors: if a deployment plan (e.g., in the form of a BPEL workflow) is attached to the topology, it will be used to run the deployment. If no deployment plan is explicitly defined, the OpenTOSCA plan generator presented in previous work [33] is utilized to automatically generate a deployment plan for the given topology. Based on predefined lifecycle interfaces such as the TOSCA lifecycle interface [19], the generator interprets the semantics of the associated artifacts to trigger their execution accordingly. For example, if a node type provides an interface owning the *install* operation, which is associated with a Chef cookbook, the plan generator knows that this artifact has to be executed to install the corresponding component.

Fig. 20 outlines the key parts of OpenTOSCA's architecture that are important when deploying an instance of a given application topology by executing a deployment plan. As mentioned previously, the deployment plan is either packaged with the topology or it is generated on demand. This plan interacts with the operation invoker through a uniform interface to invoke implementation artifacts (IAs) that are attached to operations in the TOSCA application topology. Technically, the uniform interface is implemented as a SOAP/HTTP interface because deployment plans are currently developed in BPEL. However, this is an implementation detail of the current prototype. The operation invoker itself comprises plugins to enable the interaction with different kinds of IAs. Some IAs such as the *Cloud Provider B Management IA* shown in Fig. 20 expose a REST/HTTP interface, which can be used immediately by the REST plugin of the operation invoker. Other IAs such as the *Apache Installation Script IA* shown in Fig. 20 are plain script artifacts that do not expose any API to be invoked immediately. There is a lot more complexity involved in dealing with such artifacts: scripts need to be copied to the target machine (e.g., using SSH), the script execution environment needs to be available on the target machine (e.g., by installing the Chef runtime), all input parameters have to be passed before its invocation, and finally the invocation results have to be collected. OpenTOSCA encapsulates these aspects in a separate building block, namely the artifact manager [34]. The operation invoker's artifact plugin is in charge of interacting with the artifact manager. As outlined in Fig. 21, the artifact plugin triggers artifact executions and continuously polls the status of running instances. Specialized runners enable the abstraction of invoking very different kinds of artifacts through a uniform interface by encapsulating corresponding runtime knowledge. As an example, there may be a *Juju runner* as well as a *Chef runner* implemented and placed in the runner registry. These are used to run corresponding artifacts. When an artifact invocation is triggered, the associated files such as scripts and input data as well as host information such as the IP address and SSH credentials are stored in the artifact registry and the host registry. To prepare the actual execution, the corresponding runner is retrieved from the runner registry and an artifact instance is built and stored in the instance registry. Finally, the artifact manager connects to the target machine, e.g., a VM hosted at a Cloud provider using a corresponding protocol such as SSH or WinRM[23] to perform the execution on the target VM. Besides the uniform invocation interface, the artifact manager provides improved robustness and reliability for complex deployment processes: it acts as a middleware to monitor and manage artifact executions independent of the underlying approaches such as Chef or Juju.

It is important to point out that the artifact manager is not inherently required. In case of consequently following the artifact APIfication approach discussed in Section 4.4, the functionality of all IAs are exposed through proper APIs (e.g., REST/HTTP or SOAP/HTTP). Consequently, the basic plugins of the operation invoker (REST plugin and SOAP plugin) are sufficient to interact with all artifacts involved. A complex and centralized middleware such as the artifact manager is not required in this case, keeping the deployment engine (OpenTOSCA) focused on the actual orchestration logic without considering lower-level, technical details.

5. Validation and evaluation

In order to validate our integrated modeling and runtime framework as implemented and discussed in Section 4, we conducted a comprehensive case study, which is described in the following Section 5.1. Furthermore, as we consider the transformation of arbitrary DevOps artifacts toward TOSCA (Section 4.2) as a major building block of our framework, we quantitatively measured the performance for generating corresponding TOSCA node types and relationship types. This is to evaluate the transformation performance. The evaluation is described in the following Section 5.2.

5.1. Implementation and case study

The case study to validate our integrated modeling and runtime framework is based on the Web shop application introduced in

[23] Windows Remote Management (WinRM): http://goo.gl/O5qMza.

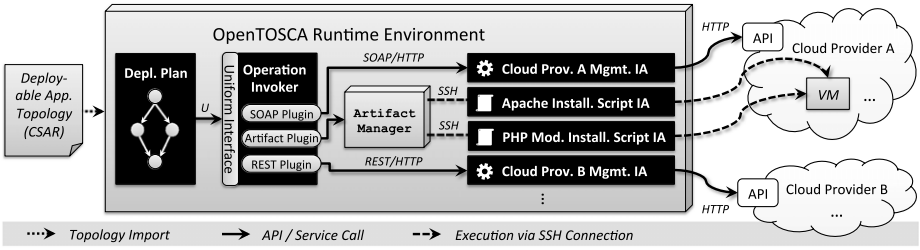

Fig. 20. OpenTOSCA architecture with operation invoker and artifact manager.

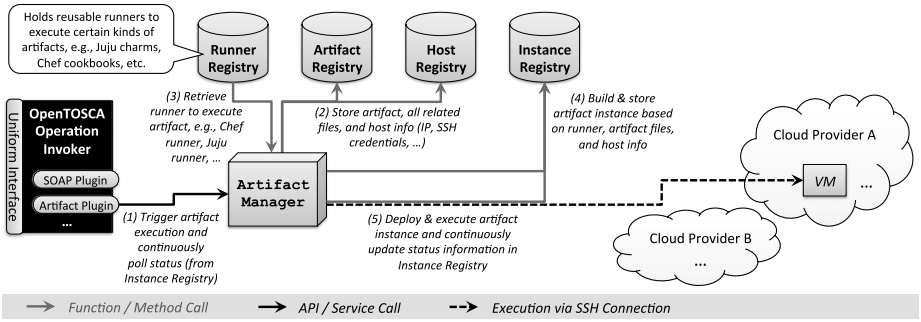

Fig. 21. Architecture overview of artifact manager.

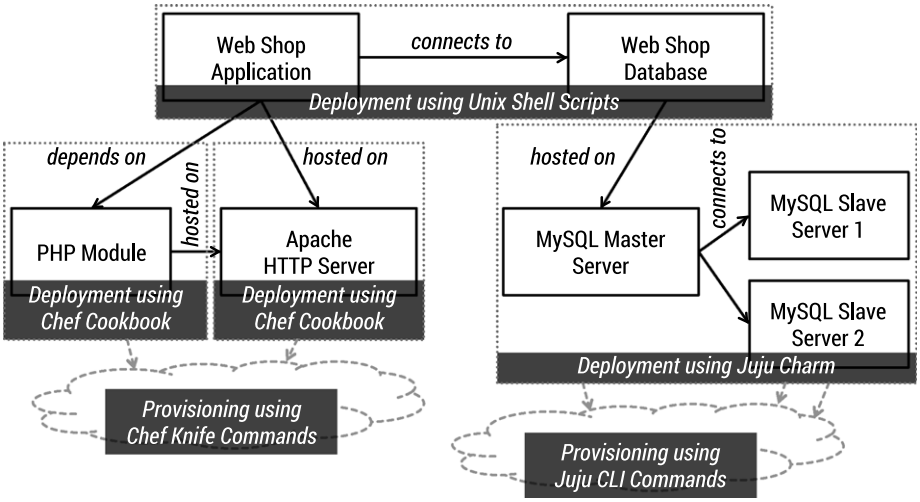

Fig. 22. Web shop deployment using different kinds of artifacts.

Section 2 as motivating scenario of our work. Therefore, we utilize a prototype implementation of the framework, which is based on a composition of open-source building blocks to provide an end-to-end toolchain considering the framework's various phases. To follow the phases provided by the framework, we initially populated the *Juju knowledge repository* and the *Chef knowledge repository* by utilizing the artifact discovery and crawling approach presented in Section 4.1. This is to have DevOps artifacts available to meet the middleware requirements of the Web shop application topology outlined as motivating scenario in Section 2. More specifically, we use the Apache cookbook[24] (Chef) and the PHP cookbook[25] (Chef) as shown in Fig. 22. For the database part we cannot utilize the MySQL cookbook[26] (Chef) because it does not support a distributed master/slave deployment out of the box. This is why we choose the MySQL charm[27] (Juju) to deploy the MySQL database server in a distributed manner. The deployment of the application-specific parts (Web shop application, database, and the application-database connection) are implemented using custom Unix shell scripts. The underlying infrastructure is provisioned by associated tooling such as Chef knife and the Juju command-line interface tool. Next, we perform the transformation [8] of the selected artifacts toward TOSCA as described in Section 4.2. The transformation processes for Juju charms and Chef cookbooks as discussed in Sections 4.2.1 and 4.2.2 are technically implemented as Java executables (JAR files). As a result, we store the generated TOSCA types in Winery [28], which acts as a standards-based knowledge base. Table 1 shows which TOSCA node types and relationship types were derived from the originally selected artifacts. It is further outlined which requirements are attached. In addition, Table 2 shows the attached capabilities and operations for each generated TOSCA type.

[24] Apache cookbook: https://supermarket.getchef.com/cookbooks/apache2.
[25] PHP cookbook: https://supermarket.getchef.com/cookbooks/php.
[26] MySQL cookbook: https://supermarket.getchef.com/cookbooks/mysql.
[27] MySQL charm: http://jujucharms.com/mysql.

Table 1
Attached requirements of generated TOSCA types (from transformation).

Original artifact	Generated TOSCA type	Attached requirements
Web shop app. deployment script (Shell)	Node type: Web shop application	– Web server – PHP runtime – Web shop database
Web shop database deployment script (Shell)	Node type: Web shop database	– MySQL database server
Web shop database connection script (Shell)	Rel. type: Web shop app. *connects to* database	– Web shop database *(valid source)*
Apache cookbook (Chef)	Node type: Apache HTTP server	– *Operating system*
PHP cookbook (Chef)	Node type: PHP module	– *Operating system* – Apache HTTP server *(derived from Web server)*
MySQL charm (Juju)	Node type: MySQL server	– *Cloud infrastructure* – MySQL master server *(only for slave servers)*
	Rel. type: MySQL master *connects to* slave	– MySQL master server *(valid source)*
…		

Table 2
Attached capabilities and operations of generated TOSCA types.

Generated TOSCA types	Attached capability	Attached operations
Node type: Web shop application	*(none)*	– Deploy
Node type: Web shop database	Web shop database	– Deploy – Backup-db – Restore-db
Rel. type: Web shop app. *connects to* database	Web shop database *(valid target)*	– Connect
Node type: Apache HTTP server	Web server	– Deploy
Node type: PHP module	PHP runtime	– Deploy
Node type: MySQL server	MySQL database server	– Deploy
Rel. type: MySQL master *connects to* slave	MySQL database server *(valid target)*	– Connect – Disconnect
…		

After generating TOSCA artifacts and storing them in the knowledge base, we use Winery [28] as a TOSCA modeling tool to build a plain topology template for the Web shop application without any implementation artifacts attached. Then, we perform the topology enrichment as described in Section 4.3 and provided by Winery [28] to make the application topology deployable by attaching corresponding artifacts from the knowledge base. Finally, OpenTOSCA [29] is utilized in conjunction with the artifact manager [34] as a deployment engine as discussed in Section 4.5 to create application instances.

5.2. Transformation performance

The transformation of arbitrary DevOps artifacts toward TOSCA is a key enabler for the integrated modeling and runtime framework presented in Section 4. In order to quantitatively evaluate the transformation performance, we run the transformation for each artifact (Apache cookbook, PHP cookbook, and MySQL charm) *ten* times. The transformations were executed on a virtual machine (1 virtual CPU clocked at 2.8 GHz, 2 GB of memory) on top of the VirtualBox hypervisor, running a minimalist installation of the Ubuntu OS, version 14.04. The average transformation time and the standard deviation for each artifact based on our measurements are shown in Table 3. Besides the actual model transformation the measured time includes the retrieval of the artifacts and all their dependencies from code repositories such as Git. Moreover, the packaging of the generated artifacts into Cloud Service Archives (CSARs) is also included in the measured time. The transformation of cookbooks should be faster because no relationship types are created. However, cookbook dependencies have to be retrieved separately because they are distributed across different repositories. This is why the transformation times are in the same order of magnitude. For the application-specific parts (Web shop application and database) we created additional types to cover the complete Web shop topology. In addition, two generic relationship types are available: *hosted on* and *depends on*. These are used to logically wire nodes in a topology if no specific implementation or operation is required.

Table 3
Measurements regarding transformation performance.

	Apache cookbook	PHP cookbook	MySQL charm
Generated Node types:	Apache HTTP server	PHP module	MySQL server
Generated Rel. types:	*(none)*	*(none)*	MySQL master *connects to* MySQL slave
Average Transformation Time:	31.6 s	25.8 s	26.5 s
Standard Deviation:	0.7 s	0.5 s	0.2 s

6. Related work

Our work and the presented framework in particular are strongly based on TOSCA as a standardized metamodel for application topologies. There are several alternatives to TOSCA such as CloudML [35] and Blueprints [36]. Moreover, pattern-based deployment approaches [37] utilize a proprietary metamodel based on Chef to transform patterns to deployable artifacts. Because TOSCA is an emerging standard and tooling support is improving as well, we decided to use TOSCA as an interoperable, intermediate metamodel. Further orchestration approaches are originating in the DevOps community such as Amazon OpsWorks [38], Terraform,[28] Kubernetes,[29] and DevOpSlang [39]. These approaches significantly simplify the usage of node-centric artifacts such as containers, VM images, and configuration definitions in an environment-centric manner as outlined in Section 3.1. This is achieved by providing mechanisms and metamodels to express

[28] Terraform: http://www.terraform.io.
[29] Kubernetes: http://kubernetes.io.

bundles and relations between node-centric entities, thereby enabling the description of scalable application topologies. Technically, TOSCA can be used as an overarching, technology-agnostic metamodel as proposed in this paper. This does not imply the need for a native TOSCA runtime environment to deploy and manage corresponding application topology models, although it is an option. Alternatively, TOSCA-based application topologies can be transformed to technology-specific models, which can then be deployed using specific solutions such as Terraform or Kubernetes.

In addition to pure deployment automation approaches that are mainly targeting the level of infrastructure-as-a-service, different platform-as-a-service (PaaS) frameworks and solutions are emerging such as Heroku,[30] IBM Bluemix,[31] and Cloud Foundry.[32] These enable the packaging of middleware and application components on a higher level, mostly abstracting away infrastructure aspects such as virtual servers and networking. Consequently, artifacts that are packaged this way can be considered as environment-centric artifacts. Corresponding transformation concepts discussed in this paper may be transferred to these kinds of artifacts.

We utilized a crawling approach to discover and capture DevOps artifacts in an automated manner, in order to store them in corresponding knowledge repositories. Related works [40,41] propose approaches to implement general-purpose crawlers to capture data from the Web. However, especially unstructured data is hard to capture in structured knowledge repositories automatically. Thus, humans may be instrumented as social compute units [42] to discover and capture DevOps knowledge using crowdsourcing approaches [43].

Model transformations play a key role in different domains such as model-driven architectures and model-driven development [44]. According to [45], three major transformation strategies can be distinguished: (i) the *direct model manipulation* is a way to immediately modify an existing source model in order to transform it to the target model. (ii) Alternatively, an *intermediate representation* can be used to render models in a general-purpose markup language such as XML. (iii) A *transformation language* providing constructs to express and apply transformations may be used to drive such model transformations. In our work we focus on the intermediate representation of transformed models based on TOSCA and rendered in XML.

7. Conclusion

Our work is primarily motivated by the fact that a huge number and variety of reusable DevOps artifacts are shared as open-source software such as Chef cookbooks and Juju charms. However, these artifacts are bound to specific tools such as Chef, Juju, or Docker. This makes it hard to combine and integrate artifacts of different kinds for establishing automated deployment processes for Cloud applications that consist of various middleware and application components. To tackle this issue we provide an integrated, standards-based modeling and runtime framework. Therefore, we utilize TOSCA as an emerging standard to base our work on a uniform metamodel. This is to enable the seamless and interoperable orchestration of arbitrary artifacts. As a foundation for our work we presented an initial classification of DevOps artifacts, distinguishing between node-centric and environment-centric artifacts. Besides the discovery and APIfication of artifacts as well as the enrichment and deployment of application topologies, the transformation of different kinds of DevOps artifacts to generate TOSCA artifacts is a major building block of the framework. The resulting TOSCA node types and relationship types can then be used to create and enrich topology models for Cloud applications to make them eventually deployable. We created a prototype implementation based on open-source building blocks to cover all phases of the integrated modeling and runtime framework. Moreover, we conducted a comprehensive case study based on the motivating scenario of our work in order to validate our approach. Because the artifact transformation is the key enabler for the whole approach, we performed an evaluation to measure and analyze the transformation performance.

In terms of future work we plan to refine our proposed classification of DevOps artifacts by considering and evaluating further conceptual differences. Moreover, we plan to implement further technical transformations based on our framework to extend the validation and evaluation of our approach. Concerning the framework's implementation and the integration of the various building blocks involved, our goal is to make the prototype more mature to create a better user experience.

Acknowledgments

This work was partially funded by the BMWi project CloudCycle (01MD11023) and the DFG project SitOPT (610872).

References

[1] M. Hüttermann, DevOps for Developers, Apress, 2012.
[2] J. Humble, J. Molesky, Why enterprises must adopt DevOps to enable continuous delivery, Cutter IT J., 24.
[3] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addison-Wesley Professional, 2010.
[4] S. Nelson-Smith, Test-Driven Infrastructure with Chef, O'Reilly Media, Inc., 2013.
[5] N. Sabharwal, M. Wadhwa, Automation Through Chef Opscode: A Hands-on Approach to Chef, Apress, 2014.
[6] P. Mell, T. Grance, The NIST definition of cloud computing, National Institute of Standards and Technology.
[7] OASIS, Topology and orchestration specification for cloud applications (TOSCA) Version 1.0, Committee Specification 01, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.
[8] J. Wettinger, U. Breitenbücher, F. Leymann, Standards-based DevOps automation and integration using TOSCA, in: Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC), 2014.
[9] M. Taylor, S. Vargo, Learning Chef: A Guide to Configuration Management and Automation, O'Reilly Media, 2014.
[10] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, TOSCA: Portable automated deployment and management of cloud applications, in: Advanced Web Services, Springer, 2014, pp. 527–549.
[11] S. O'Mahony, F. Ferraro, The Emergence of governance in an open source community, Acad. Manag. J. 50 (5) (2007) 1079–1106.
[12] S. Sharma, V. Sugumaran, B. Rajagopalan, A framework for creating hybrid-open source software communities, Inf. Syst. J. 12 (1) (2002) 7–25.
[13] J. Wettinger, U. Breitenbücher, F. Leymann, Any2API—automated APIfication, in: Proceedings of the 5th International Conference on Cloud Computing and Services Science, SciTePress, 2015.
[14] S. Soltesz, H. Pötzl, M.E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, in: ACM SIGOPS Operating Systems Review, Vol. 41, 2007, pp. 275–287.
[15] M.J. Scheepers, Virtualization and containerization of application infrastructure: A comparison.
[16] J. Fink, Docker: a Software as a service, operating system-level virtualization framework, Code4Lib J., 25.
[17] S. Günther, M. Haupt, M. Splieth, Utilizing internal domain-specific languages for deployment and maintenance of IT infrastructures, Tech. Rep., Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2010.
[18] K. Pepple, Deploying OpenStack, O'Reilly Media, 2011.
[19] OASIS, Topology and orchestration specification for cloud applications (TOSCA) Primer Version 1.0, 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html.
[20] OASIS, Web services business process execution language (BPEL) Version 2.0, 2007.
[21] OMG, Business process model and notation (BPMN) Version 2.0, 2011.
[22] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, BPMN4TOSCA: A domain-specific language to model management plans for composite applications, in: Business Process Model and Notation, in: Lecture Notes in Business Information Processing, vol. 125, Springer, 2012.

[30] Heroku: https://www.heroku.com.
[31] IBM Bluemix: http://bluemix.net.
[32] Cloud Foundry: http://cloudfoundry.org.

[23] J. Wettinger, V. Andrikopoulos, F. Leymann, Automated capturing and systematic usage of DevOps knowledge for cloud applications, in: Proceedings of the International Conference on Cloud Engineering (IC2E), IEEE Computer Society, 2015.
[24] P. Trinkle, An introduction to unsupervised document classification.
[25] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, T. Spatzier, Integrating configuration management with model-driven cloud management based on TOSCA, in: Proceedings of the 3rd International Conference on Cloud Computing and Services Science, SciTePress, 2013.
[26] J. Loope, Managing Infrastructure with Puppet, O'Reilly Media, Inc., 2011.
[27] D. Zamboni, Learning CFEngine 3: Automated System Administration for Sites of Any Size, O'Reilly Media, Inc., 2012.
[28] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, Winery—a modeling tool for TOSCA-based cloud applications, in: Proceedings of the 11th International Conference on Service-Oriented Computing, in: LNCS, vol. 8274, Springer, 2013.
[29] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner, OpenTOSCA—a runtime for TOSCA-based cloud applications, in: Proceedings of the 11th International Conference on Service-Oriented Computing, in: LNCS, Springer, 2013.
[30] W3C, SOAP Specification, Version 1.2, 2007.
[31] W3C, Web services description language (WSDL), Version 2.0, 2007.
[32] M. Hashimoto, Vagrant: Up and Running, O'Reilly Media, 2013.
[33] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger, Combining declarative and imperative cloud application provisioning based on TOSCA, in: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), IEEE Computer Society, 2014.
[34] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, M. Zimmermann, 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 Services Science, SciTePress, 2014.
[35] A. Rossini, N. Nikolov, D. Romero, J. Domaschka, K. Kritikos, T. Kirkham, A. Solberg, CloudML implementation documentation, Paasage project deliverable, 2014.
[36] M. Papazoglou, W. van den Heuvel, Blueprinting the Cloud, IEEE Internet Comput. 15 (6) (2011) 74–79.
[37] H. Lu, M. Shtern, B. Simmons, M. Smit, M. Litoiu, Pattern-based deployment service for next generation clouds.
[38] T. Rosner, Learning AWS OpsWorks, Packt Publishing Ltd., 2013.
[39] J. Wettinger, U. Breitenbücher, F. Leymann, DevOpSlang—bridging the gap between development and operations, in: Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC), 2014.
[40] P. Boldi, B. Codenotti, M. Santini, S. Vigna, Ubicrawler: A scalable fully distributed web crawler, Softw. - Pract. Exp. 34 (8) (2004) 711–726.
[41] A. Heydon, M. Najork, Mercator: A scalable, extensible web crawler, World Wide Web 2 (4) (1999) 219–229.
[42] S. Dustdar, K. Bhattacharya, The social compute unit, IEEE Internet Comput. 15 (3) (2011) 64–69.
[43] J. Howe, The rise of crowdsourcing, Wired Mag. 14 (6) (2006) 1–4.
[44] A. Kleppe, J. Warmer, W. Bast, MDA Explained—the model driven architecture: practice and promise, 2003.
[45] S. Sendall, W. Kozaczynski, Model transformation the heart and soul of model-driven software development, Tech. Rep., 2003.

Johannes Wettinger is a Research Associate and Ph.D. student at the University of Stuttgart. His research focuses on developing and operating continuously delivered software applications, addressing DevOps collaboration and end-to-end automation. Besides his participation in the EU-funded 4CaaSt project and his teaching activities, he is involved in further projects related to Cloud computing.

Uwe Breitenbücher is a Research Associate and Ph.D. student at the University of Stuttgart. His research vision is to improve Cloud application provisioning and management through automating the application of management patterns. He is part of the CloudCycle project which researches portable Cloud services with guaranteed security and compliance.

Oliver Kopp is a Research Associate and Ph.D. student at the University of Stuttgart. In his thesis, he focuses on distributed transactions and global fault handling in service choreographies. His past projects include the projects Tools4BPEL and COMPAS where compliance-driven models, languages, and architectures for services have been researched. Currently, he is also part of the CloudCycle project.

Frank Leymann is a Full Professor of computer science and Director of the Institute of Architecture of Application Systems at the University of Stuttgart. His research interests include Cloud computing, architecture patterns, workflow and business process management, service oriented computing, and integration technology. His background includes having been IBM Distinguished Engineer and elected member of the IBM Academy of Technology.